DISS. ETH NO. 28592

MODULAR SPECIFICATION AND VERIFICATION OF SECURITY PROPERTIES FOR MAINSTREAM LANGUAGES

A dissertation submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH
(Dr. sc. ETH Zurich)

presented by

MARCO EILERS

M.Sc. in Computer Science,
University of Copenhagen,

born on 2 November 1988
citizen of Germany

accepted on the recommendation of

Prof. Dr. Peter Müller, examiner
Prof. Dr. David Basin, co-examiner
Prof. Dr. Marieke Huisman, co-examiner
Prof. Dr. Toby Murray, co-examiner

2022

Abstract

Since software systems increasingly govern both vital infrastructure and people's daily lives, ensuring their safety and security is vital. Where standard measures like testing are insufficient, deductive verification can be used to mathematically prove software properties once and for all. In the past decades, there has been huge progress in the design and automation of verification techniques, to the point where there are multiple automated verification tools able to prove trace properties for statically-typed programming languages like Java or C. However, several challenges remain: Crucial security properties like non-interference are hyperproperties, which are not supported by most standard verification tools. Additionally, new verification techniques have to be developed to enable the verification of programs written in other mainstream programming languages that offer fewer or different static guarantees. In this thesis, we expand the state of the art in these two directions, by enabling the automated verification of hyperproperties in standard verification tools, and making existing verification techniques accessible for the dynamic Python language as well as Ethereum smart contracts.

First, we present a modular verification for Python programs, targeting complex safety and security properties for realistic code. We address the challenges stemming from dynamic typing and the resulting lack of static knowledge about Python programs using a mix of permissive static checks using an unsound type system and additional sound checks in the verifier. The resulting verification technique is able to support typical Python code patterns, and can be integrated with existing specification and verification techniques for safety and security properties. We implement our solution in Nagini, an automated verifier for a substantial subset of Python 3, and demonstrate its ability to verify real-world Python code and prove complex program properties.

Second, we present modular product programs, a program transformation that enables the verification of hyperproperties using standard verification tools. Unlike existing solutions, modular product programs can be constructed automatically, impose no restrictions on program control flow, and, crucially, enable modular proofs of hyperproperties using relational specifications. We apply modular product programs to proving non-interference using special information flow specifications, which can express complex properties like value-dependent sensitivity and termination-sensitive non-interference. We implement the product transformation for the Viper verification infrastructure and show that it can prove both information flow properties and other hyperproperties for challenging examples from the literature.

Third, we show how product constructions like the aforementioned modular product programs, which are usually defined for small, sequential languages, can be applied to

more complex languages and integrated with existing verification tools that are based on intermediate verification languages (IVLs). The key idea is to perform the product construction on the level of the (simple) IVL instead of the (complex) source language, thus enabling existing tools to verify hyperproperties with little engineering effort. We show that this approach is not always sound, but provide a simple criterion that guarantees soundness and can be easily checked in practice. Furthermore, we show how one can verify non-interference properties of concurrent source programs using sequential product constructions. We demonstrate the feasibility of this approach by implementing it for Nagini. Our evaluation shows that this extended Nagini implementation, developed with minimal effort, can compete with existing specialized verification tools for non-interference in terms of its expressiveness, while offering acceptable performance.

Fourth, we present a modular specification and verification technique for Ethereum smart contracts, i.e., small programs that execute in the Ethereum blockchain, which typically implement custom resources and resource transfers. Smart contract verification is challenging because contracts execute in an adversarial environment, interacting with malicious outside contracts that can perform re-entrant calls. We present a specification and verification technique that, unlike existing techniques, is sound and precise in this context and does not assume or impose limits on re-entrancy. Furthermore, we expand this technique to enable the contract-modular verification of sets of collaborating contracts for the first time. Finally, we propose a domain-specific specification language that lets users express properties directly in terms of resources and resource transfers, reducing the likelihood of erroneous specifications. We implement our technique in 2vyper, an automated verification tool for the Vyper language (a smart contract language for Ethereum which is distinct from the aforementioned Viper verification language), and demonstrate its ability to verify complex properties of real smart contracts.

Zusammenfassung

Da Softwaresysteme immer mehr unsere kritische Infrastruktur und unser tägliches Leben bestimmen, ist es wichtig, ihre Korrektheit und Sicherheit zu gewährleisten. Wenn übliche Sicherheitsvorkehrungen wie Testen nicht ausreichen, kann Verifikation benutzt werden, um Programme mathematisch ein für allemal korrekt zu beweisen. In den letzten Jahrzehnten gab es grosse Fortschritte im Design und der Automatisierung von Verifikationstechniken, so dass heutzutage mehrere Verifikationstools in der Lage sind, automatisch Eigenschaften für statisch typisierte Programmiersprachen wie C oder Java zu beweisen. Dennoch bleiben viele Probleme ungelöst: Wichtige Sicherheitseigenschaften wie Informationsflusseigenschaften sind sogenannte Hyperproperties, die von Standard-Verifikationstools nicht unterstützt werden. Ausserdem müssen neue Verifikationstechniken entwickelt werden, um andere weit verbreitete Programmiersprachen zu unterstützen, die weniger (oder andere) statische Garantien bieten. In dieser Dissertation erweitern wir den Stand der Technik in diese zwei Richtungen, indem wir die automatisierte Verifikation von Hyperproperties in Standard-Verifikationstools ermöglichen und existierende Verifikationstechniken für die dynamische Python Sprache sowie Sprachen für Ethereum Smart Contract nutzbar machen.

Zuerst präsentieren wir eine modulare Verifikationstechnik für Python-Programme, die darauf abziehlt, komplexe Korrektheits- und Sicherheitseigenschaften von realistischem Code zu beweisen. Den Mangel an statischen Garantien in Python adressieren wir mit einer Kombination aus einem optimistischen statischen Typsystem und zusätzlichen Checks als Teil der Verifikation. Die resultierende Verifikationstechnik ist in der Lage typischen Python Code zu verifizieren und kann mit existierenden Spezifikations- und Verifikationstechniken für Sicherheitseigenschaften integriert werden. Wir haben unsere Technik in Nagini, einem automatisierten Veritikationstool für Python 3, implementiert, und demonstrieren in unserer Auswertung, dass Nagini in der Lage ist, komplexe Eigenschaften von echtem Python Code zu beweisen.

Zweitens präsentieren wir Modular Product Programns, eine Programmtransformation, die es ermöglicht, Hyperproperties mit Standard-Verifikationstools zu beweisen. Anders als existierende Lösungen können Modular Product Programs automatisch konstruiert werden, schränken den Kontrollfluss des Programms nicht ein, und ermöglichen modulare Beweise mithilfe relationaler Spezifikationen. Wir wenden Modular Product Programs auf das Problem von Informationsfluss an und präsentieren Informationsflussspezifikationen, die komplexe Eigenschaften wie werteabhängige Sensitivität und terminierungssensitive Nichtinterferenz ausdrücken können. Wir implementieren die Produkttransformation für die Viper Verifikationsinfrastruktur und zeigen, dass sie sowohl Informationsflusseigenschaften als auch andere Hyperproperties für komplexe Beispiele aus der Literatur beweisen kann.

Drittens zeigen wir, wie Produktkonstruktionen wie Modular Product Programs, die üblicherweise für einfache, sequentielle Programmiersprachen definiert werden, auch auf komplexere Sprachen angewendet und in existierende Verifikationstools, die auf Zwischensprachen (IVLs) basieren, integriert werden können. Die Kernidee ist, die Produkttransformation auf der (einfachen) Zwischensprache statt der (komplexen) Quellsprache anzuwenden, und dadurch existierende Tools ohne viel Aufwand in die Lage zu versetzen, Hyperproperties zu verifizieren. Wir zeigen, dass dieser Ansatz nicht immer korrekt ist, aber identifizieren ein einfach überprüfbares Kriterium, das Korrektheit garantiert. Ausserdem zeigen wir, wie auf diesem Weg Informationsflusseigenschaften von nebenläufigen Programmen mit sequentiellen Produktkonstruktionen bewiesen werden können. Wir demonstrieren die Machbarkeit unseres Ansatzes, indem wir ihn in Nagini implementieren; unsere Auswertung zeigt, dass die erweiterte Version von Nagini, die mit minimalem Aufwand entwickelt wurde, in seiner Ausdrucksstärke mit spezialierten Verifikationstools mithalten kann und akzeptable Performance bietet.

Viertens präsentieren wir eine modular Spezifikations- und Verifikationstechnik für Ethereum Smart Contracts, also kleine Programme, die in der Ethereum Blockchain ausgeführt werden, und die oft benutzerdefinierte Resourcen und Resourcenaustausch implementieren. Die Verifikation von Smart Contracts ist schwierig, weil Smart Contracts mit anderen, möglicherweise böswilligen, Contracts interagieren müssen, die ihrerseits jederzeit den Contract zurück aufrufen können (solche Aufrufe werden als re-entrant bezeichnet). Wir präsentieren eine Spezifikations- und Verifikationstechnik, die in diesem Kontext, anders als existierende Lösungen, korrekt und präzise ist, ohne solche Aufzufe zu beschränken. Dann erweitern wir diese Technik, um auch die modulare Verifikation von Gruppen von Contracts, die zusammen eine Applikation bilden, zu ermöglichen. Schliesslich präsentieren wir eine domänenspezifische Spezifikationssprache, die es Benutzern ermöglicht, Eigenschaften direkt auf dem Level von Resourcen und Resourcenaustausch zu beschreiben und so die Wahrscheinlichkeit falscher Spezifikationen verringert. Wir implementieren unsere Lösung in 2vyper, einem automatischen Verifikationstool für die Vyper-Sprache (einer Programmiersprache für Ethereum Smart Contracts, nicht zu verwechseln mit der vorher erwähnten Viper-Sprache), und zeigen, dass 2vyper in der Lage ist, komplexe Eigenschaften von echten Smart Contracts zu beweisen.

Acknowledgements

The six years of working on my PhD had many highs and lows. Being a PhD student was in many ways a dream job for me, parts of which I've made really difficult for myself for some reason. I'd like to say a huge thank you to everyone who somehow dragged me through the difficult times, and I want to thank some people in particular.

First of all, thanks to Peter, who was probably the best possible supervisor I could have hoped for, gave me lots of freedom to explore things I found interesting, was infinitely patient when I wasn't making progress, and continues to be a great role model when it comes to putting out good work and writing about it. I'm happy I'll get to stay around for a bit longer and looking forward to the next year.

Thanks to David Basin, Marieke Huisman and Toby Murray for taking the time to review this thesis and for their feedback, which improved it quite a bit.

Thanks to Caterina, who taught me how academia in general and PL in particular work, and who was a wonderful friend for complaining about research and everything else. Thanks to Alex for playing a huge role in making our group a group, explaining verification to me, and being my badminton buddy. Look, we even managed to write a paper together in the end!

Thanks to Jérôme (and Gabriela) for integrating me into Swiss society, many *many* hours of burgers, sugar-related supermarket excursions, and the occasional beer. We actually both made it somehow, I still can't believe it. Thanks Arshavir for basically being my social life during the first months in Zurich. As you wrote, we (used to? I think that's changed a bit) never agree on anything, but I'm glad you were there both at work and afterwards. Thanks to Malte and Birte for being rays of sunshine, life goals in so many ways, and just generally really cool.

Thanks to Felix for being hilarious, often even intentionally, and for explaining lots of conundra to me. Thanks Thibault for doing *that* proof that would have haunted me for a decade otherwise, for being a lot of fun to work with, and for his excellent pronunciation of the letter H. Thanks Vytautas for being the best possible first Masters's student anyone could dream to supervise, and for always sharing his wonderfully positive outlook on life. Also thanks to you and Sam for teaching me Python while I was making a verifier for it.

Thanks to Alexandra for being my gossip-mate, Gaurav and João for many fun discussions about academia, research, and PhDs, to Lucas for hiring me, to Jonás for the pizza thingy, and to Maria for reminding me that academia can be fun. Thanks to Milos, Uri, Mitko, Valentin, Fábio, Martin, Federico, Linard, Christoph, Aurel, Dionisios, Marlies, Sandra and everyone else for making our group a fun place to be and do research. Special thanks to all group members for tolerating the state of my desk, and to my flatmates for tolerating my weird day/night cycle and, generally, the state of my life.

Thanks to my coauthors Sam Hitz, Mostafa Hassan, Caterina Urban, Christoph Sprenger, Tobias Klenze, Felix A. Wolf, Martin Clochard, David Basin, Severin Meier, Christian Bräm, Robin Sierra, Alex J. Summers, and Peter of course.

Thanks to the students and interns I've supervised and whose work contributed to this thesis in different ways: Sahil Rishi, Vytautas Astrauskas, Anatole Dahan, Mostafa Hassan, Benjamin Weber, Benjamin Schmid, Christian Knabenhans, Sascha Forster, Severin Meier, Mathias Blarer, Robin Sierra, Pavel Pozdnyakov, Constantin Müller, and Christian Bräm.

Thanks to Niko Stotz, Tamás Szabó, Giuseppe Platania, Andrzej Filinski, Andreas Graf and Klaus Birken for awakening my interest in PL stuff in the first place.

Huge thanks to my Vollidioten, especially Mary, and my Herzallerliebste, for still inviting me to stuff even though I live far away and have started to talk funny.

Thanks to Miri, Hauke, and their fluffy little zoo that massively improves basically every day of my life.

And finally, thanks a lot to my parents who have literally always supported me in everything I wanted to do. Throughout this whole thing you always told me to take care of my well-being first (no matter what that might mean for academic titles or my CV) and I really appreciate that.

Contents

Introduction 1.

Our world is ruled by computer systems: They run our infrastructure, handle our money and our most sensitive data, facilitate many of our social interactions, and sometimes even control the light bulbs in our living rooms. As a result, there are considerable incentives for malicious actors to manipulate or attack these systems, and it is important to design them as securely as possible.

The usual approaches for making systems secure in practice, such as using established design principles, code reviews, and testing, often provide a level of security which is "good enough" for their intended purpose, but does not provide any absolute guarantees. As a result, these practices are insufficient for systems with the highest security requirements, which are likely to be attacked and which will lead to substantial problems when exploited successfully. For such systems, *formal methods* can provide rigorous mathematical correctness guarantees. For software in particular, the gold standard is *deductive program verification*, which can prove programs correct with respect to complex specifications once and for all.

1.1. Security Property Verification

The central topic of this thesis is the verification of *program properties* that encompass software *safety* and *security*. In particular, we focus our attention on the *implementations* (as opposed to higher level designs or models) of *individual programs* (as opposed to larger systems consisting of different components), written in mainstream programming languages. Note that this *can* encompass reasoning about the way an individual component interacts with other components that are part of a system; here, though, we focus on proving that the component's interactions conform to some protocol, rather than proving that the protocol itself ensures some security property for the whole system (which can be proved separately).

Security has traditionally been defined in terms of the so-called CIA triad of *confidentiality*, *integrity*, and *availability* [162]. While these are somewhat broad terms, proving software security according to these criteria can require proving different concrete program properties. We focus on the following:

[162]: Myler et al. (2006), 'ISO 17799: Standard for security'

1. System availability usually requires *crash freedom*, i.e., that the program does not abort because of unexpected and unintended errors. This property can encompass memory safety in programming languages that do not guarantee this property by construction, and in general requires showing the absence of out-of-bounds accesses, divisions by zero and similar properties. Note that these properties are usually called safety properties, but they are obviously relevant for system security as well.

2. Similarly, availability may require proving *progress* properties, i.e., to stay available, a program must keep interacting with its environment, and must not get stuck, e.g., in an infinite loop or because of a deadlock.
3. Proving integrity may involve proving that, regardless of its (potentially attacker-controlled) inputs, some set of invariants that a program relies on can never be broken; this amounts to proving *functional properties* of (parts of) the code.
4. Alternatively, proving integrity may require showing that some data cannot be controlled or affected by the attacker. This property can be formalized as *non-interference*, a property that can express that inputs controlled by an attacker cannot taint certain critical variables.
5. Confidentiality requires proving *information flow security* of the involved program(s), which can again be formalized as (different kinds of) *non-interference*.
6. And finally, for systems consisting of more than one component, proving *any* security property of the system as a whole may require proving properties about individual components' interactions with their environment, i.e., their *input-output-behavior*: For example, as mentioned above, the security of a system may depend on the fact that its components communicate according to some pre-defined protocol that is known to be secure.

Note that this list is by no means exhaustive, and there are many other (even code-level) security properties that we do not consider in this thesis; for example, we will not attempt to prove the correct implementation or usage of cryptographic primitives, nor will we reason about low-level interactions between software and hardware security features.

1.2. State of the Art

Deductive program verification was first introduced by Hoare [105]; program logics known as *Hoare logics* allow proving *Hoare triples* of the form $\models \{P\}c\{Q\}$. Such a triple states that the command c, when executed in a state satisfying the *assertion* P, will not abort with an error, and will result in a state satisfying assertion Q if it terminates. Such Hoare triples express *partial correctness* properties (since they do not guarantee termination) [65]; these are sufficient to formally express crash freedom and functional correctness. Hoare logics can also express *total correctness*, whose corresponding Hoare triples also imply that c will always terminate when executed from a state satisfying P, making it possible to express progress properties as well.

[105]: Hoare (1969), 'An Axiomatic Basis for Computer Programming'

[65]: Dunlop et al. (1982), 'A Comparative Analysis of Functional Correctness'

To be able to prove correctness of substantial code bases, it must be possible to reason about different parts of the program *modularly*, i.e., in isolation, taking only the *specifications* of other parts of the program into account. However, traditional Hoare logics are not well-suited for modular reasoning about programs that work with global state, in particular program heaps: If a caller and a callee function are to be verified separately, the caller function by default does not know which part of the global state may have been modified by the callee, and explicitly specifying this information in postconditions for heaps of unknown and arbitrary size is infeasible. *Separation logics* [180] as well as *ownership models* [157] solve this problem by requiring each function or method to implicitly or explicitly state the heap footprint they may modify; caller methods can then assume that any parts of their own footprint that are not in the callee's footprint will not be touched by a call. The same concepts easily extend to reasoning about *concurrent* programs; *concurrent separation logics* [169] ensure that different threads operate on separate footprints (i.e., that programs are data race free), and can therefore be reasoned about as if each thread executes without interference. Some of the fundamental verification techniques mentioned so far have also been integrated with specification techniques for object-oriented programming languages, allowing for example to prove *class invariants* [177] that define consistency criteria which must hold for every instance of a class, and extending this concept to structures consisting of multiple objects [128, 157].

[180]: Reynolds (2002), 'Separation Logic: A Logic for Shared Mutable Data Structures'

[157]: Müller (2002), *Modular Specification and Verification of Object-Oriented Programs*

[169]: O'Hearn (2007), 'Resources, concurrency, and local reasoning'

[177]: Poetzsch-Heffter (1997), 'Specification and verification of object-oriented programs'

[128]: Leino et al. (2004), 'Object Invariants in Dynamic Contexts'

[157]: Müller (2002), *Modular Specification and Verification of Object-Oriented Programs*

Various specialized Hoare or separation logics have been proposed that allow reasoning about other program properties, for example input/output-behavior [172]. *Non-interference*, however, is a *hyperproperty* [46], that is, a property of sets of executions of a program; as we hinted at above, it is useful for expressing both integrity and information flow security properties. While many type systems and static analyses exist to check non-interference automatically, these are limited in their expressiveness and precision, and therefore usually insufficient when trying to show the security of complex, real-world systems. Therefore, it is necessary to use deductive verification techniques to verify at least the most complex parts of a code base. However, ordinary Hoare and separation logics are not able to prove non-interference, since they reason about single executions only; that is, they are limited to proving *trace properties*. As a result, some separation logics specifically target information flow security [179]. Alternatively, like other hyperproperties, non-interference can also be verified using *relational* Hoare logics [30], which explicitly reason about multiple executions at once. Another alternative is to use *self-composition* [25] or *product programs* [23], which *reduce* hyperproperties of one program to ordinary trace properties of a newly-constructed (product) program. The latter in particular allows proving hyperproperties using standard program logics. One important thing to note is that relational Hoare logics and most product program constructions only target sequential programs, whereas there are a number of specialized information flow logics, type systems, and static analyses that also target concurrent programs [72, 89, 203].

[172]: Penninckx et al. (2015), 'Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs'

[46]: Clarkson et al. (2010), 'Hyperproperties'

[179]: Prabawa et al. (2018), 'A Logical System for Modular Information Flow Verification'

[30]: Benton (2004), 'Simple relational correctness proofs for static analyses and program transformations'

[25]: Barthe et al. (2011), 'Secure information flow by self-composition'

[23]: Barthe et al. (2011), 'Relational Verification Using Product Programs'

[72]: Ernst et al. (2019), 'SecCSL: Security Concurrent Separation Logic'

[89]: Giffhorn et al. (2015), 'A new algorithm for low-deterministic security'

[203]: Smith (2007), 'Principles of Secure Information Flow Analysis'

Finally, the past two decades have seen enormous progress in mechanizing and automating proofs, including program correctness proofs. *Interactive theorem provers* like Coq [20] or Isabelle/HOL [168] allow users to define theorems and proofs, and mechanically check the proofs for correctness, yielding strong guarantees. Typically, there is also limited support for proof automation, for example using so-called tactics. On the other hand, SMT-solvers like Z3 [156] and CVC4 [22] are able to check proof obligations in first order logic that include decidable theories like linear integer arithmetic in a fully automated way. Combined with techniques for extracting proof obligations, SMT-solvers enable the creation of verification tools which automatically check programs annotated with specifications for correctness.

[20]: Barras et al. (1997), 'The Coq proof assistant reference manual: Version 6.1'

[168]: Nipkow et al. (2002), *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*

[156]: Moura et al. (2008), 'Z3: An Efficient SMT Solver'

[22]: Barrett et al. (2011), 'CVC4'

While some automated verification tools have been built directly on top of SMT solvers, the creation of *intermediate verification languages* (IVLs, examples of which include Boogie [18], Why3 [82] and Viper [159]), simple languages with associated automated verification tools, has massively simplified the creation of new verification tools: Creating a verifier for a new programming language now *only* requires encoding the new language and its specifications into the IVL in a sound way, leaving the generation of verification conditions and the interaction with provers to the IVL's infrastructure. As a result of the rise of IVLs, a number of verification tools for real languages now exist, and some verification techniques are even presented primarily as encodings into IVLs and no longer as Hoare logics or as weakest precondition calculi.

[18]: Barnett et al. (2005), 'Boogie: A Modular Reusable Verifier for Object-Oriented Programs'

[82]: Filliâtre et al. (2013), 'Why3 - Where Programs Meet Provers'

[159]: Müller et al. (2016), 'Viper: A Verification Infrastructure for Permission-Based Reasoning'

1.3. Challenges

As a result of the last decades of work, there is now a significant number of verification tools able to prove trace properties of programs written in standard statically-typed languages like Java or C.

In this thesis, our central goal is to extend the state of the art to enable

- modular verification of more advanced properties, particularly properties like non-interference which are highly relevant for proving systems secure,
- for languages that deviate from the standard statically-typed object-oriented languages, for example by being dynamically-typed (like Python) or by having an entirely different execution model (like smart contracts)
- in a way that can be automated and integrated with existing verification tools.

In particular, we address the following challenges:

1.3.1. Challenge 1: Different Programming Paradigms

Most current verification techniques (for imperative programs) target statically-typed languages like Java or C, and are sound only if the *entire* running application is verified. Verification of programs in different programming environments is far less explored: *Dynamic* languages like Python, for example, perform no static checks, therefore give almost no static guarantees, and offer some dynamic language constructs not seen in statically-typed languages. As a result, static reasoning about them is substantially more challenging, requiring more information from specifications, and having to reason about more possible behaviors. Many established ways of reasoning about programs do not directly apply in such a context. On the other hand, *smart contracts* are programs which are generally statically-typed, but which are executed in the presence of other code, which is generally unverified and potentially adversarial. As a result, verification techniques like separation logic that require that *all* code be verified again do not apply.

1.3.2. Challenge 2: Automated and Modular Hyperproperty Verification

The verification of hyperproperties like non-interference is still challenging in practice. While relational logics allow (modularly) reasoning about them, they are either not automated at all, or use specialized tools that are not found in standard verification tool chains [206]. On the other hand, self-composition and product programs allow reducing hyperproperties to trace properties of newly-constructed programs, but have other drawbacks: Self-composition does not allow for modular reasoning about procedure calls, whereas different existing product program constructions are either incomplete when dealing with different control flow in different executions [23], or must be constructed manually [24].

[206]: Sousa et al. (2016), 'Cartesian Hoare logic for verifying k-safety properties'

[23]: Barthe et al. (2011), 'Relational Verification Using Product Programs'

[24]: Barthe et al. (2013), 'Beyond 2-Safety: Asymmetric Product Programs for Relational Program Verification'

1.3.3. Challenge 3: Non-interference Verification for Concurrent Programs

Verification of hyperproperties such as non-interference is inherently more challenging for concurrent programs, since different executions can potentially have different thread schedules, which can in turn impact the results of a program. As a result, self-composition and product programs are typically defined only for sequential programs and do not easily extend to a concurrent setting. While some verification techniques for (variations of) non-interference for concurrent programs have been proposed [86, 160, 194], almost no verification techniques in this setting have been automated, and the few that have again require specialized tools [72].

[86]: Frumin et al. (2021), 'Compositional Non-Interference for Fine-Grained Concurrent Programs'

[160]: Murray et al. (2018), 'COVERN: A Logic for Compositional Verification of Information Flow Control'

[194]: Schoepe et al. (2020), 'VERONICA: Expressive and Precise Concurrent Information Flow Security'

[72]: Ernst et al. (2019), 'SecCSL: Security Concurrent Separation Logic'

1.3.4. Challenge 4: Integration of Advanced Property Verification Into Existing Tools

A significant number of verification tools based on IVLs and SMT-solvers has been developed over the past decade, targeting a number of real programming languages. These generally target standard trace properties. While new techniques are constantly being developed to prove more advanced properties, those are usually either not automated or implemented in custom tools, using custom specification languages and generally preventing easy combination with existing verification infrastructures. Even when new verification techniques are defined in terms of program transformation or other encodings that can be integrated into standard verification tools, those transformations and encodings are typically defined only for smaller languages lacking many features used in practice. Since proving the security of real software systems generally requires proving a combination of different properties, it is vital that new verification techniques are designed in such a way that existing tool chains can be extended to prove new, more advanced properties.

1.4. Contributions and Outline

We address each of these challenges by making the following contributions, each of which corresponds to a chapter of this thesis:

1.4.1. Contribution 1: Modular Verification of Python Code

In Chapter 2, we present a technique for modular verification of Python code, addressing Challenge 1. Aiming to support the verification of *typical* (but not necessarily *all*) real-world Python code with manageable specification overhead, we carefully design specifications and, where necessary, impose restrictions that enable us to reason about code statically and modularly. The result is a specification and verification methodology that builds on standard verification techniques but adapts them to the specific Python setting, for example by using the verifier for precise type checking, and supporting dynamic addition and removal of fields in a separation logic-like setting. We implement our approach in Nagini, a verifier that encodes a substantial subset of (concurrent) Python into the Viper IVL. Nagini proves the absence of runtime errors by default and allows users to specify functional correctness and progress properties, as well as input-output-behavior, i.e., several kinds of program properties that are crucial for proving security properties of the systems they belong to.

1.4.2. Contribution 2: Modular Product Programs

In Chapter 3, we introduce modular product programs, a kind of product program that allows modular verification of k-safety hyperproperties using standard verifiers, and can be constructed automatically, addressing Challenge 2. We introduce *relational specifications* that let users specify hyperproperties and reason about procedure calls modularly. Furthermore, we define *information flow specifications*, which are specification constructs specifically tailored towards modularly proving non-interference, and show how they can be encoded into modular product programs. We show that these specifications are sufficiently flexible to encode advanced concepts like value-dependent sensitivity and termination-sensitive non-interference. We implement the product transformation for Viper and show that they can be used to prove non-interference for challenging programs from the literature, as well as other hyperproperties.

1.4.3. Contribution 3: Using Modular Product Programs in IVL-Based Tools

In Chapter 4, we explore how modular product programs, which are defined for a simple sequential language, can be integrated into an existing IVL-based verification tool for more complex and concurrent languages, addressing Challenges 3 and 4. In particular, we explore under which conditions it is sound to perform the product construction *on the level of the IVL* (typically also a simple sequential language), and exploit the existing encoding from the more complex source language to the IVL to verify hyperproperties in the source language with little effort. We show that this approach is not always sound, but that there are simple conditions on the existing IVL-encoding one can check to ensure soundness, which are usually fulfilled by existing encodings, requiring only few specific adaptations. In particular, we also show how this approach can be used to prove non-interference of *concurrent* source programs using sequential product construction. The resulting technique allows us to retrofit existing IVL-based verifiers for trace properties to also verify non-interference, even for concurrent programs, with little effort. We implement the approach for Nagini by exploiting a product construction on the level of the Viper IVL, and show that the resulting tool, while slower than special-purpose information flow verifiers, can prove complex non-interference properties for challenging examples from the literature with acceptable performance.

1.4.4. Contribution 4: Modular Smart Contract Verification

Finally, in Chapter 5, we move to a different execution model and present a verification approach for Ethereum smart contracts, again addressing Challenge 1. Exploiting the special encapsulation guarantees provided by smart contract languages, we address the problem that smart contracts have to interact with an unverified and potentially malicious environment, which can in particular make *re-entrant calls* in order to corrupt contract state in a way that benefits them. Our technique is the first that can soundly prove strong correctness properties of smart contracts without having to restrict re-entrancy (which would rule out contracts that use re-entrancy on purpose). It is also the first verification technique that allows modular verification of sets of *collaborating* smart contracts, i.e., separate verification of each contract that only uses an interface description of the contracts it interacts with, abstracting over concrete implementations. Finally, we provide special domain-specific specifications for *resources* and *resource transactions*, which are the purpose of most existing smart contracts. These specifications allow users to specify desired contract behavior on a higher level of abstraction, and the associated verification technique finds common problems (like contracts that accidentally duplicate resources, or the bug in the infamous TheDAO contract) by default. We have implemented our technique in 2vyper, a verifier for the Python-like Vyper language for Ethereum smart contracts, which is again based on the Viper IVL. Our evaluation shows that our technique can verify strong properties of real, potentially collaborating smart contracts, without having to restrict re-entrancy.

1.5. Publications

The work in this thesis has been partially published in the following conference and journal publications:

Chapter 2 is based on

> *Marco Eilers and Peter Müller.*
> *Nagini: A Static Verifier for Python*

In CAV 2018, Springer [97]

Chapter 3 is based on

Marco Eilers, Peter Müller, and Samuel Hitz.
Modular Product Programs
In ESOP 2018, Springer [68]

as well as the extended version of this paper

Marco Eilers, Peter Müller, and Samuel Hitz.
Modular Product Programs
In TOPLAS 2020, ACM [69]

Chapter 4 is based on

Marco Eilers, Severin Meier, and Peter Müller.
Product Programs in the Wild: Retrofitting Program Verifiers to Check Information Flow Security
In CAV 2021, Springer [66]

Chapter 5 is based on

Christian Bräm, Marco Eilers, Peter Müller, Robin Sierra, and Alexander J. Summers.
Rich Specifications for Ethereum Smart Contract Verification
In OOPSLA 2021, ACM [43]

1.6. Further Contributions During the Thesis Work

In the time working on this thesis, the author has made additional contributions to the scientific community which are not included in this thesis:

Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller.
MaxSMT-Based Type Inference for Python 3
In CAV 2018, Springer [97]

Christoph Sprenger, Tobias Klenze, Marco Eilers, Felix A. Wolf, Peter Müller, Martin Clochard, and David Basin.
Igloo: Soundly Linking Compositional Refinement and Separation Logic for Distributed System Verification
In OOPSLA 2020, ACM [207]

2. Modular Verification of Python Code

Dynamic languages have become widely used because of their expressiveness and ease of use. The Python language in particular is popular in domains like teaching, prototyping, and more recently data science. Python's lack of safety guarantees can be problematic when, as is increasingly the case, it is used for critical applications with high correctness demands. The Python community has reacted to this trend by integrating type annotations and optional static type checking into the language [181]. However, there is currently little tool support for reasoning about Python programs beyond type safety.

[181]: van Rossum et al. (2014), *PEP 484: Type Hints*

A key reason for this lack of tool support is that the lack of static information and the general design of the language, which focuses on expressiveness, make reasoning about Python programs in many ways inherently more difficult compared to languages like Java or C. This is true even when disregarding language features like runtime code generation, which is generally regarded as making verification infeasible in practice.

The biggest difference is the lack of static type checking: Since Python performs dynamic and structural type checking, it is difficult to determine vital information about a program's behavior from its AST. As an example, reasoning about the behavior of a piece of code that performs a method call requires having some information as to the target of said call; while, in statically and nominally-typed languages, it is generally easy to determine call targets (or, at least, sets of possible call targets) from the type information available in the AST, this is much more difficult in a language like Python. This problem cannot be mitigated by simply imposing a standard, conservative type system after the fact, since typical Python programs are written in such a way that most traditional type systems would reject them outright.

Furthermore, in Python, even the high-level structure of a program (i.e., the set of modules, classes, and methods) is not static, but evolves as the program executes; declarations of classes, methods, etc. are executed inbetween other pieces of code. The result is a class of potential bugs that does not even exist in other languages, namely, runtime errors resulting from the usage of a class or method when it does not yet exist.

Finally, Python has an object model and a way of performing attribute lookups that is far more complex than the simple field reads or pointer dereferences used in other languages, and a plethora of smaller language features in Python are either more complex than their equivalents in other languages (e.g., assignments), or mirror those in other languages, but are currently poorly supported by verification tools for those languages as well (e.g., exception handling with finally-blocks).

In this chapter, which is partly based on the CAV 2018 paper "Nagini: A Static Verifier for Python" [67], we present our verification approach for Python 3. Our goal is to be able to modularly prove complex, security-related properties of typical user-level Python code. That is, we do not necessarily aim to allow verifying *any* Python code if doing so would

[67]: Eilers et al. (2018), 'Nagini: A Static Verifier for Python'

come at too big a cost, e.g., if it would make verification inherently non-modular, or result in an impractically high specification overhead even for simple programs. However, we do aim to support typical Python code patterns, while keeping the required specification overhead similar to or lower than that of existing tools for other languages. We then use our verification approach as a basis for integrating existing verification techniques for proving complex, security-related program properties.

To summarize, we make the following contributions:

- We present a novel verification technique for Python 3 code that allows the modular verification of complex properties of typical real-world code for the first time. Our technique addresses the challenge of dynamic typing by using a hybrid approach to type checking: we build on an existing static type system checking some properties optimistically (i.e., in an unsound way), and supplement it with sound checks performed by the verifier. The resulting technique can precisely reason about type information, variable- and field definedness, and thus accommodates typical Python code patterns. We allow sound reasoning about dynamically-bound calls by imposing both nominal and behavioral subtyping, but support the use of union types to be able to verify (most) code that relies on the flexibility provided by Python's standard structural subtyping. Furthermore, we verify the correctness of all top-level statements, in particular, by checking that all declarations are used only after they are defined.
- We integrate our basic verification approach with existing specification and verification techniques for advanced program properties that are vital for proving system security, like input/output-behavior, progress properties, and general safety. [1]
- We implement our approach in Nagini, an automated, modular verification tool for a substantial subset of Python 3. Nagini automates the verification of Python programs by encoding them into the Viper IVL [159], ultimately using the SMT solver Z3 [156], and allows users to specify complex correctness properties directly on the level of the Python code.
- We demonstrate Nagini's ability to verify typical Python code by applying it to a set of small individual Python programs as well as a substantial part of a Python code base not written with verification in mind. Subsequently, we show its ability to verify security-related properties by verifying the I/O behavior of two programs implementing different roles in a security protocol; this proof can then be combined with a proof of protocol correctness, resulting in a security proof of the entire system.

The remainder of this chapter is structured as follows: In Sec. 2.1, we give an overview of the state of the art of automated program verification for statically-typed programming languages. In Sec. 2.2, we present our verification technique for Python, by first outlining the challenges Python poses for static verification, and subsequently explaining our approach to solving those challenges. After showing how reasoning about Python code is possible in principle, we then show how we can integrate our basic verification technique with techniques for specifying and verifying the advanced program properties required to prove system security in Sec. 2.3. In Sec. 2.4, we briefly describe a possible approach to proving

[1] In Chapter 4, we will additionally enable the verification of confidentiality and integrity properties for Python programs.

[159]: Müller et al. (2016), 'Viper: A Verification Infrastructure for Permission-Based Reasoning'

[156]: Moura et al. (2008), 'Z3: An Efficient SMT Solver'

the soundness of our verification technique. Subsequently, we describe our implementation of the presented approach in Nagini in Sec. 2.5, we evaluate it by applying it to different verification tasks that demonstrate both its ability to verify realistic code and to express and prove complex program properties in Sec. 2.6. Finally, we describe related work in Sec. 2.7 and conclude in Sec. 2.8.

2.1. Modular Verification of Statically-Typed Object-Oriented Programs

2.1.1. Procedure and Method Specifications

In object-oriented programming languages, code is generally contained inside *methods*; in imperative programs, in *procedures*. Since the differences between the two do not matter for verification, we will generally refer to named pieces of code that take arguments and can have side effects as methods throughout this thesis.

In Chapter 1, we have already mentioned the *Hoare triple* $\vDash \{P\}c\{Q\}$, which forms the basis of deductive verification [105], and which states that when c is executed in any state satisfying P, then it will not abort with an error, and if its execution terminates, the resulting state will fulfill Q. Following the same basic idea, the basic specifications for methods are *preconditions* and *postconditions*; if a method has an assertion P as its precondition and an assertion Q as its postcondition and its body is c, then the aforementioned Hoare triple must hold [9]. This understanding of specifications is called *partial correctness*, since it does not require the method to terminate; the alternative, *total correctness*, requires that the body must terminate from every state fulfilling P [65]. Unless stated otherwise, our understanding will be that of partial correctness throughout this chapter.

Note that the exact definition of what it means to “abort with an error” may vary between different languages and verification systems, and that these basic specification constructs are often supplemented with others:

- *Loop invariants* are assertions used to verify loops: A loop’s invariant must hold when the loop is first executed, and must then inductively hold after each subsequent iteration of the loop. They are generally thought of as *auxiliary* specifications, that is, one proves a loop invariant in order to be able to prove that a method specification holds for the method containing the loop, not primarily because one is interested in the loop invariant itself.
- In languages with exception handling, *exceptional postconditions* can describe which properties the program state must have when a method does not terminate normally by returning, but instead terminates by raising an exception [122].
- In object-oriented languages, *class invariants* can describe properties that all instances of a class must have at any point while no method of the object is executing, i.e., the properties that make up a consistent state of a class instance [177]. In simple cases, class invariants have to be established by the constructor, and must

[105]: Hoare (1969), ‘An Axiomatic Basis for Computer Programming’

[9]: Apt (1981), ‘Ten Years of Hoare’s Logic: A Survey - Part 1’

[65]: Dunlop et al. (1982), ‘A Comparative Analysis of Functional Correctness’

[122]: Leavens et al. (2008), *JML reference manual*

[177]: Poetzsch-Heffter (1997), ‘Specification and verification of object-oriented programs’

subsequently be shown to be preserved by every (public) method of a class (though there are edge cases where the necessary proof obligations become more complex).

- When dealing with concurrent programs, there are a number of additional specification constructs (for example *lock invariants* [132, 169]), which we will discuss later.

[132]: Leino et al. (2009), 'A Basis for Verifying Multi-threaded Programs'
[169]: O'Hearn (2007), 'Resources, concurrency, and local reasoning'

For verification to scale to realistic programs, it is important to be able to reason *modularly*, that is, to verify a program that may consist of many different methods by verifying each method independently of the others. In particular, when verifying the body of a method modularly, one must not assume any information about the initial state except the method's precondition; in particular, no additional properties that are known to hold at the call sites of the method in different parts of the code may be assumed. Additionally, when reasoning about a call to a different method, one only checks that the precondition of the called method is established at the call site, and subsequently assumes that the method's postcondition holds after the call has returned; one does not assume any information about the resulting state that one may learn from inspecting the method's code.

This principle sometimes requires verbose specifications in practice, but it ensures that verification can scale, since neither call sites nor called code needs to be inspected. As a result, the most complex verification task will be the verification of the most complex individual method, as opposed the the entirety of the program (although some existing work suggests that the overall verification *effort*, at least for some types of applications, may still scale quadratically with the size of the entire program, since larger programs require proving more complex specifications for individual methods [144]). Additionally, it ensures that it is possible to verify libraries (for which one does not know all possible call sites) or code that uses a library for which only a specification is available, but not the source code. It also enforces principles of abstraction and information hiding, since it ensures that correct programs do not break if the code in a callee method is exchanged with different code that also fulfills the method's specification.

[144]: Matichuk et al. (2015), 'Empirical Study Towards a Leading Indicator for Cost of Formal Software Verification'

2.1.2. Reasoning about Heap-Manipulating Programs

Modular reasoning is especially complex when dealing with programs that have modifiable global state, i.e., heap-manipulating programs. Consider the Java code example in Figure 2.1.

The postcondition of method `foo` promises that the returned integer is greater than one. Looking at the code, this is initially the case, since `x.f` is initialized to be two at the beginning of the method. However, inbetween those points, method `bar` is called, which could potentially modify the value of `x.f` to be a lower value, so that the postcondition of `foo` would not hold.

In modular verification, we can consider only the specification of `bar` to ensure that this is not the case. For example, if `bar` does not modify `x.f` at all, its postcondition could state that `x.f` is unchanged. However, this approach clearly does not scale: No method can be expected to explicitly

```
class X {
  int f;

  void bar(X x)
    // requires ...
    // ensures ...
  { ... }
}

class Z {
  float floatval;

  int foo(X x, X y, int i)
    // requires true
    // ensures result > 1
  {
    x.f = 2;

    Z z = new Z();
    z.floatval = Math.pi * i;
    y.bar(x);
    return x.f;
  }
}
```

Figure 2.1.: Example of a heap-manipulating program; foo may not be correct if bar modifies the f-field. Here and throughout, we use comments to denote method specifications: the requires keyword marks a precondition and the ensures keyword denotes a postcondition.

mention all heap location it *did not* modify in its postcondition, since there is a potentially infinite (and generally unknown) amount of such heap locations.

There are different approaches to solve this problem, which is often called the *frame problem*: Dynamic frames [115] requires methods to explicitly state the set of all heap locations they may potentially modify (using so-called modifies-clauses): Callers then know that any heap location not mentioned will not be changed, and can get information about the new values of the (now known) set of potentially-modified heap locations via the postcondition. Note that the latter may still be infinite. Ownership-based approaches [157] use a similar basic idea, but additionally define an ownership hierarchy between objects, which allows the use of modifies-clauses that express framing information on a higher abstraction level well-suited to object-oriented programs.

[115]: Kassios (2006), 'Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions'

[157]: Müller (2002), *Modular Specification and Verification of Object-Oriented Programs*

Finally, *permission logics* like *separation logic* [180] and *implicit dynamic frames* [199] allow accesses (i.e., reads or modifications) of heap locations only if the accessing method currently owns a *permission* to that location. Permissions come into existence when memory is allocated and belong to the method performing the allocation by default. Subsequently, these permissions can be passed through method calls, i.e., from the caller to the callee, if they are part of the precondition of the called method, and back from the callee to the caller on return if they are part of the method postcondition. Crucially, permissions can never be duplicated, so that at any given time, there is only ever a single method that currently owns the permission to a heap location and is therefore allowed to access it.

[180]: Reynolds (2002), 'Separation Logic: A Logic for Shared Mutable Data Structures'

[199]: Smans et al. (2012), 'Implicit dynamic frames'

In separation logic, the so-called *points-to assertion* $x.f \mapsto v$ denotes both a permission to location $x.f$, and expresses that the current value of said location is v. In implicit dynamic frames, which we will use throughout this chapter, these two aspects are separated: The assertion **acc**$(x.f)$ denotes a permission to location $x.f$ but makes no statement about its value; however, unlike in separation logic, the expression

$x.f$ can then be used directly to make statements about its value. For example, the assertion $x.f \mapsto 2$ in separation is analogous to the assertion $\texttt{acc}(x.f) * x.f = 2$ in implicit dynamic frames. Here, the operator $P * Q$ denotes a *separating conjunction*, whose meaning is analogous to an ordinary conjunction, but which (in implicit dynamic frames) represents the *sum* of the permission amounts in P and Q. Thus, the assertion $\texttt{acc}(x.f) * \texttt{acc}(y.f)$ represents *two* permissions, which implies non-aliasing: If x and y were aliases, the assertion would denote two permissions to the same location, but since there is only ever one permission for a single heap location, no method can ever own two permissions to the same location. Thus, the assertion implies that x and y must point to two different heap locations. *Inhaling* an assertion adds the permissions it represents to the current state and assumes the logical constraints the assertion contains. Conversely, *exhaling* an assertion removes permissions from the state and asserts all logical constraints.

The fact that there is only a single permission per heap location is also used to tackle the frame problem: If a method owns the permission to a heap location, it is guaranteed that no other method is allowed to modify that location until it gives the permission away. In the example in Figure 2.1, method foo would have to require the permission $\texttt{acc}(x.f)$ in its precondition (otherwise it would not be allowed to modify the field in the conditional), as shown in Figure 2.2 (left). Then, there are two possibilities for the behavior of bar:

1. bar may not require permission to $x.f$ in its precondition, as shown in Figure 2.2 (middle). As a result, foo keeps the permission to $x.f$ throughout the call to bar, and therefore can assume that the call cannot possibly modify $x.f$. As a result, in this case, no information from the postcondition of bar about $x.f$ is required to prove foo correct.
2. bar may require permission to $x.f$ in its precondition, as shown in Figure 2.2 (right), meaning that foo temporarily gives up said permission, and therefore gives others the ability to modify the field. When reasoning conservatively about foo, one therefore has to assume that $x.f$ may have been changed in arbitrary ways (e.g., set to zero, as in the example); now, one has to rely on information one may get from the postcondition of bar about the final value of $x.f$ to establish foo's correctness. Additionally, for foo to be correct, bar now *has to* give back permission to $x.f$ in its postcondition, since otherwise the read of $x.f$ in foo after the call would not be allowed.

In the first case, just from the fact that bar does not take the permission to $x.f$ from its caller, we can conclude that $x.f$ (like any other locations for which the caller may have permissions that are not given to the callee) will not be changed by the call, i.e., we can *frame* any knowledge we have about the values of those locations around the call.

More sophisticated permission systems exist; one important example are *fractional permissions* [41], i.e., the ability to split up whole permissions into fractions (where, for example, $\texttt{acc}(x.f, \frac{1}{3})$ denotes a third of a permission to location $x.f$): Holding a positive permission amount for a location that is less than one then gives the ability to read, but *not* modify, said location. This is again useful for framing: If a caller has a full permission to a heap location, but the callee only requires a fraction of a permission

[41]: Boyland (2003), 'Checking Interference with Fractional Permissions'

```
class Z {
  float floatval;

  int foo(X x, X y, int i)
    // requires acc(x.f)
    // ensures result > 1
    // ensures acc(x.f)
  {
    x.f = 2;

    Z z = new Z();
    z.floatval = Math.pi ∗ i;
    y.bar(x);
    return x.f;
  }
}
```

```
class X {
  int f;

  void bar(X x)
    // requires true
    // ensures true
  {
    ... // cannot modify x.f
  }

}
```

```
class X {
  int f;

  void bar(X x)
    // requires acc(x.f)
    // ensures acc(x.f)
  {
    x.f = 0;
  }
}
```

Figure 2.2.: Left: Method foo with permission annotations; the permission to $x.f$ is required from the caller and subsequently given back. Since z is a newly-created object, the permissions to its fields are automatically given to the method creating it. Middle: A possible implementation of bar that takes no permission to $x.f$, and therefore cannot modify it, which guarantees that foo is correct. Right: An alternative implementation of bar which takes foo's permission to $x.f$ and may therefore modify it, so that foo's postcondition does not hold.

to that location in its precondition, then the caller keeps the remaining permission throughout the call. On return, it may then assume that the heap location cannot have been changed by the call, since it kept a partial permission at all the times, which implies that noone else could have had a full permission, and therefore noone else could have changed the heap location.

Permission logics easily extend to the setting of concurrent programs, as shown in concurrent separation logic [169] and similar approaches [132]. In this setting, they automatically ensure that all verified programs must be data race free, since it is not possible that one thread modifies a heap location (which would require a full permission) while another thread simultaneously reads or writes the same location (which would also require some amount of permission, which could only exist if there was more than one full permission for that heap location in total, and that is never the case). This way, permission logics enable modular reasoning about concurrent programs.

[169]: O'Hearn (2007), 'Resources, concurrency, and local reasoning'

[132]: Leino et al. (2009), 'A Basis for Verifying Multi-threaded Programs'

As hinted at before, permission systems pose the challenge that methods which access a statically unbounded number of heap locations must require permission for all of these locations in their precondition. There are two main mechanisms for doing so:

- *Iterated separating conjunctions* [180] (sometimes called quantified permissions) allow specifying permissions amounts for each element in a set that is quantified over; for example, $\forall x \in s.\, \mathsf{acc}(x.f)$ denotes a full permission to the f fields of all references in set s.
- *Recursive predicates* [180] can recursively define assertions and therefore permission amounts. For example, a predicate $\mathit{list}(x)$ defined to be $\mathsf{acc}(x.\mathsf{next}) * \mathsf{acc}(x.\mathsf{val}) * x.\mathsf{next} \neq \mathsf{null} \implies \mathit{list}(x.\mathsf{next})$ recursively gives permission to all elements of a null-terminated linked list.

[180]: Reynolds (2002), 'Separation Logic: A Logic for Shared Mutable Data Structures'

[180]: Reynolds (2002), 'Separation Logic: A Logic for Shared Mutable Data Structures'

In implicit dynamic frames, recursive predicates and method specifications like pre- and postconditions must generally be *self-framing*, that is, they must only contain logical constraints about heap locations for which they also contain some permission. As an example, the assertion $\mathsf{acc}(x.f) * x.f = 5$ is self-framing, but the assertions $x.f = 5$ and $\mathsf{acc}(x.f) * y.f = 5$ are not.

2.1.3. Automated Verification Using Intermediate Verification Languages

Verification of imperative programs based on the principles shown above can be efficiently automated in practice as follows:

- In a first step, one can automatically generate proof obligations from programs annotated with specifications, e.g. using weakest precondition calculi [60] or symbolic execution [31]. This step can be performed completely automatically *if* some auxiliary specifications and annotations are present in the program to be verified, e.g., if all loops are annotated with loop invariants.
- In a second step, these proof obligations can be dispatched to theorem provers, in particular, SMT solvers like for example Z3 [156] or CVC4 [22], which can efficiently decide assertion entailment if all assertions are limited to first order logic and do not make use of undecidable theories like non-linear integer arithmetic.

The limitations imposed by both steps are what makes automated verification challenging; for example, it is often necessary to carefully design verification techniques to avoid having to use higher-order logic or to manage the size of the generated verification conditions. Sometimes, additional user input will be required to make automated verification viable. We provide two examples here, which respectively affect one of the steps mentioned above:

- When recursive predicates are used, existing verification tools typically require users to explicitly exchange a predicate for its body (i.e., unfold the predicate definition once) and vice versa. This is done to keep the generated verification condition finite, since in general, it is unknown how far a definition needs to be unfolded to prove a given method correct.
  The statements that perform this operation, typically called unfold and fold, are an example of *ghost code*, i.e., code that is not executed at runtime, but only affects *ghost state* that is only used for verification; an example of ghost state is the permission state in permission logics, which of course do not exist at runtime.
- When assertions contain universal quantification (which makes verification undecidable), SMT-solvers typically require *triggers* (also called *patterns*) [58], i.e., syntactical hints that indicate when and for which values universal quantifiers are to be instantiated. Choosing triggers is non-trivial, and bad choices can lead to too many instantiations, leading to bad performance or non-termination, or too few instantiations, leading to incompleteness.

Additionally, there is a considerable engineering effort associated with implementing the generation and dispatch of proof obligations, which, in principle, has to be reinvested for every new programming language and every new verification technique. *Intermediate verification languages* (IVLs) like Boogie [18], Why3 [82], and Viper [159] are one way to avoid

[60]: Dijkstra (1975), 'Guarded commands, non-determinancy and a calculus for the derivation of programs'

[31]: Berdine et al. (2006), 'Smallfoot: Modular Automatic Assertion Checking with Separation Logic'

[156]: Moura et al. (2008), 'Z3: An Efficient SMT Solver'

[22]: Barrett et al. (2011), 'CVC4'

[58]: Detlefs et al. (2005), 'Simplify: a theorem prover for program checking'

[18]: Barnett et al. (2005), 'Boogie: A Modular Reusable Verifier for Object-Oriented Programs'

[82]: Filliâtre et al. (2013), 'Why3 - Where Programs Meet Provers'

[159]: Müller et al. (2016), 'Viper: A Verification Infrastructure for Permission-Based Reasoning'

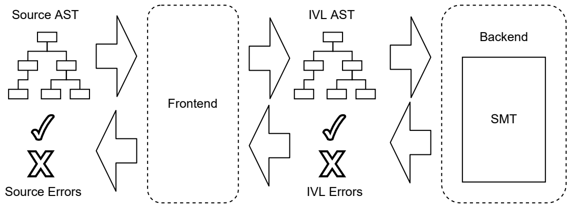


Figure 2.3.: Architecture of IVL-based verification tools. A new frontend is created for each new source language; the backend verifiers that compute verification conditions only have to be implemented once per IVL.

the potentially massive duplication of work that occurs when performing verification for multiple languages: They are simple (usually imperative) programming languages with integrated specification mechanisms. *Backend verifiers* for these languages implement techniques for proof obligation generation and dispatch them to SMT solvers or other theorem provers and therefore enable automated verification. This infrastructure can then be built upon to create a verifier for one or more actual programming languages: The source language only has to be *encoded* into the IVL by a *frontend*, which does not require computing proof obligations or directly communicating with theorem solvers. Figure 2.3 shows the resulting architecture of an IVL-based verifier.

Such an encoding can be performed in many different ways; after all, any encoding is sound if the encoded program verifies only if the original program is correct. However, typically, frontend encodings are somewhat similar to compiling from a higher-level to a lower-level language, with some peculiarities. Figure 2.4 shows a possible frontend encoding for the program shown before in Figure 2.1 into (a simplified version of) the Viper IVL [159]; from here on out, we will mark IVL encodings using red highlighting.

[159]: Müller et al. (2016), 'Viper: A Verification Infrastructure for Permission-Based Reasoning'

The encoded version has a simplified but similar structure to the original program; there are no more classes, but every class field corresponds to a top level field, and similarly every class method corresponds to a top level method. In addition, there are uninterpreted functions and types to model functionality that is built into the original language, in this case floating point operations.

The body of the encoded method has a similar structure and control flow as the original, but displays three patterns commonly used in frontend encodings: Compilation to simpler statements, overapproximation, and assertion of verification conditions.

- An example of compilation into simpler statements is the creation of a new instance of class `Z` in the original program, which is replaced by two statements in the encoded program (simply because the IVL is a simpler, smaller language), the first one being the assignment of a new reference, and the second the invocation of the constructor (which is now also a normal method).
- There are at least two examples of abstraction here: Class types have been replaced by raw reference types and applications of floating point operations are replaced by applications of uninterpreted functions. In other words, some information is abstracted away, i.e., lost, in the encoding because it is either not crucial for verification (the class types) or is intentionally not modeled in detail, since this would result in complex verification conditions (information about floating point operations and values). This is possible in a frontend encoding because it does not impact soundness: If the program can

```
// encoding of built−in types
type float

constant float_zero: float

constant float_pi: float

function float_mult(x: float, y: float): float
  requires y != float_zero

// encoding of class X
field X_f: Int

method X_bar(this: Ref, x: Ref) returns (res: Int)
  requires pre_bar(this, x)
  ensures post_bar(this, x, res)
{ ... }

// encoding of class Z
field Z_floatval: float

method Z_foo(this: Ref, x: Ref, y: Ref, i: Int) returns (res: Int)
  requires acc(x.X_f)
  ensures res > 1 * acc(x.X_f)
{
  x.X_f := 2

  var z : Ref
  havoc z
  Z_new(z)

  z.Z_floatval := float_mult(float_pi, i)

  // encoding of call to y.bar(y, x)
  assert y != null
  exhale pre_bar(y, x)
  var tmp: Int  // result variable
  inhale post_bar(y, x, tmp)

  res := x.X_f + 5
}

method Z_new(this: Ref)
  requires true
  ensures acc(this.X_f) * this.X_f = 0
```

Figure 2.4.: Possible frontend encoding of the Java program in Figure 2.1 into a simplified version of the Viper IVL [159]. The encoding declares an uninterpreted float type and (uninterpreted) operations on that type. Classes no longer exist, method and field names are made globally unique. pre_bar and post_bar are placeholders representing the pre- and postcondition of method bar. A **havoc** statement assigns an arbitrary value. As described before, **inhale** and **exhale** statements add permissions and assume constraints, and remove permissions and assert constraints, respectively.

be verified even though some potentially useful information is not given to the verifier (and the verifier, being conservative, always assumes the worst case about anything unknown), it must still be correct when taking this additional information into account. However, if the lost information is vital to the correctness of a program, such an encoding will be incomplete.

- The third pattern, the encoding of operations in terms of verification conditions, is in this case done for the call to method bar: In the encoding, there is no call; instead, there is an exhale of the precondition of the called method, and subsequently an inhale of the callee's postcondition in the new state after the call. Essentially, such an encoding directly instructs the verifier to check a sufficient condition of the correctness of the call: The callee method will be verified under the assumption that it gets the permissions mentioned in its precondition, and all logical constraints in said precondition hold in its initial state. Therefore, it is sufficient to exhale the precondition at the call site, i.e., to assert that all logical constraints in the precondition hold, to assert the caller owns all

permissions mentioned in the precondition, and to subsequently remove those permissions from the state of the caller (which implies losing any information about heap locations to the caller retains no permission). Subsequently, since the callee will be verified to establish all logical constraints in its postcondition, and to give all permissions mentioned in its postcondition to its callee, we may inhale the postcondition at the call site. That is, we add the permissions to the caller's state, and assume all logical constraints from its postcondition.
That is, the information that there is a call here is lost in the encoding, but instead the encoding directly tells the verifier which conditions it has to check to ensure that the encoded statement would execute correctly.

2.2. Modular Verification of Python Code

In the previous section, we have explained the basics of modularly verifying statically-typed languages like C or Java. When attempting to verify programs written in a dynamic language like Python, this standard "recipe" does not directly apply for two main reasons:

First, in such languages, there are often additional behaviors that simply are not possible in typical statically-typed languages, and that may require additional specifications and checks. Examples for this are the dynamic declaration of classes and methods during the program execution, more complex versions of basic statements like assignments, and a complex model of objects and attribute lookups.

Second, a lot of information that is statically available in statically-typed languages (such as, obviously, types) is simply not there in dynamic languages, or at least it is more difficult to compute. This can make it more difficult to reason about even standard language constructs correctly. The primary example for this, which we already mentioned in the introduction of this chapter, is to determine the target of a call: While languages like Java can provide verifiers with a static type for the receiver expression for every call, which determines either the called implementation (for statically-bound calls) or a set of possible implementations (for dynamically-bound calls, where the called implementation will either be the one in the static receiver type, or an override of said implementation), no such information is available in Python programs. This holds especially true if the goal is to reason modularly, when it is not possible to, e.g., find all callers of a method to determine all possible values of its arguments.

The goal of our verification approach for Python is not necessarily to support the entire language with no limitations, but to make pragmatic decisions in order to enable verification of typical user code (i.e., not necessarily libraries that exploit all available language features to create intuitive, DSL-like APIs, although we do also target libraries that do not use such language features) with acceptable specification and verification overhead. In order to reach this goal, we are prepared to restrict the supported language subset where necessary, and enrich the specification language and encoding in order to be able to express additional information and perform additional checks specific to dynamic languages, while applying standard specification and reasoning techniques wherever possible.

In this section, we will explain our approach to verifying Python code by detailing different challenges posed by the language, and subsequently describing our solutions. We will give all examples in Python 3, highlighted in blue, and encode examples in a simplified dialect of the Viper IVL [159], highlighted in red, as above. Our focus is to highlight the points that make Python verification *different* from static language verification, *not* to explain the full encoding from Python to Viper; for aspects that do not differ substantially between Python and static languages, we will presuppose that there is *some* standard encoding (and use it in examples). Similarly, we will highlight aspects of the Python language and the Viper IVL when they are important to the discussion, but we will not provide a thorough introduction to either language.

[159]: Müller et al. (2016), 'Viper: A Verification Infrastructure for Permission-Based Reasoning'

We present different parts of our verification technique grouped by the challenges they address:

- In Sec. 2.2.1, we describe our solution to the challenge of dynamic typing, i.e., our way of extracting sufficient type information to make verification feasible, while still supporting standard Python code patterns and keeping the required annotation overhead low. Our approach is to use an *optimistic* static type system as a first step, which ensures that the general structure of the program can be statically extracted, but is intentionally unsound in several ways. In particular, the system does not soundly check the definedness of fields and variables, which is vital to support most real Python code, and it allows users to locally opt out of static type checking where necessary by using casts. We then supplement this type system with sound checks in the verifier, which use the added precision of the verifier (as well as, where necessary, additional specifications) to soundly check variable and field definedness and ensure that all casts will always succeed.
- Python's use of structural subtyping can pose problems for verification, since (as we will explain) structural subtyping allows clients to pass objects with expected structure but arbitrary behavior to any method call, which makes it difficult to precisely determine the behavior of calls. In Sec. 2.2.2, we describe our solution to this problem: Like existing tools for static language verification, we instead use a nominal type system, and perform standard checks for behavioral subtyping (supplemented with some Python-specific checks). However, to mitigate the effects of this decision on our ability to verify existing code, we also support union types, which can often mimic the intended behavior of structural typing. Additionally, to make behavioral subtyping more permissive, we supplement it with a version of (the existing notion of) predicate families, which is designed to support standard Python coding patterns.
- In Sec. 2.2.3, we discuss the problems resulting from the fact that declarations of modules, classes, and methods in Python are executed as part of the program and potentially mixed with other code, instead of being globally available from the start. In particular, code may fail because it depends on classes or methods that are not yet defined, and class or method declarations themselves may fail if they refer to other elements that are not yet defined. We explain

which properties need to be checked and show how we encode the necessary checks.

- Sec. 2.2.4 discusses two sets of features we choose not to model: First, dynamic code generation, which is generally not supported by verification tools, and second, Python's complex object model. As for the latter, since precisely modeling Python's attribute lookup behavior would impact both verification performance as well as our ability to perform checks modularly, we instead use a simplified model of attribute lookups, and verify that code does not perform any operations that would expose the actual, more complex object model. We achieve the latter by preventing code from overriding certain built-in methods like __getattribute__ which implement attribute lookups.
- Finally, in Sec. 2.2.5, we give an overview of other complex language features in Python, and briefly detail how we support them, namely exception handling with finally-blocks, concurrency, complex assignments, and the handling of default arguments.

2.2.1. Type Checking

2.2.1.1. Challenge

Consider the simple piece of Python code in Figure 2.5, which is artificial, but serves to illustrate some of the fundamental problems with verifying dynamically-typed programs.

```
1  def useArguments(x, y, z):
2    if z:
3      l = y.f1 * 2
4    y.f2 = 78
5    if x.m1():
6      return l
7    return x.m2(y)
```

Figure 2.5.: Example of Python code whose behavior (and correctness) depends on the nature of the arguments it receives.

In a dynamic language like Python, where no static type checking is performed at compile time, and where even constructs like modules, classes, methods and constructors are all first class values that can simply be referred to via some name at runtime, it is not at all obvious what this method does when invoked without knowing the calling context.

For example, x and y could refer to modules, in which case f1, f2 would be global variables in those modules, and m1 and m2 could be either top-level methods or classes in said modules, if they exist at all. Alternatively, x and y could be instances of some classes and f1 and f2 could be fields of those classes, or even fields that exist only in these specific instances of those classes, and m1 and m2 could be methods. As another alternative, x could refer directly to a class, and m1 and m2 could refer to static methods, or *class methods*, a kind of method specific to Python that implicitly takes the class object itself as its first argument when invoked. Additionally, the assignment to field f2 in line 4 can either *modify* the value stored in an existing field or global variable, or it could *create* such a field or variable, if there currently is none.

As a result, a static verifier requires a lot of additional information to modularly verify this method with respect to some specification. In particular, in the context of a permission logic, it needs to prove:

- that the read values `f1`, `m1`, and `m2` (whether they be fields, global variables, classes, or methods) exist; if permissions are required to read them, it also has to prove that these permissions are present
- that the assignment to `f2` is allowed; that is:
  - if `f2` refers to a variable or field that already exists, the corresponding write permissions (and there may be different kinds of permissions for fields and global variables) are present,
  - conversely, if they do not currently exist and "traditional" write permissions therefore are not needed (since they are usually required to modify heap locations that already exist, whereas an assignment to a non-existing name in Python creates a new variable or field and would therefore lead to new permissions coming into existence), whether any permissions are needed to *create* such a field or variable
- that the preconditions of the invocations of `m1` and `m2` hold, which in turn requires knowing what those preconditions are and against which arguments those preconditions must be evaluated (since, as mentioned before, different kinds of calls have different implicit arguments)
- that, on return, the specified postcondition of `useArguments` (if any) holds; again, this requires knowing which postconditions may be assumed to hold after each of the calls in the method, if any permissions have come into existence as a result of these calls (e.g. because a call was to a constructor and led to memory allocation)
- that the local variable `l` is defined if it is read, which will also require information from the postconditions of invoked methods; in particular, it requires knowing that `x.m1()` does not return `True` unless `z` is also `True`.

Since, in modular verification, information from the method's call site is not taken into account when verifying a method, all this information has to come from annotations of the method itself, that is, either its specification (pre- and postcondition), or some other kind of annotation that fulfills similar purposes. In a statically-typed language, most of this information would be contained in or could be derived from the types of the arguments, and the types and specifications of the methods and fields that they refer to. Other information would not be needed, since some options simply are not possible in typical statically-typed languages, like passing a module/package as an argument, creating fields on the fly, and conditionally defining local variables.

2.2.1.2. Tradeoff

There are two obvious ways of approaching this problem: One could simply superimpose an ordinary static type system on top of a dynamic language to recover the necessary information, or one could perform all type checking as part of the verification process. Each approach has different advantages and disadvantages.

Option 1: Using a static type system before verification. It is possible to impose a classical static type system over an existing dynamic language. One could argue that, since static information about the program *must* be provided in some form to make static verification possible, superimposing a static type system is the obvious strategy: Static type systems come with comparably low annotation overhead (typically a single type name per parameter, method, field, and variable), and can be kept even lower when some types can be inferred (typically locally inside methods). Additionally, they are often decidable, i.e., they do not require user interaction or complex prover support to check if a program is type correct.

On the other hand, ordinary static type systems come with two main drawbacks, limited expressiveness and imprecise reasoning.

1. Ordinary static type systems are somewhat coarse in the properties they can *express*; for example, they will not be able to express that a parameter has type X if some expression is true, and some type Y otherwise (this would require dependent typing, which in turn would make the type system undecidable). As another example, they require any name in the program to have a single type, meaning that they generally do not allow a method that could take a parameter that is either a module, a class instance, or a class (like variable y in the example). Additionally, they typically require unambiguous declarations, meaning that it must be possible to statically determine, for example, which class a name refers to; this forbids, for example, conditional re-declarations like the one shown in Figure 2.6 (left), which would otherwise be legal in Python.
2. In order to be decidable, ordinary type systems overapproximate in their *reasoning* (even within the limited expressiveness they provide), and will therefore reject some programs that would not lead to type errors at runtime. For example, in the program in Figure 2.6 (right), variable y will contain an integer if x is true and an instance of class Y otherwise. The following code then multiplies the result in the former case, and invokes a method of class Y in the latter case, both of which would succeed at runtime (assuming class Y has a method bar). However, traditional *conservative* type systems (no matter if they infer types or require annotations) would have to overapproximate the type of variable y to be one that can contain either an integer, or an instance of Y; in Python 3, this would require assigning it the type **object**, which is the supertype of *all* other types. As a result, however, the type system can no longer determine that the subsequent usages of y are both safe under the conditions they happen. Similar limitations apply to other parts of type checking; in particular, type systems that check that each variable is initialized before its first read will generally require that a read variable has been initialized on *all* paths leading to a read, even infeasible ones. As a result, conservative static type systems would also reject the initial assignment to y in Figure 2.6 (right), since it reads variable a, for which typical type systems would not be able to determine that it has definitely been initialized. For typical Python programs, this is problematic, since they often do not initialize all local variables, global variables, or object fields on all paths.

```
class X:
  def foo(self):
    ...

if *:
  class X:
    def baz(self):
      ...
```

```
if x:
  a = 12
if x:
  y = a
else:
  y = Y()
if x:
  z = 2 * y
else:
  z = y.bar()
```

Figure 2.6.: Examples of code patterns disallowed by typical static type systems.

Table 2.1.: Overview of our hybrid type checking approach. First, the type checker performs optimistic type checking based on standard type annotations; then, the verifier soundly performs all remaining checks, using the information from both the type annotations and possibly additional information from program specifications such as pre- and postconditions.

	Type checker	Verifier
Type information specification	PEP 484 type annotations	PEP 484 type annotations, type information in specifications
Type checking	Sound according to type system, but allowing unchecked casts	Sound checks throughout, casts checked for safety
Nullness check	None	Sound
Local variable definedness check	Optimistic	Sound
Field definedness	Optimistic	Sound
Global name definedness	Optimistic	Sound

Option 2: Precise reasoning inside the verifier. The second options is to leave typing as dynamic as possible, perform type checking as part of the verification process (e.g., by generating verification conditions expressing type correctness and discharging them to an SMT solver or theorem prover), and supply all necessary information in some kind of specification language. This has the advantage of allowing users to express arbitrarily precise specifications in principle, and to have the full power of a verifier for reasoning about the program.

On the other hand, this approach typically requires much more complex specifications than simply annotating variables and parameters with type names (if the potential for superior expressiveness is to be exploited), and it requires potentially more complex checks in the verifier.

The former point is more complex in practice than it might first appear, especially when using an IVL: after all, *before* a verifier or prover can check whether proof obligations are fulfilled, a frontend or verifier first needs to be able to deduce what conditions need to be checked at all, and it needs to do this in such a way that it can encode the resulting proof obligations in first order logic if automation is desired. That is, a frontend typically needs to be able to deduce *some* information about a program, method or command statically (without consulting a powerful solver) in order to even generate the proof obligation it can give to a solver. This is an important part of what makes it challenging to develop automatable verification techniques.

2.2.1.3. Our Approach to Type Checking

Based on our central design goal, which is to enable verification of *typical* client Python code with manageable specification overhead, we decide to use a *hybrid* type checking approach that builds on an ordinary type system but also uses the verifier itself, as shown in Table 2.1. In particular, we use an *optimistic* type system to check any program that is to be verified; such a type system, being decidable, can automatically establish information about the basic structure of a program, but performs a number of checks in an *unsound* way in order to allow more programs that follow typical Python coding patterns to pass type checking. Subsequently, we can then compute verification conditions based on the knowledge of the program we get from the type system, and can use the verifier to precisely check all properties that were not checked by the type system in a sound way.

More concretely, we require all verified program to be type correct according to an existing static type system for Python, introduced in PEPs (Python Enhancement Proposals) 483 and 484 [181, 182], and further extended in PEP 544 [134]. These PEPs define a system for type annotations for Python 3 and the accompanying type system. Thus, while static typing of course remains optional in Python, the system we use is an official part of Python, and widely supported. The system uses type annotations in the ordinary Python syntax and in the form of comments both for methods and local variables. Type annotations (including casts) are completely ignored at runtime; their usage comes with very low runtime overhead (only that needed for parsing and evaluating, but not checking, the type annotations that are part of the AST). The system also defines a format for *stub files* that allow users to declare types for external libraries.

[181]: van Rossum et al. (2014), *PEP 484: Type Hints*

[182]: van Rossum et al. (2014), *PEP 483: The Theory of Type Hints*

[134]: Levkivskyi et al. (2017), *PEP 544: Protocols: Structural subtyping (static duck typing)*

The type system has support for both nominal and structural subtyping; we will explain in the next section why we limit ourselves to nominal subtyping only. The system supports generic classes and functions, with upper bounds for type parameters; it supports `Optional`-types which indicate that references may be `None`, it supports casts, and it supports union types, which will also be discussed in the next section.

Crucially, this type system is intentionally unsound in three major ways:

1. It allows the use of type casts, which, in Python, are *not* checked at runtime, and therefore only represent directives to the type checker to make assumptions.
2. It *optimistically* checks the definedness of local and global variables, as well as object fields. We have already seen an example of a conditionally-defined local variable (variable `a` in Figure 2.6 (right)). Similarly, in Python, fields may exist on some instances of a class but not others, since they are not declared at the class level, but come into existence when an assignment to the field is made. As a result, it is possible that a field is only *conditionally* created in the class constructor, is created in a setter method that has been called on some instances but not others, or has been *deleted* on some instance.
3. It allows calls and field accesses whose receivers have `Optional`-types even when it cannot prove that the receiver will not be `None`.

As a result, the type system will reject some valid Python programs, but will accept most *typical* Python code, provided that correct type annotations are provided.

In particular, since the type system requires every variable to have a single type, it is impossible, for example, to have a variable like `y` in the example above that might be either a module or a class or an integer. Thus, programs like the one in Figure 2.6 (left) will no longer be allowed, however, we argue that code patterns like these are not common in practice, and would generally be considered hacky and bad style.

However, while the type system results in more limited *expressiveness*, we can almost completely mitigate the drawback of imprecise *reasoning* by delegating crucial checks to the verifier. That is, we make type information from type annotations available to the verifier, and then use the verifier to check the validity of all type casts in the program. Additionally, we use the verifier to soundly and precisely check that all read variables and fields are actually initialized. As a result, the program shown in Figure 2.6 (right) can be verified if casts are added, as shown in Figure 2.7: Unlike a type system, the verifier is sufficiently precise to show that variable a is definitely defined when it is read, and that the casts of variable y to different types will always succeed; similarly, it will be able to verify that receivers of calls and field accesses are not None with high precision. Additionally, a frontend that encodes the program into an IVL can use information from the type system to know that the call to bar has a receiver of type Y. While this information from the type system is based on unchecked information from a cast, the verifier can ensure that the cast will succeed *before* using any assumptions about the receiver type of the call.

Thus, since programmers can use casts at any point in the program, they can essentially circumvent the limited precision of the used type system completely, making the full power of the verifier available for type checking. The frontend, however, still has static type information available for every statement and name in the program, and can use it to see which method a call refers to, and which proof obligations need to be checked at which point in the program.

```python
if x:
  a = 12
if x:
  y = a
else:
  y = Y()
if x:
  z = 2 * cast(int, y)
else:
  z = cast(Y, y).bar()
```

Figure 2.7.: Program from Figure 2.6 (right) with added casts.

Since we build on an existing type system that is fully described in the aforementioned PEPs and already implemented in the mature Mypy type checker [125], we will not describe the used static type system any further. In the following subsections, we describe our encoding of type information and type checks in the verifier: We will first describe how we model type information and type checks, and subsequently, how we soundly check the definedness of local variables and fields.

[125]: Lehtosalo et al. (2017), *Mypy - Optional Static Typing for Python*

2.2.1.4. Encoding of Type Information and Checks

In Python 3, all values, including "primitive" values like integers, are objects, and have the common supertype object (which we exploited in the example in Figure 2.7). As a result, *any* value can be assigned to a variable of type object.

In the encoding, we therefore encode all values as references, and distinguish values of different types by explicitly adding type information. We do the latter by declaring a function typeof which maps every reference to its dynamic Python type (which we represent as a value of an uninterpreted type PyType). Every known type, i.e., every built-in Python type as well as every type resulting from a class declaration, corresponds to a PyType value.

We then ensure that every newly created value is associated with type information: in the example in Figure 2.7, we assume that the newly-created instance of class Y has type Y by assuming typeof(r) == Y(), where r is the reference variable containing the instance. In the other branch, we use a function int_box that takes an integer value (in this case 2), maps it into a reference value (more on that later), and declares in its postcondition that the reference has type int.

The casts in the following conditional then assert the respective conditions issubtype(typeof(y), int) and issubtype(typeof(y), Y), where issubtype is a function which encodes the reflexive and transitive subtyping relation (we will describe the modeling of this function later). Using the subtyping relation here is important because the casts of course do not require y to have *exactly* the types int or Y, but any subtype thereof.

In order to support such casts anywhere in the program, type information must also be passed along between different parts of the program, e.g., across method boundaries. That is, for example, each method parameter x of type T on the Python level is encoded as a reference-typed parameter in the IVL with a corresponding precondition issubtype(typeof(x), T). Similarly, for every permission assertion acc(x.f) in a precondition, postcondition or loop invariant, the encoding adds a type assertion, so that the resulting assertion on the IVL level is acc(x.f) * issubtype(typeof(x.f), T), where T is the type of field f. This is how type information from type annotations is made available everywhere in the program.

Additionally, users may write specifications (e.g., pre- and postconditions) that contain *additional* type information. For example, in the code example in Figure 2.7, if the last four lines were moved to a separate method that receives x and y as arguments, y's type would have to be declared as object. To be able to prove that the casts succeed, one would then need to add a precondition describing more precise type information. That is, one could write a precondition stating that if x is true, then y has type int, and otherwise its type is Y; this can be expressed using the two implications x ==> isinstance(int, y) and not x ==> isinstance(Y, y). This information, which is more precise than type information that could be expressed using the type system, can be easily encoded into a precondition on the IVL level by translating Python's isinstance expressions into issubtype constraints. The verifier can then make use of this information to prove the casts correct.

The encoded type information is not only used to enable the checking of casts, but can also be used in the encoding of statements (meaning that some statements may have different proof obligations depending on the type of the involved expressions), and is important for the modeling of various built-in functions. For example, the aforementioned int_box function, which maps integer values to reference values, has an inverse function int_unbox that extracts the integer value again. However, the latter function is clearly not a total function from reference values to integers, but is only partially defined, for references of type integer. Since our encoding makes type information available, we can now explicitly declare this in the precondition of the unboxing function, leading to the two definitions shown in Figure 2.8.

```
function int_box(prim: Int): Ref
  ensures typeof(result) == int
  ensures int_unbox(result) == prim

function int_unbox(box: Ref): Int
  requires issubtype(typeof(box), int)
```

Figure 2.8.: Boxing and unboxing functions used to wrap integer values into references; the unboxing function is partial, and is only defined for integer-typed references.

Note that this encoding means that all type checking performed by the type system is essentially repeated on the verifier level. For example, when calling any method, at the call site, the verifier will have to prove that the supplied arguments actually have the required types, since this is part of the called method's precondition. To improve efficiency, one could omit these checks, since they are already known to succeed from the type system. However, we do not currently do this, for two reasons: First, some specifications may not be well-formed unless all type information is present (e.g. because they contain applications of partial functions, as mentioned before). Second, re-doing all type checks means that we do not rely on the correctness of the type checker, which enhances the trust in our verification result.

The final piece of the puzzle is now the modeling of the subtype relation. A crucial requirement of modeling subtyping in our setting is, since our verification must be modular, that we must always assume that there may be additional types that are currently unknown. Consider, for example, a scenario with a known class X which has exactly one known subclass, SubX. If a reference r is known to be of type X (meaning it is an instance of X or any subtype of X), and it is known from elsewhere that r's type is not *exactly* X (for example because execution has entered a branch with the condition type(r) is not X), it must *not* be possible to conclude that r has type SubX; it may for example be an instance of some other class OtherSubX that is currently now known.

As a result, the subtype relation we need to model is one between an unbounded number of possible types, which is constrained by the typing relations known to us from the (part of the) program we know.

As shown in Figure 2.9, we model the subtype relation issubtype with the help of two additional relations, extends and isnotsubtype. We model known types as constants, and use the extends function to model all direct subtype relations between known types. Generic types are not encoded as constants, but as functions. Encoding the subtype relation using the extends-function then requires a universal quantifier.

Additional axioms which we will not show here are used to express that all values representing different types are non-equal, that is, for example, that the constant int is not equal to the constant X, which in turn is not equal to list(int).

```
type PyType

function issubtype(sub: PyType, super: PyType): Bool
function extends(sub: PyType, super: PyType): Bool
function isnotsubtype(sub: PyType, super: PyType): Bool

constant object: PyType
constant int: PyType
constant X: PyType
constant SubX: PyType
constant OtherSubX: PyType
function list(arg: PyType): PyType

axiom extends_relations {
  extends(int, object) *
  extends(X, object) *
  extends(SubX, X) *
  extends(OtherSubX, X) *
  forall arg: PyType :: extends(list(arg), object)
}
```

Figure 2.9.: Basic type system encoding.

We then use more axioms to define the issubtype relation. Of course, if one class extends another, this implies that it is also a subtype of the other class. Additionally, the subtype relation is reflexive and transitive. These aspects are expressed by the first three axioms in Figure 2.10.

```
axiom extends_implies_subtype {
  forall sub: PyType, sub2: PyType ::
    { extends(sub, sub2) }
    extends(sub, sub2) ==> issubtype(sub, sub2)
}

axiom issubtype_reflexivity {
  forall t: PyType :: { issubtype(t, t) } issubtype(t, t)
}

axiom issubtype_transitivity {
  forall sub: PyType, middle: PyType, super: PyType ::
    { issubtype(sub, middle),issubtype(middle, super) }
    issubtype(sub, middle) * issubtype(middle, super) ==> issubtype(sub, super)
}

axiom issubtype_antisymmetric {
  forall sub: PyType, super: PyType ::
    { issubtype(sub, super) } { issubtype(super, sub) }
    issubtype(sub, super) * sub != super ==> !issubtype(super, sub)
}

axiom issubtype_exclusion {
  forall sub: PyType, sub2: PyType, super: PyType ::
    { extends(sub, super),extends(sub2, super) }
    extends(sub, super) * extends(sub2, super) * sub != sub2 ==> isnotsubtype(sub, sub2)
        * isnotsubtype(sub2, sub)
}

axiom issubtype_exclusion_propagation {
  forall sub: PyType, middle: PyType, super: PyType ::
    { issubtype(sub, middle),isnotsubtype(middle, super) }
    issubtype(sub, middle) * isnotsubtype(middle, super) ==> !issubtype(sub, super)
}
```

Figure 2.10.: Axioms defining the issubtype relation. The expressions in curly braces are *triggers* that SMT solvers use to decide when to instantiate the quantifiers.

However, this information is not sufficient in practice: It is often critical to know that one type is *not* a subtype of another. We add three axioms to encode this fact, based on the following insights: First, the subtype relation is antisymmetric, second, if two different classes A and B both extend class C, then all subtypes of A are not subtypes of any subtype of

B. Both are expressed in the last three axioms in Figure 2.10. Note that the latter property holds only in type systems without multiple subtyping, like the one we are modeling.

2.2.1.5. Encoding of Local Definedness Checks

As explained above, the type system we use performs optimistic checking of the definedness of local variables, that is, it does not soundly ensure that variables are necessarily defined when they are read. Since reading an undefined variable leads to a runtime error, we need to perform this check soundly in the verifier.

To accomplish this, we explicitly add information in the encoding that models which names are currently defined. We do this as follows: Every local variable name is mapped to an integer identifier; we will refer to the integer identifier of name n as <n> from now on. We use an uninterpreted function isDefined, shown in Figure 2.11, to check which names are defined. When a local variable named n is assigned to, we assume isDefined(<n>) right *after* the assignment.

```
function isDefined(n: Int): Bool

function checkDefined(n: Int, v: Ref): Ref
  requires isDefined(n)
  ensures result == v
```

Figure 2.11.: Functions used to model and check the definedness of local variable names.

A second function checkDefined shown in the same Figure is used to check if read variables are defined. Every access of a variable x on the Python level is encoded by the expression checkDefined(<x>, x'), where x' is the reference-typed local variable on the IVL level that represents the Python variable. Evaluating this expression requires first showing that the precondition of checkDefined is fulfilled, i.e., that isDefined(<x>) actually holds. If the verifier cannot prove that this is the case, this will lead to a verification error. Otherwise, the value of this expression is simply the value of the variable, as it should be.

Note that we do not support *deleting* local variables, since this is rarely done in practice and virtually never necessary. Also note that, as a result of the absence of deletions, performing the definedness-check for *every* access of a variable could theoretically often be avoided, since variables will never become un-defined once they come into existence; similarly, one could often statically determine that local variables are unconditionally defined, and therefore no check in the verifier is necessary at all. These are possible optimizations that we currently do not perform.

The system we use for *global* variables is different and will be explained in Sec. 2.2.3.

2.2.1.6. Encoding and Specification of Fields

The definedness of fields is a similar problem: Our static type system checks optimistically that accessed fields *may* exist, which allows us to support code that defines fields conditionally, like the fields f2 (which is created if x is true) and f3 (which is created when set_f3 is called) in the code snipped in Figure 2.12.

```
class X:
  def __init__(self, x: bool) -> None:
    self.f1 = 1
    if x:
      self.f2 =2

  def set_f3(self, v: int) -> None:
    self.f3 = v

  def get_field(self, x: bool) -> int:
    if x:
      return self.f2
    return self.f3
```

Figure 2.12.: Dynamic creation of fields.

Thus, the verifier must perform sound checks that ensure that accessed fields definitely exist. That is, when verifying get_field, we must show that the current self object has the field f2 or f3 (depending on the value of x). Since we want to use a permission logic, we *also* have to have a permission for accessing said field. Note, however, that the latter requirement subsumes the former: If permissions exist only for fields that also exist, and accessing fields requires having the respective permission, then the permission check is sufficient to ensure that accessed fields actually exist. As a result, no additional specification is needed here: If a permission exists only for fields that have actually been created, then the problem is already solved.

What's left to do now is to create a mechanism that soundly creates permissions to fields when they are initially assigned. Clearly, it is sound to give the code that first creates a field the permission to the newly-created field. However, if a field already exists on an object, then writing to that field must *require* that the method modifying the field *already has* the field permission; it must *not* create another instance of the permission.

Our solution for this problem is as follows: First, we do not allow adding fields to objects outside their classes. That is, an instance of a class may have some subset of the fields that are assigned anywhere in the code of the class or any of its superclasses, but no others. Second, creating a field requires a special permission. That is, there is a different kind of permission, which we denote by MayCreate(e, f) on the Python level, that represents the right to *create* the (currently non-existing) field f on the object e, but *not* the right to read said field (since it currently does not exist). Assigning to this field will create it, and exchange the permission MayCreate(e, f) for an ordinary field permission acc(e.f), which allows both reading from and writing to the (now existing) field. Deleting the field reverses the process, i.e., it consumes an ordinary permission acc(e.f) and exchanges it for a permission MayCreate(e, f) that allows re-creating the field.

Writing to a field now requires *either* having the MayCreate(e, f) permission, in which case the write creates the field exchanges the create-permission for an ordinary permission acc(e.f), or having the ordinary write permission acc(e.f), in which case the permission state does not change.

In the encoding, we encode field creation permissions using an uninterpreted predicate MayCreate(x: Ref, n: Int), where the first argument is the receiver object and the second is the integer-encoded name of the field, so that the permission MayCreate(e, f) on the Python level is represented by the predicate MayCreate(e, <f>) on the IVL level. A write e1.f = e2 on the Python level is then encoded as shown in Figure 2.13.

Figure 2.13.: Encoding of a field write. The expression perm(MayCreate(e1, <f>)) returns the amount of MayCreate-predicates with the shown arguments held by the current method, that is, the condition will be true if the method holds such a permission.

```
if (perm(MayCreate(e1, <f>)) >= 1) {
  exhale acc(MayCreate(e1, <f>))
  inhale acc(e1.f)
}
e1.f := e2
```

That is, if a permission to create the field is currently held (which is only the case if the field does not already exist), it is exchanged for an ordinary permission, then the write is performed as usual. If neither permission is held, then the write cannot be verified.

Class constructors implicitly get the permission to create all fields in the class as their precondition; this is possible because the set of possible fields for each class is bounded and statically known, as described above. The constructors may then create *some* of the fields, and potentially pass on the permission to create others back to their caller in their postcondition using MayCreate-assertions.

Finally, we add a new specification construct that represents the right to write to a field, whether it currently exists or not. This kind of permission is useful for example for setter methods, like set_f3 in the example in Figure 2.12: This method can be called on an object which currently does not have an f3 field, in which case it requires the permission MayCreate(self, f3), or it can be called on an object that already has a field f3, in which case it requires permission acc(self.f3). We use a permission called MayWrite(e, f), which represents the right to write to the field f on instance e, i.e., it represents *either* MayCreate(e, f) if the field does not exist, or acc(e.f) otherwise.

In the encoding, this new permission is represented differently in different situations:

- When *getting* (i.e., *inhaling*) such a permission (e.g., in the precondition of a method), the verifier has to assume that it either has the permission to create or the ordinary field permission, but does not know which. In this case, the permission is encoded as nondet() ? MayCreate(e, f) : acc(e.f), where nondet() is a boolean expression whose value is unconstrained and unknown to the verifier (so that it has to prove the method can be verified in both cases)[2].
- When *giving away* (i.e., *exhaling*) such a permission (e.g. when calling a method that requires it in its precondition), the method who has to do so can choose whether it gives away a create-permission or an ordinary one, since either will be sufficient for the called method to succeed. Thus, we encode the permission using the expression perm(MayCreate(e, <f>)) >= 1 ? MayCreate(e, <f>) : acc(e.f), i.e., if the caller has the create-permission, it will give up that permission, otherwise it will give up an ordinary permission.

2.2.1.7. Summary

The example in Figure 2.14 uses a combination of the aspects discussed before, and is annotated with type annotations. Its IVL encoding (exclud-

[2] In the actual encoding, we use an unconstrained boolean function for this purpose.

ing the declarations of type constants, functions, and axioms) is shown in Figure 2.15.

```
class X:
    def __init__(self, flag: bool) -> None:
        # ensures acc(self.f1)
        # ensures flag ==> acc(self.f2)
        # ensures !flag ==> MayCreate(self, f2)
        if flag:
                x = 15
        self.f1 = "initialized"
        if flag:
            self.f2 = x

    def set_field(self, val: int) -> None:
        # requires MayWrite(self, val)
        # ensures acc(self.val) && self.val == val
        self.f2 = val

    def get_field(self) -> int:
        # requires acc(self.val)
        # ensures acc(self.val) && self.val == result
        return self.f2


def do_stuff(args: Tuple[object...]) -> str:
        # requires len(args) == 2
        # requires args[0] ==> isinstance(str, args[1])
        # requires not args[0] ==> isinstance(X, args[1])
        if args[0]:
            return cast(str, args[1])
        return cast(X, args[1]).f1
```

Figure 2.14.: Example of a Python program using type annotations, casts, and specifications according to our model. The constructor returns a field creation permission, and the setter requires a MayWrite-permission. The precondition of do_stuff contains additional type information outside the type system, which allows proving the casts in the method's body correct.

2.2.2. Subtyping

The second major challenge in the static verification of Python is the use of *structural typing*. The example in Figure 2.16 (left) illustrates the problem.

The three classes A, B (which is a subclass of A), and C (which is declared independently of both A and B) all offer a method foo. Method client expects a single object as an argument, and calls method foo on it. We will assume that client was written with the intention to receive instances of either A, B, or C as arguments. However, in standard Python, with dynamic and structural typing, client can be called with any object that has a foo method without raising a type or attribute error.

Clearly, though, client is not correct for any possible argument that has a foo method with a matching signature. Take, for example, class D, shown in Figure 2.16 (right). Structurally, D is identical to A, but its behavior is not: While the foo methods in A, B, and C can be called with the argument 2 without raising an error, doing so on an instance of D will result in a division by zero. In other words, client is correct only if its argument fulfills certain expectations about the behavior of its foo method. In a verification setting, we say that client expects that its argument has a foo method whose precondition is fulfilled by the argument 2.

Clearly, this fact must be represented in the signature or specification of client. In languages that use nominal subtyping, a standard way of solving this problem is to enforce *behavioral subtyping* [138], i.e., to enforce that all methods in subtypes comply with the specifications of the

[138]: Liskov et al. (1994), 'A Behavioral Notion of Subtyping'

```
function cast(t: PyType, r: Ref): Ref
  requires issubtype(typeof(r), t)
{ r }

method X___init__(self: Ref, flag: Ref)
  requires issubtype(typeof(self), X)
  requires issubtype(typeof(flag), bool)
  ensures acc(self.f1) ∗ issubtype(typeof(self.f1), int)
  ensures bool_unbox(flag) ==> acc(self.f2) ∗ issubtype(typeof(self.f2), str)
{
    var x: Ref
    if (bool_unbox(flag)){
            x = int_box(15)
            assume isDefined(<x>)
    }
    if (perm(MayCreate(self, <f1>) >= 1) {
      inhale acc(self.f1)
    }
    self.f1 = str_box(<initialized>)
    if (bool_unbox(flag)){
      if (perm(MayCreate(self, <f2>) >= 1) {
        inhale acc(self.f21)
      }
        self.f2 = isDefined(<x>, x)
    }
}

method X_set_field(self: Ref, val: Ref)
  requires issubtype(typeof(self), X)
  requires issubtype(typeof(val), int)
  requires [∗ ? (acc(self.val) ∗ issubtype(typeof(self.val), int)) : MayCreate(self, <val>),
            perm(MayCreate(self, <val>)) >= 1 ?
              MayCreate(self, <val>) :
              (acc(self.val) ∗ issubtype(typeof(self.val), int))]
  ensures acc(self.val) ∗ issubtype(typeof(self.val), int) ∗ self.val == val
{
  if (perm(MayCreate(self, <val>) >= 1) {
    inhale acc(self.val)
  }
  self.val = val
}

method X_get_field(self: Ref) returns (res: Ref)
  requires issubtype(typeof(self), X)
  requires acc(self.val) ∗ issubtype(typeof(self.val), int)
  ensures issubtype(typeof(res), int)
  ensures acc(self.val) ∗ issubtype(typeof(self.val), int) ∗ res == self.val
{
  res := self.val
}

method do_stuff(args: Ref) returns (res: Ref)
  requires issubtype(typeof(args), tuple(object)
    requires tuple_len(args) == 2
    requires tuple_get(args, 0) ==> issubtype(typeof(tuple_get(args, 0)), str)
    requires tuple_get(args, 1) ==> issubtype(typeof(tuple_get(args, 1)), X)
    ensures issubtype(typeof(res), str())
{
    if (object___bool__(tuple_get(args, 0))) {
        res := cast(str, tuple_get(args, 1))
        goto end
    }
    var tmp: Ref
    tmp := cast(X, tuple_get(args, 1))
    res := tmp.f1
    goto end
    label end
}
```

Figure 2.15.: Partial IVL encoding of the Python code in Figure 2.14. Casts are checked using the partial function cast. Type information from the type system is encoded into pre- and postconditions. The MayWrite-assertion in the precondition of set_field is encoded using an inhale-exhale assertion of the form [P, Q], which behaves like assertion P when inhaled and like assertion Q when exhaled.

```python
class A:
  def foo(self, x: int) -> int:
    return 4 // x

class B(A):
  def foo(self, x: int) -> int:
    return 3 // x

class C:
  def foo(self, x: int) -> int:
    return 2 // x

def client(a: A) -> int:
  # expects A, B, or C
  x = 2
  return a.foo(x)
```

```python
class D:
  def foo(self, x: int) -> int:
    return 5 // (x - 2)

client(D())
```

Figure 2.16.: Example of Python code using structural subtyping (left), and outside code that may break the existing code (right).

supertype methods they override. In our example, assuming A.foo has the precondition x != 0, this would mean enforcing that foo in its subclass B (and all other subclasses) has the same or a weaker precondition, which is the case here. Then, client could declare its argument type to be A, and could be verified because of the knowledge that all possible arguments have a foo method with a precondition that is weaker than x != 0, which is definitely fulfilled by the argument 2. As a result, it would still be possible to call client with instances of A and B, but no longer with instances of C, since the latter is not explicitly declared to be a subclass of A and therefore, in a nominal type system, not its subtype. Such an approach would be incomplete, since passing instances of C will not result in an error in our program, and should therefore be allowed.

When using structural subtyping instead, relying on behavioral subtyping is infeasible altogether: Class D is a structural subtype of class A simply by virtue of having the same structure, but clearly, one cannot demand that D's methods must fulfill the specification of class A, a class that might be entirely unrelated and not even known to the implementer of D. Thus, when using structural typing, behavioral information would have to be specified independently of type information: In our example, method client could (in addition to a type signature that enforces that its argument has a method named foo) have a precondition stating that the precondition of the foo method must be fulfilled by the value 2.

Such an approach is possible in practice and would provide maximum flexibility (in fact, such an approach is often used to specify function arguments in the preconditions of higher order functions) [114], but it results in a large specification overhead and complex verification conditions, which in turn affects verification performance.

[114]: Kassios et al. (2010), 'Specification and verification of closures'

We therefore propose using a compromise solution:

1. We use *nominal* subtyping instead of Python's standard structural typing, and additionally enforce behavioral subtyping. As described above, this standard approach makes it easy to prove client correct when annotating its argument with type A, but would, by itself, come at the cost of disallowing calling client with instances of C and therefore be incomplete.
2. To mitigate this incompleteness, we support *union types*. That is, we allow annotating the parameter of client with type Union[A, C] to state that any instance of (a subclass of) A *or* (a subclass of) C may

be passed. The body of client is then verified under the assumption that a may be an instance of A *or* of C.

3. Finally, we relax the requirements of behavioral subtyping, which, when used naively, is problematic when used in the context of permission logics (we will show why in an example later in this section), by supporting *predicate families* [171] as specification constructs.

[171]: Parkinson et al. (2005), 'Separation Logic and Abstraction'

The result is a verification technique that (through the use of union types) can accommodate many examples that make use of structural typing, like the one shown in Figure 2.16 (left), but comes with a lower annotation overhead and less complex verification conditions compared to a solution that explicitly specifies the expected behavior of arguments.

Note, however, that union types are not a perfect replacement of (desired) structural subtyping; in particular, they can be used only if a method (like client in our example) knows all possible types it expects as arguments, and can therefore list them explicitly in the union type. If the author of client did not know about class C and therefore did not list is as a legal argument type, it would not be possible to pass instances of C without adapting the signature of client.

In the remainder of this section, we will explain each part of our solution in more detail: We will first discuss the parts of our type system that deal with subtyping and union types, and then describe how we integrate union types into the type encoding in our verifier. Next, we describe how we encode behavioral subtyping checks (which contain some Python-specific elements that would not be needed in languages like Java), and subsequently explain our design and encoding of predicate families (which we also adapt to be best suited to support typical Python coding patterns).

2.2.2.1. Nominal Type System

As mentioned before, for the initial optimistic static type checking, we use the type system outlined in PEPs 483 and 484 [181, 182], which is a nominal type system. There is an extension to this type system, defined in PEP 544 [134], which adds the option to use structural typing; we support *only* nominal type hints according to the former system, *not* structural ones according to the latter.

[181]: van Rossum et al. (2014), *PEP 484: Type Hints*
[182]: van Rossum et al. (2014), *PEP 483: The Theory of Type Hints*

[134]: Levkivskyi et al. (2017), *PEP 544: Protocols: Structural subtyping (static duck typing)*

PEPs 483 and 484 already contain support for union types; we adopt the notation proposed in this PEP, where Union[A, B, C] denotes the union of types A, B, and C; such a union can have any number of *component types*. Type checking for union types is conservative: when a field read or a method call is performed on a receiver whose type is a union type, the type system ensures that the read or call is possible on *all* component types. Similarly, the type system checks that method overrides in subtypes are valid (in terms of their signatures); the example in Figure 2.17 illustrates both valid and invalid overrides of method foo in class Y:

- The override in SubY1 is valid: It makes a parameter type more general (contravariance), a return type more specific (covariance), it does not change parameter names, and it adds the option to call the method with three instead of one or two arguments, but does not remove any option that exists in the superclass.

```python
class SuperX:
  pass

class X(SuperX):
  pass

class SubX(X):
  pass

class Y:
  def foo(self, x: X, s: str = "") -> X:
    ...

class SubY1(Y):
  def foo(self, x: SuperX, s: str = "", s2: str = "") -> SubX:  # valid
    ...

class SubY2(Y):
  def foo(self, z: X, s: str = "") -> X:  # invalid
    ...

class SubY3(Y):
  def foo(self, x: SubX, s: str = "") -> X:  # invalid
    ...

class SubY4(Y):
  def foo(self, x: X, s: str = "") -> SuperX:  # invalid
    ...

class SubY5(Y):
  def foo(self, x: X) -> X:  # invalid
    ...
```

Figure 2.17.: Example of valid and invalid method overrides.

- The override in SubY2 is invalid, since it changes the name of the parameter x to z, so that a call with a named argument y.foo(x=X()) would be invalid on the subclass.
- The override in SubY3 is invalid, since it makes the type of the parameter x more precise, so that a call y.foo(X()) would be invalid on the subclass.
- The override in SubY4 is invalid, since it makes the method's return type more general, so that code that calls the function and expects an instance of X might get an object it does not expect when the call is made on an instance of the subclass.
- The override in SubY5 is invalid, since it removes the option to call the method with two arguments, so that a call with a named argument y.foo(X(), "Hello") would be invalid on the subclass.

2.2.2.2. Union Type Support

Like for all other types, we represent type information for union types in the encoded program. Union types are integrated into our existing type encoding as follows: Each union type is (like every other generic type) modeled using a function; the function takes the component types as parameters. We then use two straightforward axioms to model subtype relations between union types and other types. The code snippet in Figure 2.18 shows the function declaration and the axiom for union types with two components; if the verified programs uses union types with different numbers of components, the required function declarations and axioms can be generated on demand.

```
function union_type_2(arg_1: PyType, arg_2: PyType): PyType

axiom union_subtype_2 {
  forall arg_1: PyType, arg_2: PyType, X: PyType :: { issubtype(X, union_type_2(arg_1,
        arg_2)) } issubtype(X, union_type_2(arg_1, arg_2)) == (issubtype(X, arg_1) ||
        issubtype(X, arg_2))
}

axiom subtype_union_2 {
  forall arg_1: PyType, arg_2: PyType, X: PyType :: { issubtype(union_type_2(arg_1,
        arg_2), X) } issubtype(union_type_2(arg_1, arg_2), X) == (issubtype(arg_1, X) *
        issubtype(arg_2, X))
}
```

Figure 2.18.: Function and axioms defining subtyping for a union type with two components.

The first axiom states that a type is a subtype of a union type if it is a subtype of at least one of the components; the second states that a union is a subtype of another type if all its component types are subtypes of that other type. Using these axioms, the verifier can deduce, for example, that a parameter of type Union[A, C] must either be an instance of A or of C (or their subtypes).

Once this information is available to the verifier, we can use it to encode calls and field accesses: The call a.foo(x) from the initial example, where a has type Union[A, C], will be encoded as shown in Figure 2.19 (left).

```
if (issubtype(typeof(a), A)) {
  A_foo(a, x)
} else {
  C_foo(a, x)
}
```

```
if (issubtype(typeof(a), B)) {
  B_foo(a, x)
} else {
  A_foo(a, x)
}
```

Figure 2.19.: Encoding of a call to method foo on a receiver of type Union[A,C] (left) and Union[A,B], where A and C are unrelated types and B is a subtype of A.

That is, the encoding checks which component of the union the actual receiver type is a subtype of, and then verifies it like a call to that type's version of the called method. Note that the call to C_foo will have a precondition that ensures that the receiver actually has type C, which is why the else-case can be left unguarded. For unions whose components are subtypes of each other, our encoding always checks the more specific types first; that is, for the type Union[A, B], where B is a subtype of A, the encoding of a call to foo will be that shown in Figure 2.19 (right), so that the case where the subclass method is invoked (which will have more precise specifications) is always explicitly taken into account; if the cases are checked in the reverse order, the else branch would be dead.

Similarly to method calls, a field access a.f on a receiver of type Union[A, C] will be encoded as a conditional issubtype(typeof(a), A) ? a.A_f : a.C_f.

This way of precisely encoding calls and field accesses using type information results in precise verification of some corner cases like the one shown in Figure 2.20. Here, the verifier knows that the call to foo is performed only if a has type C, and therefore will verify the client *only* against the specification of foo in C.

```
1  def client(a: Union[A, C]) -> None:
2    if isinstance(A, a):
3      return
4    a.foo(x)
```

Figure 2.20.: Example using union types; the call to foo in line 4 will only be executed on receivers of type C.

2.2.2.3. Behavioral Subtyping Checks

The notion of behavioral subtyping [138] states that subtypes must adhere to the specification of their supertypes. In our setting, this means that when a subclass method overrides a superclass method, its precondition must not get stronger, and its postcondition must not get weaker.

[138]: Liskov et al. (1994), 'A Behavioral Notion of Subtyping'

```
class X:
  def bar(self, i: int) -> int:
    # requires i > 5
    # ensures result > 5
    ...

class SubX1(X):
  def bar(self, i: int) -> int:  # valid
    # requires i > 0
    # ensures result > 10
    ...

class SubX2(X):
  def bar(self, i: int) -> int:  # invalid
    # requires i > 10
    # ensures result > 5
    ...

class SubX3(X):
  def bar(self, i: int) -> int:  # invalid
    # requires i > 5
    # ensures result > 0
    ...
```

Figure 2.21.: Example of Python method overrides that are either valid or invalid with respect to behavioral subtyping.

Consider the example shown in Figure 2.21. A client performing a call x.bar(6), where x has static type X, fulfills the precondition of X.bar and may assume, from the postcondition of X.bar, that the returned result will at least be 6. If, at runtime, x contains an instance of SubX1 instead, the (now weaker) precondition of the actual receiver is still fulfilled, and the (stronger) postcondition of the actual receiver still fulfills the assumption the client makes about the return value, making this a valid override.

On the other hand, with a stronger precondition in SubX2, the caller may unknowingly violate the (stronger) precondition of the actual receiver, and with instances of SubX3, its assumption about the return value may not be fulfilled by the (weaker) postcondition of the actual receiver, making both overrides invalid.

```
class X:
  def bar(self, i: int) -> int:
    # requires P
    # ensures Q
    ...

class SubX(X):
  def bar(self, i: int) -> int:
    # requires P'
    # ensures Q'
    ...
```

```
method SubX_bar_override_check(self:
        Ref, i: Ref) returns (res: Ref)
  requires issubtype(typeof(self), SubX)
  requires issubtype(typeof(i), int)
  requires P
  ensures Q
{
  res := SubX_bar(self, i)
}
```

Figure 2.22.: Override in Python (left) and generated behavioral subtyping check (right). P, P', Q, and Q' represent arbitrary assertions. For the call to SubX_bar, the verifier will check that its precondition P' is fulfilled, and that its postcondition Q' is sufficient to establish Q.

We check behavioral subtyping in the verifier by encoding a check that a call to an overriding method fulfills the specification of the overridden method. Note that, as an alternative, it would be possible to guarantee behavioral subtyping by using *specification inheritance* [59], which combines a method's declared specification with the specifications of the methods

[59]: Dhara et al. (1996), 'Forcing Behavioral Subtyping through Specification Inheritance'

it overrides into an *effective specification* which, by construction, fulfills behavioral subtyping; the method's implementation is then verified against this effective specification instead of the declared one. We choose not to do this because we believe that, in practice, it makes verification more difficult for programmers, who, while specifying a program and debugging proofs, essentially have to mentally compute these effective specifications to know what the current proof goal is. On the contrary, checking behavioral subtyping in a verifier is simple and usually requires no additional effort from programmers.

For the general case shown in Figure 2.22 on the left, we generate the behavioral subtyping check shown on the right. This method verifies only if the superclass precondition P is sufficient to establish (i.e., stronger than) the subclass precondition, and the subclass postcondition is sufficient to establish (i.e., stronger than) the supertype postcondition.

Once this check is in place, calls to dynamically-bound methods can now *always* be assumed to fulfill the specification of the statically-referenced method, and can be verified using that specification. This kind of encoding is preferable to explicitly encoding such calls as explicit case splits on the receiver type and calls to all possible overrides, since such an encoding requires knowledge of all possible overrides and is therefore not modular.

```python
class X:
  def __init__(self, i: int) -> None:
    # ensures acc(self.f) && self.f >= i
    self.f = i

  @classmethod
  def construct(cls) -> X:
    # ensures acc(result.f) && result.f >= 2
    return cls(2)

class SubX1(X):
  def __init__(self, i: int) -> None:
    # ensures acc(self.f) && self.f == i
    self.f = i

class SubX2(X):
  def __init__(self, i: int) -> None:
    # ensures acc(self.f)
    self.f = 0

x = X.construct()
x_prime = X().construct()
x1 = SubX1.construct()
x2 = SubX2.construct()
```

Figure 2.23.: Python code using class methods.

One peculiarity of Python is that sometimes, behavioral subtyping is required even for constructors, namely, when a class has a so-called *class method*, as in the example in Figure 2.23. Class methods can be called on classes or on class instances; crucially, both on (instances of) the class they are defined in, and on (instances of) its subclasses. When invoked, a class method implicitly gets the class object of its receiver as the first argument. This class object (i.e., its constructor) can be called, resulting in the creation of a new instance.

However, as a result, the verifier has to ensure that a class method in a superclass also executes correctly and fulfills its specification when called on a subclass, meaning that the constructor signature of the subclass must be compatible with that of the superclass (this is checked by the type system) and that the specification of the constructor of the subclass is compatible with that of the superclass.

In the example in Figure 2.23, the constructor of SubX1 has a specification that is compatible with that of X (they have the same trivial precondition and the subclass constructor has a stronger postcondition); however, the constructor of SubX2 does not, since it does not ensure that the final value of field f is at least the integer given as an argument. As a result, if SubX2 were admitted, construct would violate its postcondition when cls is SubX2, and variable x2 at the end would have a field with a lower value than promised by construct. To prevent cases like this, we perform behavioral subtyping checks as shown below not just for ordinary methods, but *also* for class constructors, if the class has some superclass that has at least one class method (otherwise the check is not necessary).

2.2.2.4. Predicate Families: Design

In the context of permission logics, behavioral subtyping is not fulfilled by many real programs with a naive treatment of specifications: The requirement that preconditions of overriding methods must not be stronger than those of the overridden methods is problematic if those preconditions contain the permissions to read or modify heap locations, since subclasses typically have *more* fields than superclasses. The example in Figure 2.24 illustrates the problem.

```
class Socket:
    def __init__(self, ip: str, ...) -> None:
        # ensures acc(self.ip) && ...
        # set up state
        self.ip = ip
        ...

    def send(self, msg: Message) -> None
        # requires acc(self.ip) && ...
        # ensures acc(self.ip) && ...
        ...

class BufferedSocket(Socket):
    def __init__(self, ip: str, ...) -> None:
        # ensures acc(self.ip) && ... && acc(self.send_buffer)
        super().__init__(self, ip, ...)
        self.send_buffer = []
        ...

    def send(self, msg: Message) -> None:
        # requires acc(self.ip) && ... && acc(self.send_buffer)
        # ensures acc(self.ip) && ... && acc(self.send_buffer)
        if len(self.send_buffer) + 1 >= BUFFER_LEN:
            super().send(self.concat(self.send_buffer, msg))
            self.send_buffer = []
        else:
            self.send_buffer.append(msg)
```

Figure 2.24.: Example demonstrating the problem of requiring behavioral subtyping in a permission logic: Overriding methods require more permissions than the methods they override, making their preconditions stronger.

In the example, we have a socket class representing a TCP/IP socket which offers a send operation. The constructor sets up the state of the socket, which will include some number of fields that store, for example, the target IP address. The subclass, BufferedSocket, reuses the sending functionality from its superclass, but sends messages in batches to improve efficiency. As a result, it needs an additional field to store the buffer of messages that have not been sent yet. The send method of this subclass now needs access to all the fields that the superclass method needs to access, and it *additionally* needs to modify the buffer field. In a permission logic, that means that its precondition contains all permissions required by the superclass method and the permission to the buffer field, making it *stronger*, which is not allowed by ordinary behavioral subtyping.

However, cases like this one are clearly very common, and conceptually (abstracting from the view of the permission logic), the subclass *does* fulfill behavioral subtyping: In both classes, the send method requires access to all class fields that are created by the respective constructor; there are more such fields in the subclass, but the constructor also creates more fields, so no dynamically-bound calls can actually ever run into problems.

The established solution to this problem is the use of *predicate families* [171], that is, abstract predicates that can be re-defined for every subclass in a class hierarchy. In this case, we can declare an abstract notion of having a socket that is initialized and running, and define what this means for both classes. For an ordinary socket, this predicate would contain the permissions (and possible value constraints) established by its constructor; for the subclass, it *additionally* contains the permission to the buffer-field (and possibly a constraint about the length of the contained list). Then we can use this predicate in the specifications of the respective constructors and send methods, resulting in the code in Figure 2.25.

[171]: Parkinson et al. (2005), 'Separation Logic and Abstraction'

```
class Socket:
  def __init__(self, ip: str, ...) −> None:
    # ensures self.running()
    # set up state
    self.ip = ip
    ...

  def send(self, msg: Message) −> None
    # requires self.running()
    # ensures self.running()
    ...

  predicate running(self):
    acc(self.ip) ∗ ...

class BufferedSocket(Socket):
  def __init__(self, ip: str, ...) −> None:
    # ensures self.running()
    super().__init__(self, ip, ...)
    self.send_buffer = []
    ...

  def send(self, msg: Message) −> None:
    # requires self.running()
    # ensures self.running()
    if len(self.send_buffer) + 1 >= BUFFER_LEN:
      super().send(self.concat(self.send_buffer, msg))
      self.send_buffer = []
    else:
      self.send_buffer.append(msg)

  predicate running(self):
    acc(self.ip) ∗ ... ∗ acc(self.send_buffer)
```

Figure 2.25.: Example from Figure 2.24 using a predicate family running in the specification.

Now, the contracts for the send methods of both socket classes are identical (on the abstract level where the different predicate definitions are ignored), so that behavioral subtyping is trivially fulfilled.

In practice, some notion of predicate families is almost always required to allow verifying real code in languages with subclassing and behavioral subtyping requirements. We will now explain the specific design and implementation we have chosen for Python.

One central design choice is whether subclasses can *completely* re-define the meaning of a predicate for itself, which is the case, for example, in VeriFast [110], or if they can only *add* constraints (and permissions), which is the case in Spec# [130]. In the example in Figure 2.25, the notation we used suggests that running is completely re-defined by the subclass, but actually, it would have been sufficient to state that the subclass *adds* the permission to self.send_buffer to the predicate. The main advantage of the former option is obviously flexibility; the advantage of the latter is that predicates can be unfolded without exact type knowledge.

[110]: Jacobs et al. (2011), 'VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java'

[130]: Leino et al. (2008), 'Using the Spec# Language, Methodology, and Tools to Write Bug-Free Programs'

```
def print_socket_info(s: Socket) -> None:
  # requires s.running()
  # ensures s.running()
  unfold s.running()
  print(s.ip)
  ...
  fold s.running()
```

Figure 2.26.: Example which can be verified if predicate families are defined in an additive way, but not if their definitions for different types are completely independent.

As an example, consider the piece of code in Figure 2.26. If predicate families are defined in an additive way, then this method that takes a Socket object (but does not know if it gets an instance of the exact Socket class, the BufferedSocket class, or some other subclass) and requires the running predicate can still unfold the predicate and know that it gets *at least* the permissions and constraints in the version of running defined in the Socket class, but *may* get additional constraints and permissions added by subclasses. In this case, just the permissions to access the ordinary Socket fields are sufficient for the implementation, and this method can be verified with additive predicate families. With completely independent predicate family members, the running predicate *cannot* be unfolded in this setting (or one could get no information from such an unfolding, since one may have an instance of the predicate of some currently unknown subclass that contains no permissions at all). While this problem can be worked around when getter-functions are used to access object information (the unfolding can then happen *inside* the dynamically-bound getter functions, where more precise type information about the receiver is available), typical Python code often reads and modifies fields directly outside their classes, and does not use getter and setter functions as often as, for example, Java code. We therefore use an additive definition of predicate families, and (unlike in Figure 2.25) subclasses only have to state the added permissions and constraints when re-defining predicates; the permissions and constraint from superclasses are implicitly added.

Predicate families do not come for free; their introduction creates two new potential sources of unsoundness and incompleteness. We will explain these cases (and our approach for preventing them) now.

The example in Figure 2.27 consists of a Cell class that contains a single integer field, for which it has a setter, as well as another integer field that is only set in the constructor; its subclass adds another field that shadows the first field, and overrides the setter accordingly. The inv predicate

```
class Cell:
  def __init__(self, i: int) −> None:
    # ensures self.inv(i)
    self.f = 0
    self.i = i
    fold self.inv(i)

  def set(self, i: int) −> None:
    # requires self.inv(_)
    # ensures self.inv(i)
    unfold self.inv(_)
    self.i = i
    fold self.inv(i)

  predicate inv(self, i: int):
    acc(self.f) ∗ acc(self.i) ∗ self.i == i

class SubCell(Cell):
  def __init__(self, i: int) −> None:
    # ensures self.inv(i)
    self.f = 0
    self.i = i
    self.i2 = i
    fold self.inv(i)

  def set(self, i: int) −> None:
    # requires self.inv(_)
    # ensures self.inv(i)
    unfold self.inv(_)
    self.i = i
    self.i2 = i
    fold self.inv(i)

  predicate inv(self, i: int):
    acc(self.i2) ∗ self.i2 == i ∗ self.f >= 0

def outside_set_1(x: Cell, i: int) −> None
  # requires x.inv(_)
  # ensures x.inv(i)
  unfold x.inv(_)
  self.i = i
  fold x.inv(i)

def outside_set_2(x: Cell) −> None
  # requires x.inv(?i)
  # ensures x.inv(i)
  unfold x.inv(_)
  self.f = −2
  fold x.inv(i)
```

Figure 2.27.: Example illustrating potential soundness and modularity issues of verification using predicate families. Preconditions using placeholders, like x.inv(_), represent instances of the predicate with arbitrary arguments; the notation **requires** x.inv(?i) also represents an instance with an arbitrary argument, and binds that argument to variable i.

family contains the permissions for all fields in the class, and constrains the fields. Two methods defined outside any of the classes attempt to also implement the functionality of setters for each of the fields in the original class. This code showcases three potential problems:

1. Notice that the **set** method in both classes does exactly what it needs to satisfy its postcondition **self**.inv(i); in the superclass, that means modifying one field, in the subclass, two fields. Both setters are correct. Imagine, however, a scenario where SubCell does *not* override the setter, and therefore simply inherits it: The resulting code would be incorrect, since **set**, when called on an instance of SubCell, *must* also modify field i2 to satisfy its postcondition with its updated predicate definition. In other words, folding the **self**.inv(i) predicate at the end of the setter should be possible if the receiver has exactly type Cell, but not if the setter is inherited and **self** has type

SubCell. However, clearly, it would be wrong to reject the definition of set in Cell, since it does exactly what is required of it in that class; the solution is to prevent *inheriting* this method in SubCell.
In general, any method that folds a predicate family member needs to be re-verified when it is inherited in a class that adds to the predicate definition, since the inherited code may not fulfill the added constraints.

2. There is a similar problem in outside_set_1: Its implementation is correct for instances of Cell but not SubCell because it does not modify the i2 field, which would be required to re-establish the predicate self.inv(i). However, this method is defined outside both classes and is not inherited, and it may be defined and verified in a context where SubCell is not even known. However, adding class SubCell should not change the correctness of outside_set_2, which does not depend on it; that would be non-modular. Therefore, this method must be rejected even when SubCell is not known: Its body simply does not fulfill its specification for all possible subclasses of Cell and their definitions of inv; it only does so for instances of Cell itself.
   In general, it should not be possible to create a *new* predicate (here, self.inv(i) for a new i, instead of the previous self.inv(_) with some potentially different argument) without knowing the exact type of the receiver, since this may always require establishing conditions that are currently unknown.
3. outside_set_2 is similar again: it is correct for instances of Cell but not SubCell, because its modification of the f field is forbidden by the added conjunct self.f >= 0 in SubCell's definition of inv. The same modularity concerns apply as for the previous method, so we must either forbid outside_set_2 or the definition of inv. In this case, we opt for the latter, and forbid subclass predicate additions from constraining heap locations without also adding some amount of permission to those same locations themselves; in other words, each subclass' addition to a predicate family must be *self-framing*. In this case, this requires either splitting the permission to self.f into two parts and adding one part only in the predicate for SubCell, in which case outside_set_2 no longer gets a full permission without knowing its argument has type SubCell, and therefore cannot modify the field. Or class SubCell is simply forbidden from adding the constraint.
   In general, self-framing predicate additions ensure that outside code can *maintain* predicate instances (in this case, self.inv(i) for some specific i) when making modifications without knowing the precise type of the receiver: Since all additional potentially unknown constraints frame themselves, they cannot be violated by any legal change that is allowed with only supertype permissions.

2.2.2.5. Predicate Family Encoding

We encode all members of a family that is defined on the Python level (where we interpret each predicate defined as part of a class as a family) into a *single* predicate on the IVL level. That is, the definition of the predicate family pred in Figure 2.28 on the left is encoded as the predicate on the right in the IVL, where pred_unknown is an abstract predicate, whose purpose we will explain soon.

```
class X:
  def foo(self, ...):
    ...

  def bar(self, ...):
    ...

  predicate pred(self, arg: Y):
    P

class SubX1(X):
  # foo is inherited

  def bar(self, ...):  # override
    ...

  predicate pred(self, arg: Y):
    P1

class SubX2(X):
  predicate pred(self, arg: Y):
    P2

class SubSubX1(SubX1):
  predicate pred(self, arg: Y):
    P1'
```

```
predicate pred(self: Ref, arg: Ref) {
  issubtype(typeof(self), X) * issubtype(typeof(arg), Y) *
  (
   issubtype(typeof(self), X) ==>
     P
  ) *
  (
   issubtype(typeof(self), SubX1) ==>
     P1
  ) *
  (
   issubtype(typeof(self), SubX2) ==>
     P2
  ) *
  (
   issubtype(typeof(self), SubSubX1) ==>
     P1'
  ) *
  (
   (
     typeof(self) != X * typeof(self) != SubX1 *
     typeof(self) != SubX2 * typeof(self) != SubSubX1
   ) ==>
   pred_unknown(self, arg)
}
```

Figure 2.28.: Generic predicate family definition in Python (left) and its IVL-encoding (right).

Unfolding the predicate family on the Python level then simply corresponds to unfolding this single predicate on the IVL level, and the verifier will gain information and permissions from the predicate according to its knowledge of the type of the receiver. For example, when unfolding the above predicate, if the verifier knows that under certain conditions the receiver is definitely a subtype of SubX1, it can conclude that it now has the permissions from (and knows the information contained in) P and P1, but does not know for certain that it also has P1'.

The definition of this predicate is obviously not modular, since its definition will be extended when a new class is known, and thus, code that unfolds the predicate will get more information after an unfold, and have to prove more to be able to fold a predicate. However, we have carefully designed our encoding in such a way that the additional knowledge, or the additional proof obligations resulting from an extended predicate, *never* change the verification result for existing code. That is, while the encoding itself is technically non-modular, the verification process still is. We will first explain the proof obligations we generate (based on the Python code example in Figure 2.28 (left)), and subsequently justify why these proof obligations are sound and modular in the sense just described.

```
method X_foo(self: Ref, ...)
  requires ...
  ensures ...
{
  assume typeof(self) == X
  ...
}
```

```
method SubX1_foo_inherit_check(self: Ref, ...)
  requires ...
  ensures ...
{
  assume typeof(self) == SubX1
  ...
}
```

Figure 2.29.: Verification of methods using precise type information (left), and re-verification of inherited methods using new type information (right).

First, methods defined in a class are verified under the assumption that the type of the receiver is *exactly* the type of the class the method is defined in, as shown in Figure 2.29 (left). Second, methods that are *inherited* are re-verified, now with the knowledge that the receiver type is *exactly* the type of the class that inherits the method, as shown in Figure 2.29 (right). Essentially, we treat inherited methods as if they were re-defined in the class that inherits them. Third, method calls on a super()-receiver (that is, statically-bound calls to a superclass method) are *inlined* at the call site.

Fourth, as mentioned above, when methods are overridden, the overriding method is verified like any method definition, and the behavioral subtyping check is generated as shown before in Figure 2.22 (right).

As a result of these four principles, all methods are now verified w.r.t. the precise type of each possible receiver they can ever be called on: All cases are checked with the most precise information available about the receiver types, making verification both sound and as complete as possible with respect to receiver types.

Regarding the modularity of handling predicate families whose receiver is *not* the current self-object, we make two choices: First, we ensure that no new predicate can ever be folded without knowing the precise type of the receiver (preventing the issue shown above in outside_set_1), by using an abstract pred_unknown shown in Figure 2.28 (right). If the verifier cannot prove that the receiver of the predicate to be folded is a direct instance of a known class, it must prove that the method has an instance of this predicate, which it can never do. When the exact type is known, folding the new predicate family instance is sound, because it is exactly known which conditions must be fulfilled. Note, however, that it would *not* be sound to just replace the abstract predicate with the expression false: This would have the same effect when folding a predicate family, but when unfolding one, one would gain the knowledge that the receiver has a known type, which in general is not desired and would be unsound.

Second, we ensure that modifications of the contents of a predicate cannot break constraints defined in currently unknown subclasses (the issue shown above in outside_set_2) by encoding a proof obligation that all family additions (i.e., in this case, P, P1, P2, and P1' are individually self-framing. Since all such potentially existing but currently unknown constraints frame themselves, it is ensured that they cannot be broken by any modification.

2.2.3. Global Statements and Declarations

As part of its dynamic nature, Python handles declarations (for example of classes and methods) differently from most statically-typed languages: In Python, declarations of such constructs are ordinary statements which, when executed, evaluate the definition of a class or method, and bind it to a name in the current namespace. That is, these declarations are part of a global sequential program; they are executed at a point in time and in a given state, and they can succeed or fail in that state. Moreover, classes and functions can in principle be re-defined, and declarations of all kinds can be mixed with any other statements, including statements performing actual computations, as part of the global top-level program. Similarly, module imports are also statements that are part of this program; their effects depend on the current state, and importing modules not only adds declarations to the scope, but also executes other top-level statements in the imported module.

For verification, this poses the following challenges:

- To support real code, it must be possible to not only verify code written in methods, but also code that is written as top-level statements in a module.
- Verifying the top-level code requires proving that it only (transitively) invokes methods and (transitively) refers to classes that are already declared in the current state.
- Since declarations of methods and classes are also statements that can refer to other declarations (e.g. a class declaration can refer to another class as its superclass), it is also required to prove that declarations will successfully execute in the current state.
- All previously mentioned points require modeling which global names are defined (in different name spaces) in the current state.
- They also require modeling the effects of imports statements, which cause the execution of all top-level statements in the imported module; this is challenging because Python's module import mechanism is inherently non-modular.

In this section, we will explain our approach to addressing these challenges.

2.2.3.1. Top-Level Computation

The behavior of top-level statements in Python is inherently non-modular. The piece of code in Figure 2.30 illustrates this: In this example, executing module A has a different outcome than executing module B.

```python
a = 2
import B
assert a == 7
a += 3
```

```python
from A import a
assert a == 2
a += 5
```

Figure 2.30.: Code consisting of two modules, module A on the left and module B on the right.

If module A is executed, it will assign the value 2 to variable a, then it will load module B, which will in turn execute its top-level statements. B's first statement, however, says to import module A. Since A is already in the process of being executed, this statement will not re-execute the statements in module A, but will only import the (already existing) variable a into module B. Since a's current value is 2, the subsequent assertion will succeed, and the next statement will add 5 to a's value, setting it to 7. Then, the execution of A's top level statements continues after the import; its assertion will also succeed, and it will then add 3 to a.

If module B is executed, the behavior is different. B's first statement says to import module A, which, as before, leads to the execution of the top-level statements in A. The first statement sets the value of a to 2. The subsequent import of B does nothing, since B is already being executed, and its namespace is currently empty, so nothing can be imported. Then, the assertion in A is executed, which now fails, since a's value is still 2, and not 7, as the assertion demands.

This example illustrates that the order in which statements are executed is different depending on the starting point, and as a result, modules can be correct when executed from one starting point, but incorrect when executed from another. Since *any* module is, in principle, a valid starting point for a program execution, the top-level statements have to be verified separately for any such starting point.

Thus, by default, when tasked to verify a given Python module, we verify the top level statements as if they were executed from that starting point. For this purpose, we create a special method in the IVL encoding that contains the top-level statements. For most statement types, the encoding from Python to the IVL is identical no matter if the statement is located in a method or on the top-level of a module; however, there are differences, which we will discuss now.

2.2.3.2. Global Variables

Global variables differ from local variables, since they can be read and modified by arbitrary methods.

For verification, we differentiate between two types of global variables: *mutable variables* and *constants*. While Python has no built-in concept of a constant, obviously the concept of a constant is still useful for many programs in practice, and they will simply contain top-level global variables that are set once and never modified. We heuristically designate any global variable that has a name written in capital letters, and has a primitive type, to be a constant, and enforce that it is only written to once. All other global variables are treated as mutable variables.

The main distinction between the two types of global variables is that methods will require permissions to access (read or modify) mutable variables, whereas any method can read (but never modify) any constant in the program. On the IVL level, we encode constants as global functions that return the value of the constant; mutable variables are encoded as functions that return a *reference* that has a *value* field, which contains the actual value of the variable.

```
1  MY_CONSTANT = 34
2
3  my_var = 42
4
5  def modify():
6    # requires acc(my_var)
7    # ensures acc(my_var)
8    global my_var
9    my_var += MY_CONSTANT
```

Figure 2.31.: Program using a global constant (in line 1) and a mutable global variable (line 3); to access the latter, methods require a permission.

As with local variables, we have to ensure that global variables are defined when they are read. For mutable variables, this problem can be solved by requiring the respective permission (the same solution as for fields), as shown in Figure 2.31. For constants, we have to ensure that they are already defined at the point where they are used. We use the same mechanism for this as we use for any other declarations, which we will describe in the following subsection.

2.2.3.3. Declarations

As explained above, top-level statements can refer to declarations that come only later in the program. Figure 2.32 (left) shows a simple example where a class is instantiated before it is declared. When executed, a program like this would abort with a NameError.

```
a = A()

class A:
  pass
```

```
class B(A):
  pass

class A:
  pass
```

Figure 2.32.: Example code that refers to a class before it is defined.

Similarly, however, *other declarations* can refer to classes or methods that are not yet defined. The example in Figure 2.32 (right) shows a class B that tries to subclass A before A is declared. Like the first program, this would result in an error when executed.

```
def foo(a: A):
  ...

class A:
  pass
```

```
def foo(...):
  bar()

foo(...)

def bar(...):
  ...
```

Figure 2.33.: A method whose *definition* requires a class to be defined (left), and one whose *execution* (but not definition) requires another method to be defined.

A similar error will be produced by the program in Figure 2.33 (left), where method foo annotated its parameter a with type A before that type is defined. For methods, however, it is important to make a distinction between their *definition* and their *execution*. For example, the program in Figure 2.33 (right) contains a method foo whose body will execute method bar. bar is defined after foo, but this by itself is no problem: When foo is defined, only its signature (i.e., type annotations on its parameters and its return value, as well as all expressions that are used as default values for its parameters) is evaluated, meaning that only the dependencies of its signature must already be defined. In this example, the definition of foo would be executed without an error.

However, after that, foo is *executed* at a point when bar has not yet been defined. As a result, the *execution* of foo will fail at this point. If the top-level call to foo was moved after the definition of bar, this program would execute without an error.

To verify that the execution of a program does not result in an attempt to reference a class, method or variable that has not yet been defined, we perform the following steps:

- As for local variables, we explicitly track the set of global names that are currently defined. Unlike for local variables, on the global level, we track this information for *any* kind of name that could be referenced by other code, i.e., for variables, methods, and classes. For this purpose, we use, for every module, a set module_names of integers that represent the names currently defined in that module. Whenever a name n is defined, we generate the statement module_names := module_names union Set(single(<n>)), which adds said name to the set, where <n> represents the integer-encoding of name n, as before. The function single maps integers to the uninterpreted type Name; we will explain the purpose of this function when discussing imports.

- We compute the set of (transitive) dependencies of each top-level statements. For methods, we distinguish between the dependencies of the definition and the dependencies of the execution, as explained before. Where necessary, we overapproximate, so that this computation can be performed statically on the AST without the help of an SMT solver.
- As part of the encoding of any global statement, we encode a proof obligation stating that the dependencies must already be defined. For example, if a statement refers to a name n, we add the assertion assert <n> in module_names.

2.2.3.4. Imports

The final part of the encoding is the encoding of imports.

First, as we explained above, an import statement executes the top-level code of an imported module, *unless* said module has been executed before. We model this by adding a boolean flag module_defined for every module in the program that represents whether the module has been imported yet. When encountering an import statement for a given module with global statement S, we then generate the code shown in Figure 2.34 on the IVL level.

```
if (!module_defined) {
  module_defined := true
  // encoding of S
  ...
}
```

Figure 2.34.: Encoding of an import of a module whose body contains the top-level statement S.

Second, an import adds names to the current modules namespace. Those names will either be ordinary names, like in the case from A import a in the example above, which only adds the name a to the current module. Or they will be combined names consisting of the name of the imported module and the names of the imported elements in the module. As an example, a statement import A in a state where module A contains a name a would instead make the name A.a available in the current module, as well as all other names currently defined in A, concatenated to the name of the module.

We also support this fact in our modeling of available names in a module: As shown before, we use a function call single(<n>) to represent the simple name n; now, we use a function call combine(single(<n1>), single(<n2>)) to represent the combined name n1.n2.

When updating a module named n1, in the encoding, we declare a new set of names new_names that we constrain to contain all names that *combine* the name of the module and a name currently in the imported module. We then add the names in this new set to the set of names in the current module, as shown in Figure 2.35.

Figure 2.35.: Import encoding that adds imported names to the set of currently known names, where module_names_2 is the set of names that exist in the imported module.

```
var new_names : Set[Name]
assume forall name: Name :: { (combine(single(<n1>), name)) in new_names }
   (name in module_names_2) == (combine(single(<n1>), name)) in new_names)
module_names := module_names union new_names
```

2.2.4. Limitations: Dynamic Object Model and Runtime Code Generation

Python has some dynamic aspects that we have so far glossed over, which we choose not to support at all.

The first one is unsurprising: Python allows generating code at runtime (using the eval method and others); this is not supported by any existing static verifiers, and generally excluded. Instead of supporting this case, we prove that no such functions are called in verified code.

The second central limitation is that we choose not to model Python's object model and internal calling procedures precisely. As an example, we have so far treated a field read e.f as if it was an atomic operation that (like in Java or C++) directly returns the value of a fixed location in memory, namely, a location that is part of the object e (or, if it is a static field, its class). However, in Python, such a field read conceptually involves several method invocations that can perform arbitrary computations and modify the state. The simple Java-like field read semantics is simply the high-level outcome of the *default* behavior of such methods.

More precisely, a field read (or, more precisely, an attribute lookup) e.f in Python has the following steps (simplified because there are a number of special cases):

1. e is evaluated to some value v
2. __getattribute__ is called on v, with the argument "f".
   The default implementation of __getattribute__ in the object class will first look for the key "f" in the object's own attribute dictionary (which can be accessed from outside as v.__dict__) and, if that fails, in the object's class' attribute dictionary. If it finds an entry, it will return it, otherwise, it will raise an AttributeError.
   However, __getattribute__ can be *overridden* in classes, leading to arbitrary different behavior.
3. If the call to __getattribute__ resulted in an AttributeError, the __getattr__ method is invoked on the object, again with argument "f". This method is not implemented by default, and is intended to be defined by programmers to define *computable* attributes dynamically.
4. If result of step one is a *descriptor* object, that is, one that implements a __get__ method, and if it was retrieved from the __dict__ of the object's class, then the __get__ method is called and the result is returned; otherwise, the result of the previous steps is returned directly.

This process (which, as mentioned, is still a simplified version of the actual process) is obviously quite complex for one of the most common and basic operations that can be performed in the language. In addition, since it can be customized in different ways by defining or re-defining __getattribute__ or __getattr__, it basically allows programmers to completely re-implement the behavior of a field write or read.

Attribute lookups are not the only basic language features that can be completely re-defined by programmers: Other so-called "magic" methods are used to implement other basic language operations (such as the `__new__` method which usually handles object creation), and classes themselves are instances of *metaclasses* that define these basic operations. Programmers can define custom metaclasses that redefine the behavior of many such basic language operations, and classes that are instances of these custom metaclasses will subsequently behave accordingly.

During verification, if all language operations are modeled exactly, one needs precise information about the behavior of all such methods for any given object used by the code; otherwise it will be impossible to prove even basic code properties. Note that this is a case where behavioral subtyping does not help in practice: One could give a specification to the standard implementations of all such magic methods, and then demand that all subclasses (or submetaclasses) that override them also stick to these specifications. However, in practice, the specifications of the standard implementations would either precisely express the standard behavior, in which case overrides would basically be forbidden from making any meaningful changes and the additional flexibility would be useless. Or the standard specifications would be relatively weak, in which case overrides would have more freedom to implement custom behavior, but in cases where only the (weak) standard specifications are known, verification would be virtually impossible because of the lack of information about many basic language operations. Note that all these decisions also affect specifications, since permissions are usually specified per field, so if the mechanisms that define how fields and field lookups work are more dynamic, permission specifications would have to be adapted accordingly.

While we do not claim that it would be impossible to define a specification and verification mechanism that models, for example, attribute lookups precisely and is precise enough to verify real code, we do believe that this would come at the cost of significantly more complex specifications in many standard cases, and would almost certainly also affect verification performance due to the need to model many additional layers of indirection. Since our goal is to be able to verify *user* code, and features like metaclasses that redefine basic object functions are typically only used in libraries, we therefore choose not to model these features precisely, and instead model the simpler default behavior *directly*. In addition, we ensure that verified code does not re-define any of these basic features, by checking (syntactically) that metaclasses are not used and magic methods responsible for basic object operations are not overridden.

We do, however, allow defining or overriding *some* magic methods that do not modify fundamental language aspects: For example, we allow overloading of arithmetic operators (which can be done, for example, by implementing methods like `__add__` and `__mul__`, which implement the addition and multiplication operators, respectively), and we allow defining length and boolean properties of objects by implementing the `__len__` and `__bool__` functions; the former is called on an object if an object is passed to the built-in `len` function, and the latter is called when a non-boolean object is used where a boolean is expected, for example, as the condition in a loop.

In addition, we build support for *some* parts of the standard library that internally *use* language features like the complex attribute lookup directly into the verifier. As an example, the property decorator can be used to annotate getter and setter functions, and makes them accessible through field read and write operations (the implementation of this decorator internally makes use of descriptors, mentioned above). An example is shown in Figure 2.36.

```
class MyClass:
  def __init__(self):
    self._field = 0

  @property
  def field(self) -> int:
    print("getter called")
    return self._field

  @field.setter
  def field(self, val: int) -> None:
    print("setter called")
    if val >= 0:
      self._field = val

myclass = MyClass()
myclass.field  # prints "getter called", returns 0
myclass.field = 5  # prints "setter called"
myclass.field  # prints "getter called", returns 5
myclass.field = -5  # prints "setter called"
myclass.field  # prints "getter called", returns 5
```

Figure 2.36.: Example of code that uses the property decorator.

We directly implement a treatment of such property functions in the verifier that does not model the internals involving descriptor objects, but treats a read from the field field like a call to the field method in the class, and handles writes to the field analogously.

2.2.5. Other Complex Language Features

Python is a comparatively rich and complex language and offers some other language features that offer interesting verification challenges (some of which are unrelated to its dynamic nature). While we will not go in detail about these features, we will briefly list some interesting aspects here.

2.2.5.1. Complex Assignments

Python allows for complex assignments, whose left hand side can contain, for example, nested tuples and starred expressions. As an example, the statement d, (g, h), *e, z = l, where l is a list, will assign the first element of l to variable d and the *last* element of l to z; it will take the second element of l, ensure that it is a container (e.g. a tuple or a list) containing exactly two elements, and assign those to variables g and h. Any remaining elements of l will be assigned as a tuple to variable e. If l does not have at least three elements, or if its second element is not a container containing exactly two elements, the assignment will fail.

A verifier has to model this behavior precisely and prove that all conditions under which the assignment can fail will not occur at runtime.

Moreover, assignments of this form can not only occur as individual statements, but several other statements also have assignment-like behavior. For example, for loops, which assign the element of the current iteration to a loop variable, can also have an expression containing nested tuples and starred expressions like the one in our example instead of a single variable.

In most cases, correctly modeling this kind of language feature is not difficult in principle, but the number of statements in the language and their subtlety still make it non-trivial to model them correctly in all cases. We support arbitrarily complex assignments both as top-level statements and wherever they occur in other supported parts of the Python language by applying a general encoding for assignments that models their behavior precisely in all such places.

2.2.5.2. Default Arguments

Python methods can have default arguments, as shown in line 15 of Figure 2.17. In Python, these arguments can contain arbitrary expressions, and those expressions are evaluated *once* when a method is *defined*. That is, those expressions must be shown to evaluate without raising an error when verifying the global top-level code (see the previous subsection), and later invocations of methods with default arguments must take into account that their state may have been modified in the meantime. We solve this problem by treating default values as if they were explicitly assigned to global variables.

2.2.5.3. Exception Handling

Like many other object-oriented languages, Python offers a mechanism for raising exceptions, and for catching them using try–except–finally-blocks. While not fundamentally different than the equivalent in, for example, Java, verifying code that uses exception handling is still difficult. Figure 2.37 illustrates the challenge.

```
 1  def exception_example(x: bool) -> int:
 2    try:
 3      try:
 4        other_method()
 5        if x:
 6          return 1
 7      except MyException:
 8        ...
 9      finally:
10        ...
11      ...
12    except MyOtherException:
13      ...
14    finally:
15      ...
16    return 0
```

Figure 2.37.: Example demonstrating complex control flow as a result of try-except-finally-blocks.

Nested try–except–finally blocks give rise to a large number of possible paths through the method. As an example, if the call to other_method returns with an exception of type MyException, the code will jump to the handler in line 8, then to the finally-block in line 10, then continue execution in line 11, and (barring other exceptions being raised) execute the *other* finally-block in line 15, and then return in line 16. If, on the other hand, the raised exception is of type MyOtherException, execution will directly jump to line 10, then skip line 11 and jump to the handler in line 13, subsequently go to the finally-block in line 15 and return in line 16 as before. If no exception is raised and the return-statement in line 6 is executed, the code will execute both finally-blocks but skip everything else; if no exception is thrown and x is false, then both finally-blocks are executed, but with line 11 inbetween and line 16 afterwards. Similarly complex behavior can be seen when using break statements in loops inside (potentially nested) try-blocks.

Modeling these options correctly, in particular, the fact that at the end of a finally-block the execution can jump to many different places depending on whether there is still an unhandled exception or the execution is returning, is challenging in practice. We solve it by storing and maintaining data that marks the kind of exit out of a try-block that is currently performed, and branching depending on this data at the end of a finally-block. We will not describe our encoding in detail here, but it has been reverse-engineered from our implementation and described by Rubbens [184] in his Master's thesis, where he also compares it to other possible encodings.

[184]: Rubbens (2020), 'Improving Support for Java Exceptions and Inheritance in VerCors'

2.2.5.4. Concurrency

Finally, since we aim to support concurrent Python programs, we have to correctly model the forking and joining of threads, and thread interactions through locks. As stated before, the permission system we use ensures that verified code does not contain any data races. In such a system, thread interactions can be modeled as follows [132]:

[132]: Leino et al. (2009), 'A Basis for Verifying Multi-threaded Programs'

- When forking a new thread that is to execute some method m, the forking thread has to *exhale* m's precondition (conceptually giving the permissions in said precondition to the new thread).
- Conversely, when joining a thread that has executed some method m, the joining thread may *inhale* m's postcondition. However, to prevent duplicating permissions in the postcondition, if a thread is joined multiple times, either only one join must result in an inhale of the postcondition, or each join must give the joining thread only a fraction of the permissions in the postcondition.
- Locks are associated with *lock invariants* that contain permissions to the heap locations protected by the lock, as well as conditions that must always be fulfilled when the lock is not held by any thread. When acquiring a lock, the lock invariant may be *inhaled*.
- Conversely, when releasing a lock (and when sharing it for the first time), the lock invariant must be *exhaled*.

We implement this model and add specifications and proof obligations specific to Python's design of threads and locks (both of which are objects in Python). For example, we need to ensure that no thread object is started twice (which would lead to a runtime error) and do this by creating a permission MayStart which comes into existence when a thread-object is created, and is consumed when it is actually forked (which happens when the start-method is called on the thread-object). Similarly, we create a permission ThreadPost that represents the right to inhale a thread's postcondition when joining it. Additionally, since the creation of a thread-object (which determines which method the thread will execute and with which arguments) and the invocation of the start-method may happen at different points in the program, we add specification constructs that allow passing this information between different methods, as shown in Figure 2.38.

```
def foo(c: Cell, i: int) -> None:
    # requires acc(c.val)
    # ensures acc(c.val)
    # ensures c.val == old(c.val) + i
    ...


def create_thread(c: Cell):
    t = Thread(target=foo, args=(c,1))
    fork_thread(t, c)


def fork_thread(t: Thread, c: Cell):
    # requires MayStart(t)
    # requires getMethod(t) = foo || getMethod(t) = bar
    # requires getArg(t, 0) = c
    c.val = 15
    t.start(foo)
    join_thread(t, c)


def join_thread(t: Thread, c: Cell)
    # requires ThreadPost(t)
    # requires getMethod(t) = foo || getMethod(t) = bar
    # requires getArg(t, 0) = c
    # requires getOld(t, arg(0).val) = 15
    t.join(foo, bar)
```

Figure 2.38.: Three methods that create, fork, and join a new thread. The information which method the new thread will execute is passed between methods through custom specification constructs; the rights to start the thread and to assume its postcondition when joining it are modeled as permissions. We provide custom versions of the start and join methods that can be passed several options for the method the thread is executing. When the thread is forked, we generate a proof obligation that the thread's actual method (which can be specified using the getMethod function) is among the enumerated methods, and conditionally exhale the precondition of all options. When joining the thread, (some fraction of) its postcondition may be inhaled if the joining thread owns (some fraction of) the ThreadPost permission. The function getArg can be used to specify the arguments with which the thread's method was executed. Similarly, the function getOld expresses information about the program state at the point when the thread was forked (in this case, that the val field of its first argument had the value 15); this is useful in case the thread's method's postcondition relates its result or resulting state to its initial state via old-expressions.

As for locks, we require users to subclass Python's built-in Lock-method and define a predicate invariant in said subclass, as shown in Figure 2.39. That is, client code *must* work with subclasses of Lock (so that it is statically known, from the type of the lock, which invariant to in- or exhale) and are not allowed to use instances of the supertype Lock directly.

```
class CellLock(Lock[Cell]):

    predicate invariant(self, c: Cell):
        return Acc(c.value) * c.value > 0

def client(cl: CellLock)
```

Figure 2.39.: Declaration of lock invariants in a subclass of the built-in Lock-class.

2.3. Specification of Advanced Properties

Our goal is to be able to prove security properties of software systems. As explained in Chapter 1, this often requires proving a number of different kinds of properties on the code level. In this section, we explain the specification constructs we provide to achieve this goal, with one exception: We will explain our concept for non-interference specifications in the following two chapters.

2.3.1. Crash Freedom

The most basic property needed to ensure a system's *availability* is to prove that programs do not crash. We prove this property by default, that is, proof obligations for crash freedom are generated even if programmers provide no specification whatsoever.

We use different mechanisms to prevent different kinds of crashes, some of which we have already mentioned:

- Type errors and name errors are ruled out by the respective proof obligations in the Viper encoding (guided by hints from the external, untrusted, type checker) that ensure type correctness as well as the definedness of all referenced names, both local and global. This part includes a check that receivers of method calls or field accesses are never None.
- We use permissions to ensure that fields can only be accessed if they actually exist. In addition, we use predicates to bundle permissions, as is standard in permission logics.
- For operations on built-in types, we prove by default that they do not fail. For example, we generate proof obligations that ensure that code does not divide by zero, that it does not access lists out of bounds, and that it does not look up values in dictionaries for keys that do not exist.
- We allow user code to raise exceptions *only* if this is explicitly stated in the specification. For this purpose, we use explicit *exceptional postconditions*, denoted by the exsures keyword, that denote that a method may terminate with a specific kind of exception, and provides an assertion that describes the program state in this case. If a method does not have an exsures clause in its postcondition, this is interpreted as exsures False, so we prove by default that all exceptions that may be raised inside the method are also caught inside the method. Figure 2.40 shows an example.

```
def find_entry(l: List[Entry], id: int) -> Entry:
  # requires acc(l) && forall e in l: acc(e.id)
  # ensures acc(l) && forall e in l: acc(e.id)
  # ensures result in l && result.id = id
  # exsures NotFoundException e: acc(l) && forall e in l: acc(e.id)
  # exsures NotFoundException e: forall e in l: e.id != id
  ...
```

Figure 2.40.: Without requiring any specification to express this explicitly, we always prove that a method does not abort with an error from, for example, accessing a list out of bounds. Additionally, we prove that no method raises an exception *unless* it explicitly states that it may using an exceptional postcondition, like in this case: We then prove that if the exception is raised, the exceptional postcondition holds.

These checks are standard in existing automated verification tools, with the exception of the checks in the first two points, which, as we explained in the previous section, are part of our concept for verifying a dynamic language.

2.3.2. Progress

The second property that is required to ensure system availability is progress, i.e., programs must not get stuck. Often, ensuring progress is equated with proving *termination*, i.e., all loops in the program must terminate after a finite number of iterations, and the number of recursive calls must similarly be bounded; in concurrent programs, it may also involve proving other properties like the absence of deadlocks.

However, termination is too strong a property for programs that are intended to potentially run forever, like for example servers: the desired property here is *finite blocking*, i.e., no thread in the program will get stuck at a specific program point forever. To achieve this, we implement a verification technique by Boström and Müller [39]. This technique uses *obligations* as its main specification construct. Obligations, like permissions, are held by the current method, and can be passed around between methods. As the name suggests, holding an obligation obliges a thread to *fulfill* the obligation within a finite number of steps.

[39]: Boström et al. (2015), 'Modular Verification of Finite Blocking in Non-terminating Programs'

For example, methods can have a termination obligation in their precondition, which obliges the method to terminate. Termination is also relevant for thread interactions: A method may only join another thread if that thread, when forked, was given an obligation to terminate; otherwise, the joining thread could block forever waiting for the other thread to terminate. Similarly, when a thread acquires a lock, it also acquires an obligation to release the lock again within a finite amount of steps; Figure 2.41 shows an example that illustrates this.

```
def consumer_thread(l: CellLock) -> None:
  while True:
    l.acquire()
    print(l.cell.value)
    l.release()

def producer_thread(l: CellLock) -> None:
  while True:
    l.acquire()
    new_val = compute()
    l.cell.val = new_val
    l.release()

def compute() -> int:
  # requires MustTerminate
  ...
```

Figure 2.41.: Example of a concurrent program consisting of a producer thread and a consumer thread that communicate via a cell-object protected by a lock. Our system for proving finite blocking ensures that every thread that acquires a lock will release it again in a finite amount of time, by giving it a release-obligation when acquiring the lock. This obligation can only be fulfilled by releasing the same lock again. Any operations invoked before releasing the lock (like, here, the compute-method) must be proven to terminate, otherwise their invocation is not allowed.

The verification mechanism proposed by Boström and Müller [39] provides a system that ensures that all obligations will eventually be fulfilled (which also involves showing that deadlocks cannot occur, since that is obviously a possible cause of infinite blocking). It was adapted for use in Python verification as part of a Master's thesis by Astrauskas [11], who also designed the encoding of obligations into the Viper IVL. For details about this encoding or the exact design of the obligation specifications in Python, we refer the reader to this thesis.

[39]: Boström et al. (2015), 'Modular Verification of Finite Blocking in Non-terminating Programs'

[11]: Astrauskas (2016), 'Input-output verification in Viper'

While proofs of termination and absence of deadlocks are standard and supported in many existing tools, support for showing finite blocking is non-standard and goes beyond what typical verifiers currently offer.

2.3.3. Functional Properties

As explained, proving a security property of a system sometimes requires describing and verifying the functional behavior of its components, either as an auxiliary proof obligation to prove other properties, or to prove the *integrity* of a system by proving that important invariants can never be violated. As already stated, Nagini primarily uses pre- and postconditions to express functional behavior. To ensure that specifications are sufficiently expressive, we support a number of mechanisms to specify desired behavior:

- Assertions can contain *quantifiers*, which are important for example to express properties of data structures of statically unbounded size. As an example, a method finding the least element in a list or set may use a universal quantifier to specify that all elements in said collection are greater than or equal to the returned result, and that the returned result is contained in the collection.
- Programmers can define *pure functions* and use them in specifications. In particular, those functions can be *recursive*; recursive functions are often a useful way to express properties of recursive data structures, whereas quantifiers are generally useful to express properties of flat data structures like Python's (array-like) lists.
- We provide a number of built-in data types to be used in specifications. In particular, it offers support for pure sequences, sets, and multisets, which can be used as ghost state and used in specifications.
- In addition to the built-in types, programmers can define custom data types, particularly ADTs, to model custom concepts that can be referred to in specifications.

This set of specification constructs and data types constitutes the state of the art in most automated verification tools. Figure 2.42 shows a program that uses some of these specification constructs to model the desired behavior.

```
ADT msg = identifier(int) | key(str) | hash(msg) | encrypt(msg, str) | pair(msg, msg)

pure
def parse(b: bytes) -> msg:
  ...

class Message:
  ...

  predicate state(self):
    ...

  pure
  def abstract(self) -> msg:
    # requires self.state()
    ...

  def marshall(self) -> bytes:
    # requires self.state()
    # ensures self.state()
    # ensures parse(result) = self.abstract()
    ...
```

Figure 2.42.: Example program that manipulates stateful Message-objects and marshalls them into a bytestring format. An ADT is used to abstractly represent messages in specifications, and two pure functions are used to map both Message-objects and bytestrings to abstract messages. The permissions of a message object and all constraints that determine a valid message are encapsulated into the predicate state.

2.3.4. I/O Behavior

Finally, in order to specify the interactions of a system's component with its environment, which is often required to prove security properties of the entire system, we support a system for input-output-specifications. We build on a specification and verification technique proposed by

Penninckx et al. [172], which allows specifying I/O behavior in the form of *petri nets*.

[172]: Penninckx et al. (2015), 'Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs'

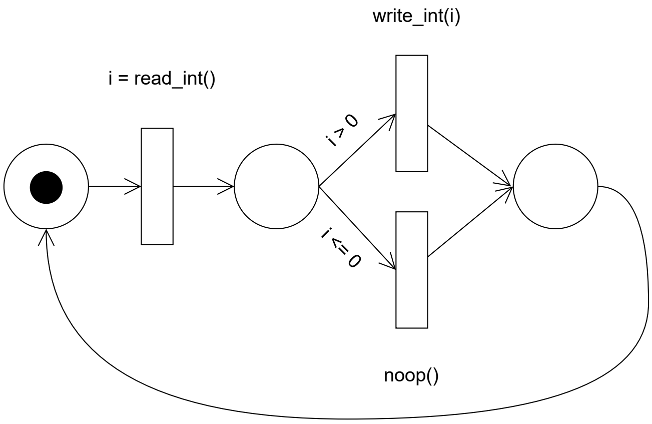


Figure 2.43.: Example of a simple I/O specification.

Figure 2.43 shows an example: In the initial state, a *token* is in the initial *place*. Performing an I/O operation requires having a token in a place from which such an operation is possible, and will move the token to the output place of the operation. In the example, the token is initially in the left place, from which only a read_int operation can be performed. After performing such an operation, reading some value i, the program may now perform a write_int with the argument i if the read i is strictly positive, otherwise, it may only perform a noop. After that, the program may start from the beginning by reading an integer again.

Specifying I/O behavior in this way allows for very expressive specifications and allows for modular verification. Going beyond the existing system by Penninckx et al., we combine the verification of I/O behavior with the system for ensuring finite blocking described above: Tokens can be designated to be obligations, meaning that the program is forced to make progress in the petri net by moving all its tokens after finite amounts of time. For example, when the token in the petri net in Figure 2.43 is declared to be an obligation, then a program *must* continually keep reading and writing integers; it may never cease doing so by, for example, running into an infinite loop or calling a non-terminating method between send and receive operations (see the example in Figure 2.44). The system for verifying this property was again developed by Astrauskas and is described in his thesis [11]. While I/O verification based on petri nets can easily be integrated into any separation logic based verifier, and is supported, for example, in VeriFast [110], our system is the only one that allows proving progress in the context of this specification approach.

[11]: Astrauskas (2016), 'Input-output verification in Viper'

[110]: Jacobs et al. (2011), 'VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java'

Crucially, the I/O specification mechanism we use is compatible with the one used by Sprenger et al. [207] in their Igloo framework, which allows soundly combining proofs about systems consisting of several components at an abstract level with proofs about the I/O behavior of individual components into a sound proof about the overall system and its implementations. That is, through the use of this I/O verification technique, our proofs about Python programs can be integrated into proofs about the systems they are a part of in a seamless manner.

[207]: Sprenger et al. (2020), 'Igloo: soundly linking compositional refinement and separation logic for distributed system verification'

```
io_operation print_positives(t):
  exists i, t2 :: read_int(t, i, t2) *
  i > 0 ==> (exists t3 :: write_int(t2, i, t3) * print_positives(t3)) *
  i <= 0 ==> (exists t3 :: noop(t2, t3) * print_positives(t3))

def keep_printing_postives():
  # requires print_positives(?t)
  while True:
    # invariant print_positives(?t)

    # unfold print_positives(t)
    i = os.read()
    if i > 0:
      print(i)
    wait()
```

Figure 2.44.: Textual specification of the petri net shown in Figure 2.43, where repetition is defined using corecursion. Since we prove that the program makes progress inside the petri net, we only allow this program if method wait promises to terminate.

2.3.5. Extensions

Our specification approach has been extended to support *abstract read permissions* [102], which allow for more expressive (and often simpler) specification of permission amounts than standard fractional permissions, in a Bachelor thesis by Schmid [193]. Additionally, Weber, in his Master's thesis [224], extended our specification and verification approach with a means to verify closures and higher-order functions.

[102]: Heule et al. (2013), 'Abstract Read Permissions: Fractional Permissions without the Fractions'

[193]: Schmid (2018), 'Abstract Read Permission Support for an Automatic Python Verifier'

[224]: Weber (2017), 'Automatic Verification of Closures and Lambda-Functions in Python'

2.4. Soundness

In this section, we will discuss possible ways of proving the soundness of (parts of) our verification approach. Since, currently, no such proof exists, it is of course possible that our technique is unsound in some cases; even though our goal has been to design a sound verification technique, we may, for example, have overlooked a language feature which violates an assumption we made when designing our technique (or, in fact, newer versions of Python might add such language features even if our assumption was initially correct). In addition, even if the technique is sound as designed, implementation errors may result in unsound behavior in practice, and so could incorrect (assumed) specifications for built-in types. In fact, our implementation makes one such unsound assumption deliberately, since it assumes that each object is equal to itself according to built-in equality methods. This assumption is vital for completeness, but is actually violated by the special floating point value NaN (like in other languages), which is not equal to itself. As long as no formal soundness proof exists, one could use alternative ways of increasing the confidence in the correctness of our technique, e.g., by testing assumed specifications or by using fuzzing techniques similar to those used for compilers [232]. In the remainder of this section, however, we will elaborate on the ways a formal soundness proof could be attempted.

[232]: Yang et al. (2011), 'Finding and Understanding Bugs in C Compilers'

First, it is important to note that we incorporate a number of existing verification techniques for which formal soundness proofs (or less formal but detailed soundness arguments) already exist. Examples of this are implicit dynamic frames [199] and concurrent separation logic [216], predicate families [171], as well as the verification techniques for I/O properties [172] and finite blocking [40] we build on. Of course, those

[199]: Smans et al. (2012), 'Implicit dynamic frames'

[216]: Vafeiadis (2011), 'Concurrent Separation Logic and Operational Semantics'

[171]: Parkinson et al. (2005), 'Separation Logic and Abstraction'

[172]: Penninckx et al. (2015), 'Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs'

[40]: Boström et al. (2015), 'Modular Verification of Finite Blocking in Nonterminating Programs'

soundness proofs are generally carried out with respect to a minimal programming language much simpler than the one we are dealing with, so that it is not guaranteed that the soundness proof carries over to our setting, or that the combination of these techniques is necessarily sound (in fact, in the next chapter, we will discuss an instance of such a combination potentially being unsound). Additionally, most of these techniques are defined in terms of Hoare logics, separation logics, or weakest precondition calculi, and our adaptation of them into an IVL encoding might have introduced errors.

In principle, we could choose to transfer these existing proofs to our new language and setting, either by completely re-doing the proofs in the new setting (reusing the proof structure), or by attempting to prove that an existing soundness result carries over from the original language and setting to ours. However, when dealing with mainstream languages of the complexity of Python, any such attempt would likely be a major undertaking. To the best of our knowledge, there is currently no general approach that would allow the re-use of soundness results in new settings, and as a result, most existing verification tools that integrate different existing techniques do not prove their implementations (e.g., IVL encodings) of these techniques sound, or only do so for a minimal core of their language [219]. This is certainly not satisfactory, but it is probably not surprising, since even defining the formal semantics of an expressive assertion language for a mainstream programming language (without actually proving a specific verification technique correct) is currently a major undertaking [123].

However, we do believe it would be possible to prove the soundness of different *aspects* of the verification technique presented in this chapter. The simplest way of doing so, we believe, would be to build on an existing logic or weakest-precondition calculus for Python, and show that the proof obligations imposed by our technique entail those of said logic or calculus, as has been done in the past for an encoding of Java bytecode into the Boogie IVL [124]. However, to the best of our knowledge, no such logic or calculus currently exists for Python.

Thus, the best basis for such a proof that we are aware of is small-step semantics for Python 3 defined by Politz et al. [178]. Building on this semantics, and either an axiomatic semantics or an operational semantics of the used subset of the Viper IVL, one would have to define a coupling invariant that relates the states of encoded Viper programs to the Python states they represent. The definition of such a coupling invariant would be simple for some aspects of the encoding, e.g., the contents of local and global variables. Other seemingly simple aspects might be surprisingly non-trivial, depending on the proof structure one aims for. For example, as we explained in Sec. 2.2.1.5, if a local variable is defined in a Python state, this is modeled in the Viper encoding by assuming the truth value of a boolean function for a specific input. That is, if this function is known to be true for a given input, the variable is definitely defined, if the variable may not be defined, then the function's value is unknown (i.e., the Viper program has states where its value is true and states where its value is false), but the function is never known to be false. As a result, a coupling invariant modeling this part of the encoding would necessarily have to map *sets* of Viper states to individual Python states. On the other hand, it is often desirable or even necessary to map individual Viper

[219]: Vogels et al. (2015), 'Featherweight VeriFast'

[123]: Lehner (2011), 'A formal definition of JML in Coq and its application to runtime assertion checking'

[124]: Lehner et al. (2007), 'Formal Translation of Bytecode into BoogiePL'

[178]: Politz et al. (2013), 'Python: The Full Monty'

states to sets of Python states, since the Python encoding may not contain some information present on the Python level. As a result, the concrete form of such a coupling invariant would have to depend on the parts of the encoding that need to be represented, and the properties that need to be proved.

Once such a mapping exists, several aspects of our approach could be proved sound. We will provide three examples here:

1. We could prove the soundness of our simplified object model. That is, we could show that when accesses to certain attributes and functions are forbidden, Python's actual complex object model is indistinguishable from our simplified object model, i.e., all accesses that succeed in the simplified version also succeed in the complex version, and return the same result.
2. We could prove the soundness of our type encoding. This part could be done in two steps: First, we could prove that our subtyping axioms are consistent and model a correct view of Python types (without multiple subtyping). Second, we could prove that the type assumptions we generate in the program are sound. We believe that this proof would be feasible in practice, since only a small part of the Python state (object types) needs to be modeled, and the crucial part of the coupling between Python states and Viper states is somewhat simple. Note that the soundness of the external type checking algorithm does not need to be proved, since we do not rely on its soundness and perform a full proof of type correctness in the verifier.
3. As part of our verification of the global, top-level program, we compute the dependencies of each top-level statement using a simple static analysis, to be able to check that (some overapproximation of) the dependencies of each statement is definitely defined at the point it is executed. The computation of these dependencies could be proved sound completely independently of its use in the IVL encoding, and subsequently, the entire verification of top-level statements could be proved sound as well.

We leave any such proofs as future work.

2.5. Implementation

We have implemented the specification and verification concept we have described in this chapter in the tool Nagini, a frontend for the Viper verification infrastructure [159], which is open source and available online[3]. In this section, we describe its architecture, usage, and some relevant details of the implementation.

[159]: Müller et al. (2016), 'Viper: A Verification Infrastructure for Permission-Based Reasoning'

[3] https://github.com/marcoeilers/nagini

2.5.1. Specifications

Nagini consists of two parts: The verifier itself, and a library of *contract functions*. While we have shown specifications as comments so far, Nagini's actual implementation uses these contract functions to allow users to add specification to the actual program text, in the style of Code Contracts [77]. That is, for example, a postcondition stating that a method

[77]: Fähndrich et al. (2008), *Code Contracts*

gives back permissions to a list l, and that its returned integer result is greater than all list elements, can be expressed by writing the statement Ensures(list_pred(l) **and** Forall(l, **lambda** e: Result() > e)) at the beginning of the method, where Ensures, list_pred, Forall, and Result are all contract functions declared in Nagini's contract library.

The call to the Ensures function introduces a postcondition, the call to list_pred represents a built-in predicate that gives permission to the contents of a list, the Result function represents the result returned by the method and can only be used in postconditions, and the Forall function call represents a universal quantifier; its first argument must be either a type or (as in this case) a collection, and its second argument must be a lambda-expression that must be true for all elements of the collection or all instances of the type.

Similarly, there are contract functions for preconditions, loop invariants, ghost statements (e.g. for folding predicates), permissions, obligations, and all other specification constructs supported by Nagini. Additionally, decorators are used, for example, to mark Python methods that are intended to be pure functions, or that represent predicates. Figure 2.45 shows an example of an annotated Python program, including a predicate and a termination obligation.

```python
from nagini_contracts.contracts import *
from typing import List
import db

class Ticket:
    def __init__(self, show: int, row: int, seat: int) -> None:
        self.show_id = show
        self.row, self.seat = row, seat
        Fold(self.state())
        Ensures(self.state() and MayCreate(self, 'discount_code'))

    @Predicate
    def state(self) -> bool:
        return Acc(self.show_id) and Acc(self.row) and Acc(self.seat)

def order_tickets(num: int, show_id: int, code: str=None) -> List[Ticket]:
    Requires(num > 0)
    Exsures(SoldoutException, True)
    seats = db.get_seats(show_id, num)
    res = []  # type: List[Ticket]
    for row, seat in seats:
        Invariant(list_pred(res))
        Invariant(Forall(res, lambda t: t.state() and
                         Implies(code is not None, Acc(t.discount_code))))
        Invariant(MustTerminate(len(seats) - len(res)))
        ticket = Ticket(show_id, row, seat)
        if code:
            ticket.discount_code = code
        res.append(ticket)
    return res
```

Figure 2.45.: Example program demonstrating Nagini's specification language. The contract functions are imported from Nagini's contract module in the first line. Note that functional specifications and postconditions are largely omitted to highlight the different specification constructs.

This specification style has the advantage that the syntax of specifications is exactly the same as for expressions in ordinary Python code, making them comparatively easy to write for Python programmers. Additionally, since specifications of this kind are part of the actual program's AST (which usually would not be the case if specifications were written in comments) they can be read by the verifier without requiring an additional parser.

When executing verified code, these specifications can be removed from the program automatically.

2.5.2. Architecture and Verification Workflow

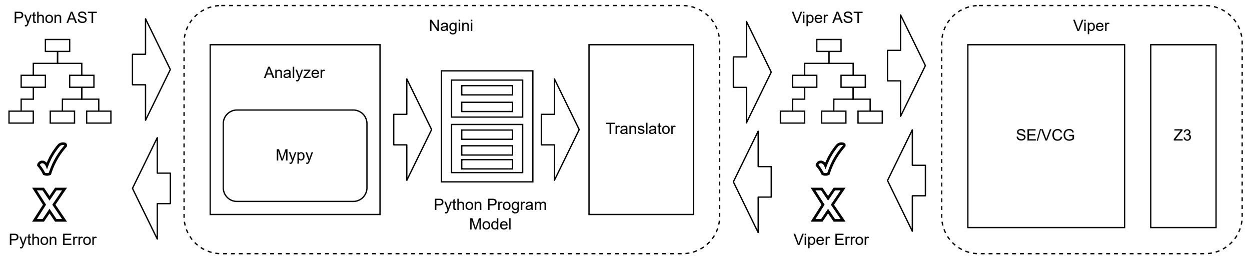


Figure 2.46.: Nagini architecture and verification workflow.

Nagini is implemented in Python, which enables it to re-use Python infrastructure including Python's built-in parser and AST. It interacts with Viper, which is implemented in Scala, via JPype[4], a library that allows Python code directly with the Java Virtual Machine (JVM), i.e., to directly invoke Viper's code from Python code.

Nagini's high-level architecture and verification workflow are shown in Figure 2.46. To verify a Python program, Nagini first invokes the Mypy type checker on the Python program [125]. Mypy implements the type system mentioned before; for variables (both local and global), it also implements partial type inference. We use a slightly modified version of Mypy which ignores type errors based on the potential None-ness of receivers: Ordinary Mypy will complain when a field e.f is accessed or a method e.m() is called and the type of e is an Optional type, i.e., it may be None. We instead let the verifier make this check, which is of course more precise.

[125]: Lehtosalo et al. (2017), *Mypy - Optional Static Typing for Python*

If the program contains type errors, Nagini rejects the program right away. Otherwise, it now performs an *analysis* of the program, during which it builds up a model of the Python program: It collects information about all modules in the program, including all classes, methods, fields, and variables. This step is necessary because, as explained earlier, Python programs are simply sequences of statements (some of which will be declarations), and Python ASTs contain very little information by themselves; references, for example, do not contain any information regarding their target, and can refer to variables, modules, classes, or functions. As part of the analysis step, Nagini extracts type information from Mypy, and enriches its internal model with this information. The leaves of the created model of the Python program contain references to the analyzed Python AST nodes.

In the second step, Nagini then walks through the created model and performs the encoding of each part of the program (by directly calling constructors of Viper AST nodes through JPype), using the references to the Python AST nodes they represent when necessary. Each created Viper AST node is annotated with an identifier that allows mapping it back to the Python AST node that it represents; this information is used for error reporting later. The created Viper AST is then handed to either one of Viper's two backends, one of which is based on symbolic execution and

[4] https://pypi.org/project/JPype1

one of which uses verification condition generation through an encoding into the Boogie IVL [18]. Viper will use the SMT solver Z3 [156] to verify the program; it will then either report successful verification, in which case Nagini also reports a success, or it returns a list of verification failures in the Viper program, which include the Viper AST nodes that caused the error. Nagini then converts each error message back to a message that can be understood on the Python level, using the attached identifiers to find the Python AST nodes that caused the error. When desired, Nagini additionally generates a counterexample; examples of error messages and a counterexample will be shown later in this section.

[18]: Barnett et al. (2005), 'Boogie: A Modular Reusable Verifier for Object-Oriented Programs'

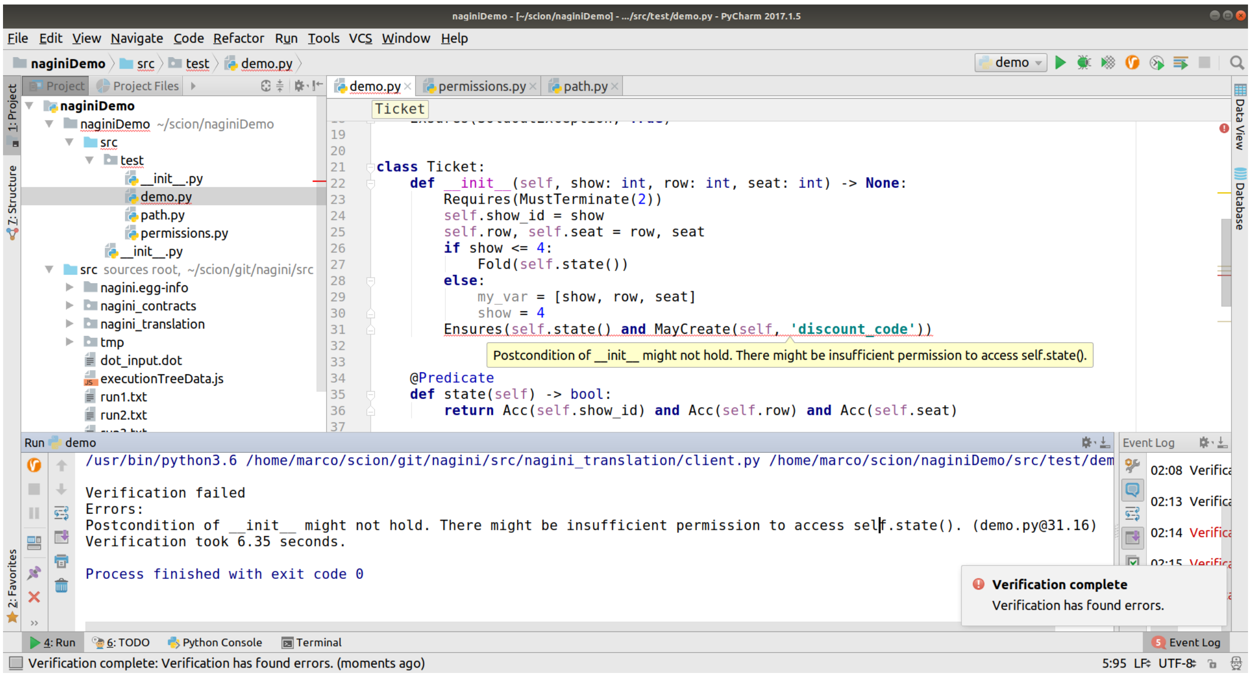
[156]: Moura et al. (2008), 'Z3: An Efficient SMT Solver'

2.5.3. Tool Interaction

Nagini is a command line tool which, for basic usage, only expects a path to a file to be verified as an argument. It will then automatically find all other relevant files that are imported by the main file.

In addition, Nagini has a number of command line options to customize its behavior:

- Users can use the "–select" option to enumerate some set of methods and classes to verify; Nagini will then verify *only* the selected elements, and not the remaining parts of the input program. Other elements are included only if the selected parts depend on them; in particular, if selected methods depend on (i.e., call) other methods, Nagini will include stubs of those other methods but not their bodies, so that their correctness is assumed. Similarly, functions and axioms that model the type system (as described before) will be generated only for those types that are (transitively) referenced in the selected code.
- The "–ignore-global" option explicitly disables verification of top-level code, and of the order of global declarations, as described in Sec. 2.2.3.
- The obligation system explained in Sec. 2.3 (which causes some performance overhead even in code that does not use obligations) can be disabled using the flag "–ignore-obligations".
- Nagini's output during the encoding can be influenced by setting a log level; additionally, the "–print-viper" flag can be used to prompt Nagini to output the generated Viper program, which is useful for debugging.
- When the "–counterexample" flag is used, Nagini will output a counterexample for every verification error, as we will describe in the next subsection.
- While a Nagini installation comes with matching versions of Viper and Z3, users can supply paths to custom versions of all dependencies to use those instead of the defaults.
- Finally, Nagini has a server mode that starts running a Nagini instance, performs some setup work, and then waits to be invoked on input programs. An accompanying client script invokes the server on a specific file and reports the generated result.
  This mode is useful to improve performance in practice, since a substantial amount of work (starting a JVM, initializing some parts of Viper, and parsing some Viper code that is internally



```
class Ticket:
    def __init__(self, show: int, row: int, seat: int) -> None:
        Requires(MustTerminate(2))
        self.show_id = show
        self.row, self.seat = row, seat
        if show <= 4:
            Fold(self.state())
        else:
            my_var = [show, row, seat]
            show = 4
        Ensures(self.state() and MayCreate(self, 'discount_code'))

    @Predicate
    def state(self) -> bool:
        return Acc(self.show_id) and Acc(self.row) and Acc(self.seat)
```

Postcondition of __init__ might not hold. There might be insufficient permission to access self.state().

```
/usr/bin/python3.6 /home/marco/scion/git/nagini/src/nagini_translation/client.py /home/marco/scion/naginiDemo/src/test/dem

Verification failed
Errors:
Postcondition of __init__ might not hold. There might be insufficient permission to access self.state(). (demo.py@31.16)
Verification took 6.35 seconds.

Process finished with exit code 0
```

Verification complete
Verification has found errors.

Verification complete: Verification has found errors. (moments ago)

Figure 2.47.: Nagini PyCharm plugin: Nagini can be run by using a button at the top right, and subsequently, verification errors will be highlighted in the code.

used to model parts of Python's standard library) needs to be performed for every run of Nagini; in this mode, this work can be performed once up front, and does not have to be repeated for subsequent usages. Additionally, this mode allows re-using a single JVM instance, which is advantageous because the JVM just-in-time-compiles Viper's code, and repeated invocations will therefore lead to better performance.

In addition to usage from the command line, we have developed an experimental Nagini plugin[5] for the popular PyCharm IDE[6], which adds a button to invoke Nagini to the IDE and reports verification errors errors highlighted directly in the code. Figure 2.47 shows a screenshot of Python code in the PyCharm IDE with the Nagini plugin, including Nagini's specifications and a reported error.

2.5.4. Error Reporting and Counterexamples

As stated before, Nagini maps Viper verification errors back to messages that can be understood on the Python level. As a (very artificial) example, consider a version of the program in Figure 2.45 where constructor of the Ticket class has been modified as shown in Figure 2.48. Here, the Fold statement is executed only conditionally, namely if the show value is at most four, and otherwise (to illustrate a changed state in the counterexample below) a list is created and assigned to a new local variable, and the value of show is set to four. Verification fails for this modified version, because the predicate self.state(), which the postcondition promises to give back to its caller, does not always exist.

Viper will report the following error message:

[5] Available at https://github.com/marcoeilers/nagini-pycharm

[6] https://www.jetbrains.com/pycharm/

```
def __init__(self, show: int, row: int, seat: int) -> None:
    self.show_id = show
    self.row, self.seat = row, seat
    if show <= 4:
        Fold(self.state())
    else:
        my_var = [show, row, seat]
        show = 4
    Ensures(self.state() and MayCreate(self, 'discount_code'))
```

Figure 2.48.: Alternative constructor of class Ticket that is incorrect and will result in a verification error.

```
[postcondition.violated:insufficient.permission] Postcondition of Ticket___init__ might not
        hold. There might be insufficient permission to access Ticket_state(self). (example.
        py@27.16)
```

This error message references the internal names (i.e., the names used in the Viper encoding) of the constructor and the state predicate. Using the information it previously attached to the created AST nodes, Nagini maps these back to the original Python AST nodes, to create the following message:

```
Postcondition of __init__ might not hold. There might be insufficient permission to access
        self.state(). (example.py@27.16)
```

This message now uses the correct names. In other cases, Nagini also changes the kind of error message it reports: For example, if a local variable is read before it is defined, Viper will report an error stating that the precondition of a function checkDefined does not hold (see Sec. 2.2.1.5). Nagini will map this error back to an error that clearly states that a local variable has been read before its definition.

As mentioned before, Nagini can also generate counterexamples for every reported error. A counterexample for the example we just described could look like this:

```
Old store:
  show -> 5,
  row -> True,
  seat -> 45,
  self -> Ticket0
Old heap:
  Ticket0 -> { }
Current store:
  show -> 4,
  row -> True,
  seat -> 45,
  self -> Ticket0
  my_var -> list0
Current heap:
  Ticket0 -> { show_id -> 5, row -> True, seat -> 45 },
  list0 -> { len(list0) -> 3, list0[0] -> 5, list0[1] -> True, list0[2] -> 45 }
```

A programmer can learn from this counterexample that an error can occur (the self.state() is not a part of the heap) in a situation where the show variable was initially 5.

In general, Nagini shows a value for each local variable at the beginning of the invocation of the current method or constructor (old store) and at the point when the error occurs (current store). The two stores can differ because additional local variables can be defined (like, in this case, my_var), or the values of the parameter variables can be changed by the method (like show in this case). References to class instances will be represented simply by a name ("Ticket0" in this case for the self-variable) that contains the type of the instance.

The heap (again in two versions) then shows the values of the fields of each of those instances; crucially, it will only show values for fields for which the current method has permission. Predicate instances are also shown here; if the current heap contained an instance of self.state(), then this predicate would be shown as part of the heap. For built-in types like lists, we show their important attributes, in this case the list's length and the values for its three different indices. We use this format (as opposed to showing a list literal [5, True, 45]) because often, list contents will be unconstrained, leading to counterexamples with lists too large to show as literals, where only the values of *some* specific indices are relevant for the counterexample.

Note that, in our example, the row parameter has type int but contains a boolean value. This is possible because in Python, bool is a subtype of int, and this fact is modeled accurately in Nagini; every integer variable can also contain a boolean value.

To generate counterexamples, Nagini extracts the stores and heaps of the generated Viper program from Viper, and then translated all elements back to the Python level. That is, it maps the names of variables, fields, classes and predicates back to their names on the Python level, and it translates all values. The latter step is more complex than it might seem, since, as we mentioned before, *all* values on the Viper level are simply uninterpreted reference values. Translating a value v back to the Python level therefore requires two steps:

1. Determine the type of the value, that is, evaluate the value of typeof(v) in the counterexample. Since this actual type may actually be an unknown type (since the encoding allows having more types than the currently known ones to be modular, as discussed previously), we then have to find the most precise known supertype of this actual type by evaluating the issubtype relation. In the example, this step would tell us that row actually contains a boolean value.
2. Using this type information, evaluate v. For example, if v has type bool or int, this involves evaluating the respective internal unboxing function bool_unbox or int_unbox; if it is a list, this involves evaluating its length and its known values.

2.5.5. Limitations

Nagini's implementation has a number of limitations that are not fundamental to the used verification approach.

Performance: As it is a prototype implementation, Nagini's performance could be improved in a number of ways. In particular, some elements of its encoding (e.g. type checks, checking the definedness of variables, wrapping integers into references) are needed to be able to verify code in *some* cases, but could sometimes be omitted; for example, Nagini could check on the AST level if a variable is definitely assigned before it is read, and could subsequently omit checks of variable definedness in the encoding. Similarly, integer wrapping, type checking, and some other aspects of the encoding could be omitted in some cases if a static check on the AST is sufficient to determine that they are not needed for a specific program or method.

Additionally, the implementation has not been optimized with regards to caching the results of frequently-used operations; in particular, since the interaction between Python code and the JVM through a library that must rely on reflection in its own implementation is likely not very performant, we believe that performance could be improved by the results of method and constructor invocations on the JVM.

Furthermore, as we will also show in the evaluation section, Nagini's implementation is currently unable to encode only a part of a given input program; its flag to select certain parts to verify only leads to the exclusion of unneeded program parts *after* the encoding and before verifying the encoded program. This limitation is not fundamental and can be removed, but this would require refactoring the code base.

We leave these optimizations as future work.

Language features: In addition to the language features already discussed in Sec. 2.2.4, there are a number of Python's language features currently not supported by Nagini. One example is the nested definition of functions or classes: Nagini currently expects all classes to be top-level definitions, and only allows function definitions inside classes or on the top level. While nested functions can complicate verification due to captured state, in many cases, they do not complicate verification at least in principle; thus, their support would not be difficult to add. As mentioned before, Weber [224] created an extended version of Nagini that supports closures and higher-order methods in general, but this support is currently not part of the standard Nagini version.

[224]: Weber (2017), 'Automatic Verification of Closures and Lambda-Functions in Python'

There are some more involved language constructs that are currently not supported; one important aspect is asynchronous computation using the async and yield statements. Additionally, since Python is a comparatively large language, there are a number of simpler statement types that are currently not supported, like for example set and map comprehensions (list comprehensions are supported).

Standard library: Nagini models Python's basic types, like integers, strings, lists, sets and dictionaries. Support for floating point numbers exists, but is very incomplete; they are modeled simply as uninterpreted types. Other types are currently not included by default.

Nagini is built to be easily extendable with new (uninterpreted) types and built-in methods, even without modifying the verifier code at all: New built-in types can be added simply by adding entries to an internal JSON-file. To add a method (or pure function) to a built-in type, it is sufficient to add a function or method entry containing the method's signature in said JSON-file, and to then add a Viper function or method that encodes this method's behavior to a file in its resources folder.

However, adding new non-reference types (e.g. modeling Python's floating point numbers as actual, interpreted floats on the Viper level) would require (simple) changes in the actual code of the verifier.

Note that Nagini also supports the use of stub files to model the behavior of (non built-in) types and libraries; that is, users can supply files containing Python classes and method stubs that contain only signatures and specifications, but no implementation.

Counterexamples: Counterexamples are currently only supported if Viper's symbolic execution backend is used. Additionally, heap parts whose corresponding permissions are quantified are currently not displayed in Nagini's output.

2.6. Evaluation

In addition to having a comprehensive test suite of over 12,500 lines of code, we have evaluated Nagini on three sets of examples to demonstrate its ability to verify 1) typical simple algorithm implementations, 2) code from a real larger code base, and 3) properties relevant for proving the security of a larger system.

2.6.1. Verification of Simple Algorithms

Table 2.2.: Experiments. For each example, we list the lines of code (excluding whitespace and comments), the number of those lines that are used for specifications, the length of the resulting Viper program, properties (SF = safety, FC = functional correctness, FB = finite blocking, IO = input/output behavior) that could be verified (✓), could not be verified (✗) or were not attempted (-), and the verification times with Viper's SE backend, sequential and parallelized, in seconds.

	Example	LOC / Spec.	Viper LOC	SF	FC	FB	IO	T_{Seq}	T_{Par}
1	rosetta/quicksort	31 / 10	635	✓	-	✓	-	8.48	8.31
2	interactivepython/bst	145 / 65	947	✓	✓	-	-	57.44	41.80
3	keon/knapsack	33 / 10	864	✓	-	-	-	19.39	14.49
4	wikipedia/duck_typing	19 / 0	486	✓	-	-	-	1.82	1.92
5	example	40 / 19	736	✓	-	✓	-	6.11	5.91
6	verifast/brackets_checker	143 / 82	1081	✓	✓	✓	✓	7.66	6.63
7	verifast/putchar_with_buffer	139 / 88	865	✓	-	✓	✓	4.74	4.29
8	chalice2viper/watchdog	66 / 22	769	✓	-	✓	-	3.66	3.41
10	parkinson/recell	46 / 25	561	✓	✓	-	-	2.09	2.07

The first part of our evaluation consists of (parts of) implementations of standard algorithms from the internet[7], the example from Fig. 2.45, as well as examples from other verifiers translated to Python. Table 2.2 shows the examples and which properties were verified; the functional property we proved for the binary search tree implementation is that it maintains a sorted tree. The examples cover language features like inheritance (example 10), comprehensions (3), dynamic field addition (6), operator overloading (3), union types (4), threads and locks (9), as well as specification constructs like quantified permissions (6) and predicate families (10). Nagini successfully verifies all examples.

All runtimes shown in this section were measured by averaging over ten runs on a Lenovo Thinkpad T450s running Ubuntu 16.04, Python 3.5 and OpenJDK 8 on a warmed-up JVM. They show that Nagini can effectively verify non-trivial properties of real-life Python programs in reasonable time. Due to modular verification, parts of a program can be verified independently and in parallel (which Nagini does by default), so

[7] We chose examples that do not make use of unsupported dynamic features or external libraries from rosettacode.org, interactivepython.org and github.com/keon/algorithms.

that larger programs will not inherently lead to performance problems. This is demonstrated by the speedup achieved via parallelization on the two larger examples; for the smaller ones, verification time is dominated by a single complex method. Additionally, the annotation overhead for these examples is, on average, less than one line of specification or proof annotation per line of code, though, as we will show in Sec. 2.6.3, this ratio can be much higher when verifying more complex properties.

2.6.2. Verification of Real-World Code

The second set of examples is taken from the source code of the Python implementation of the router for the SCION internet architecture [21]. More precisely, the verified methods are the ones that are executed when the SCION router forwards a standard SCION packet that does not use any extensions. These examples are challenging because they are taken from a larger code base that was not written with verification in mind. The code has a large number of dependencies to other parts of the SCION code and to outside libraries, which need to be modeled via stubs. The fact that this code can be verified demonstrates Nagini's high coverage of Python language features and, more generally, its ability to verify code from real, large code bases.

[21]: Barrera et al. (2017), 'The SCION Internet Architecture'

The examples were verified by Forster [83] as part of his Bachelor Thesis; all in all, the verified code base consists of 334 lines of verified code, plus a large number of stub files to model dependencies. More details on the code can be found in this thesis. We have taken the code and modified it in a number of places to accommodate changes that have been made in Nagini since the project was conducted.

[83]: Forster (2018), 'Static Verification of the SCION Router Implementation'

Since the verified code is taken from a router, there are many different (security-related) properties to be verified, in particular, crash freedom, progress (i.e., termination), as well as output-behavior. Part of the output-behavior is showing that the correct data is forwarded, which in turn requires modeling and/or verifying functional behavior of methods that modify package data. In addition to I/O specifications and termination obligations, verifying this code requires using a large number of predicates and predicate families to model the properties and permissions that constitute the class invariants of various classes (e.g. the router itself, packets, and packet components), often in conjunction with quantifiers. A common example is that an object representing e.g. a path contains several lists of other objects (e.g., of steps in the path), and therefore its class invariant contains the (quantified) class invariants of all objects in the list. We also make extensive use of pure functions to model aspects of those classes, sometimes using methods that already existed in the code base and marking them as pure, so that they are verified for crash and side effect freedom and can subsequently be used in the specifications of other methods. Additionally, ADTs were used throughout to model SCION packets and express the desired functional behavior in specifications.

Table 2.3 shows all example methods and functions and details, for each one of them, what kind of properties were proved. Note that, due to the interwoven nature of the code base and the resulting large number of dependencies between different parts, the verification of the depicted 17 methods required Nagini to read and partly encode a total of 56 modules, 188 classes, and 591 methods (whose behavior, where it is relevant, we modeled using stubs).

We verify each method individually, since the amount of dependencies makes it infeasible to verify all methods at once (Nagini does not terminate in a reasonable amount of time if this is attempted), demonstrating the need for modular verification. However, many methods share some dependencies (e.g., predicates, pure functions used in specifications, and stubs of called methods) which have to be re-checked for well-definedness and other properties on each verification run, so there is a significant overlap between the work that is done when some closely related methods are verified. In practice, it is therefore advisable to verify small sets of closely-related methods at once. Alternatively, it might well be possible to specify the code in a way that reduces common dependencies, but this was not a focus of the aforementioned Bachelor project.

For each method, we show both the total verification time needed by Nagini, and the amount of time used by Viper to verify the encoded program. The latter is interesting because Nagini currently always reads and encodes the *entire* program it is given, and subsequently removed unneeded parts if it is instructed to verify only a specific subset of the program (in this case, a single method and its dependencies). This is a limitation of the implementation that exists for historical reasons, however, it is in no way a fundamental limitation of Nagini's encoding or verification approach. As an unfortunate consequence of this, for the SCION code base with its many dependencies, Nagini reads and encodes the aforementioned 56 modules, 188 classes and 591 methods every time even when asked to verify a single method, and there is therefore a significant static overhead in its encoding time. We therefore also show the verification times in Viper that directly show the actual verification effort required per method.

The verification times for all examples are shown in Table 2.3. As expected, the encoding times are basically constant for all examples, always taking 31-35 seconds, which is obviously a significant overhead. Verification times, on the other hand, vary between 1.5 seconds for the simplest methods, 10-35 seconds for many normal-sized methods, and 234 and 114 seconds for the two most most complex ones. We believe that (ignoring the constant encoding overhead) the verification times are acceptable in practice for a code base of this complexity, and could likely be reduced by further optimizing Nagini (see the discussion of limitations of the current implementation).

2.6.3. Verification of Security Properties

Finally, we have verified an implementation of a two-party authentication protocol standardized as ISO/IEC 9798-3, as part of the evaluation of the Igloo methodology [207]. In the protocol, the *initiator* A first sends out a message A, B, N_A, where B is the intended recipient of the message, i.e., the *responder*, and N_A is a nonce. B then responds with the message N_B, N_A, A, where N_B is another nonce, and signs the entire message with its private key.

[207]: Sprenger et al. (2020), 'Igloo: soundly linking compositional refinement and separation logic for distributed system verification'

For this protocol, Sprenger et al. have proved an injective agreement property [140] on an abstract model of the protocol with a Dolev-Yao

[140]: Lowe (1997), 'A Hierarchy of Authentication Specification'

Table 2.3.: Verified methods from the SCION implementation, verified properties, and verification times, averaged over five runs. T denotes the total verification time, and its parts T_E and T_V denote the time used by Nagini for the encoding and the time used by Viper to verify the encoded program, respectively. Verified properties include crash freedom (Safe), termination (Term), functional correctness (Func), and input/output behavior (IO). Methods marked as pure are verified for crash and side effect freedom, so that they can be used in the specifications of other methods.

Class	Method	Safe	Term	Func	IO	Pure	T	T_E	T_V
Router	handle_extensions	x	x				120.96	34.42	86.54
	verify_hof	x	x	x			70.12	34.64	35.49
	send	x	x	x	x		149.23	35.38	113.85
	_calc_fwding_ingress	x	x	x			269.07	34.40	234.67
	_egress_forward	x	x	x	x		132.78	34.76	98.02
	_link_type	x	x				44.81	31.04	13.78
	_process_flags	x	x				37.36	34.21	3.16
	_validate_segment_switch	x	x	x			53.47	31.29	22.18
SCIONPath	get_curr_if	x				x	44.65	33.71	11.94
	get_fwd_if	x				x	45.10	33.87	11.22
	get_hof	x				x	43.15	32.78	10.37
	get_hof_ver	x	x	x			52.46	31.34	21.12
	get_of_idxs	x					42.84	35.14	7.70
	get_iof	x				x	43.87	33.97	9.90
	is_on_last_segment	x				x	43.78	34.89	8.89
	inc_hof_idx	x	x	x			66.49	32.31	34.18
HopOpaqueField	verify_mac	x					33.98	32.37	1.61

attacker [62], and refined the system according to the Igloo methodology to I/O specifications that the individual implementations of both parties have to fulfill. We have then implemented both roles in Python and verified them with respect to this derived functional specification. That is, due to the pre-existing correctness proof on the protocol level and the connection between protocol level proof and I/O specification according to the Igloo methodology, correctness proofs of the implementations lead to a system that is proved correct by a single full-stack soundness proof only depending on assumptions about the environment (i.e., the network and the adversary) and assumptions made during code verification; as for the latter, the proof assumes the correctness of libraries for encryption and socket I/O.

On the level of the implementation, we soundly model the fact that actual communication happens in the form of bit strings, while the abstract model is specified in terms of abstract terms. The specifications of both roles have I/O components that state which messages may be sent and received, and proving them correct has functional components; in particular, it requires proving that the sent messages are composed correctly and received messages are validated correctly by both parties. The latter requires modeling abstract message terms, including nonces, identifiers, asymmetric encryption, and tuples (which we do using ADTs), and it requires modeling the mapping between bitstrings and abstract message terms.

The code consists of a total of 886 lines of code, of which 82 are implementation and 804 are specification. The initiator and responder programs take 24.97 and 29.82 seconds to verify, respectively, again averaged over five runs.

While the implementations are comparatively simple, the proved specification along with its connection to the higher-level protocol proof

[62]: Dolev et al. (1983), 'On the security of public key protocols'

demonstrates Nagini's ability to verify implementation properties that are required to prove security properties of entire *systems*.

2.7. Related Work

The past decades have seen the development of a large number of automated verification tools. While some such tools are constructed in a monolithic way, like VeriFast [110] and SecC [72], most modern tools are, like Nagini, built on top of intermediate verification languages. Examples are Dafny [127], VCC [48], Spec# [130], and GPUVerify [34], which are built on top of Boogie [18], Frama-C [54] and Krakatoa [81], which are built on top of Why3 [82], and VerCors [38], Prusti [12], and Gobra [228], which are (like Nagini) built on top of Viper [159].

For each of these verifiers, the specific challenges that arise tend to be different based on the targeted languages; for example, Prusti, which targets the Rust language, focuses on extracting information from Rust's type system to reduce specification overhead, GPUVerify and VerCors both target GPU kernels and have to soundly model massively parallel execution, and many of the tools mentioned above target object-oriented programs and therefore have to model dynamic binding and object invariants. All of them, however, model statically-typed languages.

Not many existing tools target deductive verification for dynamic languages. To the best of our knowledge, Nagini is the only verifier specifically built to verify Python code. The only existing alternative for verifying Python code against custom specifications is the K-framework [183], which allows users to define a language syntax and semantics, and, based on that, automatically derives an interpreter, compiler, and even automated verifier for the language. Guth has defined a semantics for Python 3 in this framework [95], thus enabling verification of Python code. Since this semantics accurately captures the behavior of, for example, attribute lookups, this approach is in principle able to reason about any Python code; this is unlike our approach in Nagini, which does not model some parts of the language, and furthermore requires additional restrictions like static typing. On the other hand, K generally does not offer any high-level specification constructs that allow users to express properties in an abstract and modular way; properties have to be specified on the level of the state of the program. We believe that this is a substantial drawback when targeting verification of real, substantial code bases.

The only other tools for reasoning about Python we are aware of perform various kinds of static [84, 85, 97, 126, 142, 154] or dynamic [93] analysis. Most of them focus on inferring type information, while some [85, 215] go beyond that and either find errors or infer specifications on the input data.

JavaScript is another popular dynamic language, for which a number of verification tools has been developed. JavaScript differs from Python in that it does not have similarly complex semantics for operations like attribute lookup, or meta classes that modify class behavior; however, it has its own set of complex language features which particularly make local reasoning difficult (including, for example, an emulated store that

[110]: Jacobs et al. (2011), 'VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java'

[72]: Ernst et al. (2019), 'SecCSL: Security Concurrent Separation Logic'

[127]: Leino (2010), 'Dafny: An Automatic Program Verifier for Functional Correctness'

[48]: Cohen et al. (2010), 'Local Verification of Global Invariants in Concurrent Programs'

[130]: Leino et al. (2008), 'Using the Spec# Language, Methodology, and Tools to Write Bug-Free Programs'

[34]: Betts et al. (2012), 'GPUVerify: a verifier for GPU kernels'

[18]: Barnett et al. (2005), 'Boogie: A Modular Reusable Verifier for Object-Oriented Programs'

[54]: Cuoq et al. (2012), 'Frama-C - A Software Analysis Perspective'

[81]: Filliâtre et al. (2007), 'The Why/Krakatoa/Caduceus Platform for Deductive Program Verification'

[82]: Filliâtre et al. (2013), 'Why3 - Where Programs Meet Provers'

[38]: Blom et al. (2014), 'The VerCors Tool for Verification of Concurrent Programs'

[12]: Astrauskas et al. (2019), 'Leveraging Rust types for modular specification and verification'

[228]: Wolf et al. (2021), 'Gobra: Modular Specification and Verification of Go Programs'

[159]: Müller et al. (2016), 'Viper: A Verification Infrastructure for Permission-Based Reasoning'

[183]: Rosu et al. (2010), 'An overview of the K semantic framework'

[95]: Guth (2013), 'A formal semantics of Python 3.3'

[84]: Fritz et al. (2017), 'Cost Versus Precision for Approximate Typing for Python'

[85]: Fromherz et al. (2018), 'Static Value Analysis of Python Programs by Abstract Interpretation'

[97]: Hassan et al. (2018), 'MaxSMT-Based Type Inference for Python 3'

[126]: Lehtosalo et al. (2018), *Google LLC*

[142]: Maia et al. (2012), 'A Static Type Inference for Python'

[154]: Monat et al. (2020), 'Static Type Analysis by Abstract Interpretation of Python Programs'

[93]: Grue et al. (), *PyAnnotate*

[85]: Fromherz et al. (2018), 'Static Value Analysis of Python Programs by Abstract Interpretation'

[215]: Urban (2020), 'What Programs Want: Automatic Inference of Input Data Specifications'

behaves differently than in other languages and makes reasoning more difficult [88]).

JaVerT is a symbolic execution tool that precisely models JavaScript's complex semantics using the JSIL intermediate representation [189]. However, like other existing symbolic execution tools for JavaScript [135, 191, 195, 227], JaVerT is not able to *modularly* reason about code. More recently, JaVerT 2.0 [190] can perform symbolic testing, compositional testing, and verification using bounded symbolic execution. To be compositional, it explicitly specifies frame information (including negative frame information that states that some attributes are *not* present; somewhat similar to our MayCreate permissions), and uses an instrumented semantics to incorporate this information. Specifications are structural in nature and therefore more verbose than nominal types; unlike our approach, JaVerT 2.0 is limited to standard functional properties using pre- and postconditions. The authors' later work, Gillian [143], implements separation logic based modular reasoning for JavaScript based on similar principles.

Since its creation, Nagini has been used for multiple purposes. As mentioned before, Forster [83] used Nagini to verify the correctness of a part of the Python implementation of the SCION router. Nagini has also been used to verify two of the three case studies of the Igloo framework [207]; one of them has been mentioned in section Sec. 2.6, the other is an implementation of a distributed leader election algorithm. More recently, Nagini has been used by Gangopadhyay et al., who use it to verify a novel combination of rule-based methods and reinforcement learning for automated driving [87]. At Marktoberdorf Summer School 2018, Nagini has been used for teaching purposes as a case study for building automated verifiers.

The encoding generated by Nagini has similarly been used for different purposes: Becker et al. have used Nagini-generated Viper programs as case studies for tool for finding problematic quantifier instantiations in SMT solvers [28]. Rubbens has considered Nagini's encoding of type information, predicate families, and exception handling for use in the VerCors verifier [184]. Dardinier et al. formally examine the topic of inlining method calls in the context of permission logics, and use Nagini's encoding of field writes with its use of MayCreate permissions as a case study [55]. Ongoing work by Dohrau et al. builds a learning-based inference for permission specifications for Viper programs, and shows that it can infer specifications for Viper programs generated by Nagini.

Nagini's code base has been the basis for 2vyper, a verifier for Ethereum smart contract which we will present in Chapter 5. Nagini's implementation of counterexample extraction has similarly served as the basis for the later implementation of this feature in Viper itself [100].

As mentioned before, Nagini served as a case study for an implementation of abstract read permissions in Viper by Schmid [193], and it has been extended with support for verifying code using closures by Weber [224]. It has been further extended to verify information flow security [66, 148], which we will discuss in Chapter 4.

[88]: Gardner et al. (2012), 'Towards a program logic for JavaScript'

[189]: Santos et al. (2018), 'JaVerT: JavaScript verification toolchain'

[135]: Li et al. (2014), 'SymJS: automatic symbolic testing of JavaScript web applications'

[191]: Saxena et al. (2010), 'A Symbolic Execution Framework for JavaScript'

[195]: Sen et al. (2015), 'MultiSE: multipath symbolic execution using value summaries'

[227]: Wittern et al. (2017), 'Statically checking web API requests in JavaScript'

[190]: Santos et al. (2019), 'JaVerT 2.0: compositional symbolic execution for JavaScript'

[143]: Maksimovic et al. (2021), 'Gillian, Part II: Real-World Verification for JavaScript and C'

[83]: Forster (2018), 'Static Verification of the SCION Router Implementation'

[207]: Sprenger et al. (2020), 'Igloo: soundly linking compositional refinement and separation logic for distributed system verification'

[87]: Gangopadhyay et al. (2021), 'Hierarchical Program-Triggered Reinforcement Learning Agents for Automated Driving'

[28]: Becker et al. (2019), 'The Axiom Profiler: Understanding and Debugging SMT Quantifier Instantiations'

[184]: Rubbens (2020), 'Improving Support for Java Exceptions and Inheritance in VerCors'

[55]: Dardinier et al. (2022), 'Verification-Preserving Inlining in Automatic Separation Logic Verifiers (extended version)'

[100]: Hegglin (2021), 'Counterexamples for a Rust Verifier'

[193]: Schmid (2018), 'Abstract Read Permission Support for an Automatic Python Verifier'

[224]: Weber (2017), 'Automatic Verification of Closures and Lambda-Functions in Python'

[66]: Eilers et al. (2021), 'Product Programs in the Wild: Retrofitting Program Verifiers to Check Information Flow Security'

[148]: Meier (2018), 'Verification of information flow security for Python programs'

2.8. Conclusion

In this chapter, we have presented our approach for statically verifying programs written in Python 3. Our approach enables verification of a dynamic language with manageable specification and performance overhead compared to the verification of static languages, and is able to support most typical real-world code. On top of our approach for handling dynamic language features, we added the ability to specify and verify a number of complex program properties vital for proving the security of software systems, going beyond the abilities of most existing verification tools. We implemented our approach in the automated verifier Nagini, and showed that it is able to handle complex code from large code bases and prove complex properties in practice.

Modular Product Programs 3.

In this chapter, which is based on the ESOP 2018 paper "Modular Product Programs" [68] as well as the TOPLAS 2020 paper of the same name [69], we present our approach for modular verification of hyperproperties.

[68]: Eilers et al. (2018), 'Modular Product Programs'

[69]: Eilers et al. (2020), 'Modular Product Programs'

As explained in Chapter 1, the past decades have seen significant progress in (automated) reasoning about program behavior, in particular, in proving *trace properties* like functional correctness or termination. However, important security-related program properties cannot be expressed as properties of individual traces; these so-called *hyperproperties* relate different executions of the same program.

An important such property is *non-interference*, which expresses that some subset S of a program's inputs cannot influence the values of some subset O of the program's outputs. It is a property of *pairs* of program executions, since it states that, for any single execution, any second execution with potentially modified values for the inputs in S (but identical values for all other inputs) will end up with identical values for all outputs in O.

Non-interference can be used to model *confidentiality* in the form of *secure information flow*, by proving that secret (high-sensitivity) program inputs do not influence the values of public (low-sensitivity) outputs (i.e., the public outputs leak no information about the secret inputs). It can also be used to model *integrity*, by proving that attacker-controlled (low-integrity) inputs cannot influence the values of important variables which must be outside the control of the attacker and are therefore labeled as high-integrity.

As pointed out in Chapter 1, an important attribute of reasoning techniques about programs is *modularity*, which is vital for scalability, and to verify libraries without knowing all of their clients. A technique is modular if it allows reasoning about parts of a program in isolation, e.g., verifying each method separately and using only the *specifications* of other methods.

Fully modular reasoning about hyperproperties thus requires the ability to formulate *relational* specifications, which relate different executions of a method, and to apply those specifications where the method is called. As an example (in the setting of information flow security), the statement

```
if (l) { r := l } else { r:= f(l, h) }
```

where `l` is a low-sensitivity input, `h` is a high-sensitivity input, and `r` is a low-sensitivity output, can be proved to fulfill non-interference if `f`'s relational specification guarantees that its result, when given one low and one high input, is also low, i.e., that `f` itself also fulfills non-interference.

[30]: Benton (2004), 'Simple relational correctness proofs for static analyses and program transformations'
[206]: Sousa et al. (2016), 'Cartesian Hoare logic for verifying k-safety properties'
[231]: Yang (2007), 'Relational separation logic'

Relational program logics [30, 206, 231] allow directly proving general hyperproperties. However, automating relational logics is difficult and requires building dedicated tools. Alternatively, self-composition [25] and product programs [23, 24] reduce a hyperproperty to an ordinary trace property, thus making it possible to use off-the-shelf program verifiers for proving hyperproperties. Both approaches construct a new program that combines the behaviors of multiple runs of the original

[25]: Barthe et al. (2011), 'Secure information flow by self-composition'

[23]: Barthe et al. (2011), 'Relational Verification Using Product Programs'
[24]: Barthe et al. (2013), 'Beyond 2-Safety: Asymmetric Product Programs for Relational Program Verification'

program. However, by the nature of their construction, neither approach supports modular verification based on relational specifications: Method calls in the original program will be duplicated, which means that there is no single program point at which a relational specification can be applied. For the aforementioned example, self-composition yields the following program:

```
if (l) { r := l } else { r := f(l, h) };
if (l') { r' := l' } else { r' := f(l', h') }
```

Non-interference can now be verified by proving the trace property that identical values for l and l' in the initial state (with potentially different values for h and h') imply identical values for r and r' in the final state. However, such a proof cannot make use of a relational specification for method f (expressing that f does not leak any information about its high input into its output). Such a specification relates several executions of f, whereas each call in the self-composition belongs to a single execution. Instead, verification requires a *precise functional specification* of f, which *exactly* determines its result value in terms of the input. Verifying such precise functional specifications increases the verification effort and is at odds with data abstraction (for instance, a collection might not want to promise the exact iteration order); inferring them is beyond the state of the art for most methods [211]. Existing product programs allow aligning or combining some statements and can thereby lift this requirement in some cases, but this requires manual effort during the construction, depends on the used specifications, and does not solve the problem in general. Some techniques for proving equivalence between different programs use product constructions that enable using relational specifications between different functions. However, they cannot exploit the inherent advantages resulting from combining a program with itself as opposed to an arbitrary other program [98, 119].

[211]: Terauchi et al. (2005), 'Secure Information Flow as a Safety Problem'

[98]: Hawblitzel et al. (2013), 'Towards Modularly Comparing Programs Using Automated Theorem Provers'
[119]: Lahiri et al. (2013), 'Differential assertion checking'

In this chapter, we present modular product programs, a novel kind of product programs that allows modular reasoning about hyperproperties. Modular product programs enable proving k-safety hyperproperties, i.e., hyperproperties that relate finite prefixes of k execution traces, for arbitrary values of k [46]. We achieve this via a transformation that, unlike existing products, does not duplicate loops or method calls, meaning that for any loop or call in the original program, there is exactly one statement in the k-product at which a relational specification can be applied. Like existing product programs, modular products can be reasoned about using off-the-shelf program verifiers.

[46]: Clarkson et al. (2010), 'Hyperproperties'

We demonstrate the expressiveness of modular product programs by applying them to prove non-interference, a 2-safety hyperproperty that can model both information flow security and integrity properties. Focusing on the former, we then show how modular products enable proving traditional non-interference using natural and concise information flow specifications, and how to extend our approach for proving the absence of timing or termination channels, and supporting declassification in an intuitive way.

To summarize, we make the following contributions:

- We introduce modular k-product programs, which enable modular proofs of arbitrary k-safety hyperproperties for sequential programs using off-the-shelf verifiers.

```
method main(people)
      returns (count)
{
  i := 0;
  count := 0;
  while (i < |people|) {
    current := people[i];
    f := is_smoker(current);
    count := count + f;
    i := i + 1;
  }
}
```

```
method is_smoker(person)
      returns (res)
{
  // first two bits of person encode
  // how frequently person smokes.
  smoking := person mod 4;
  if (smoking != 0) {
    res := 1;
  }else{
    res := 0;
  }
}
```

Figure 3.1.: Example program. The parameter people contains a sequence of integers that each encode attributes of a person; the main method counts the number of smokers in this sequence.

- We prove soundness and completeness of the modular product program transformation.
- We demonstrate the usefulness of modular product programs by applying them to non-interference, with support for complex properties like value-dependent classification [160], declassification, and preventing different kinds of side channels.
- We implement our product-based approach for verification of both secure information flow and arbitrary other safety hyperproperties in the Viper verification infrastructure [159].
- We show that our tool can automatically prove non-interference and other hyperproperties of challenging examples.

[160]: Murray et al. (2018), 'COVERN: A Logic for Compositional Verification of Information Flow Control'

[159]: Müller et al. (2016), 'Viper: A Verification Infrastructure for Permission-Based Reasoning'

The structure of this chapter is as follows: After giving an informal overview of our approach in Sec. 3.1 and introducing our programming and assertion language in Sec. 3.2, we formally define modular product programs in Sec. 3.3. We prove the soundness and completeness of the approach in Sec. 3.4. Sec. 3.5 demonstrates how to apply modular products for proving non-interference. We describe and evaluate our implementation in Sec. 3.6, discuss related work in Sec. 3.7, and conclude in Sec. 3.8.

3.1. Overview

In this section, we will illustrate the core concepts behind modular k-products on an example program. We will first show how modular products are constructed, and subsequently demonstrate how they allow using relational specifications to modularly prove hyperproperties.

3.1.1. Relational Specifications

Consider the example program in Figure 3.1, which counts the number of smokers in a sequence of people. Now assume we want to prove a very simple hyperproperty, namely, that the program is *deterministic*: its output state is completely determined by its input arguments. Determinism is a 2-safety hyperproperty which states that, for two terminating executions of the program with identical inputs, the outputs will be the same. This hyperproperty can be expressed by stating that when main terminates from two states that fulfill the *relational* (as opposed to *unary*) precondition $\mathsf{people}^{(1)} = \mathsf{people}^{(2)}$, the resulting two states fulfill the relational postcondition $\mathsf{count}^{(1)} = \mathsf{count}^{(2)}$, where $x^{(i)}$ refers to the value of the variable x in the ith execution. We write this relational specification of main as $\{\!\{\text{people}^{(1)} = \text{people}^{(2)}\}\!\}\text{main}\{\!\{\text{count}^{(1)} = \text{count}^{(2)}\}\!\}$.

Intuitively, it is possible to prove this specification by giving is_smoker a precise functional specification like $\{true\}$is_smoker$\{$res = (person mod 4)!$\}$, meaning that is_smoker can be invoked in any state fulfilling the precondition *true* and that res =(person mod 4)! will hold if it returns. From this specification and an appropriate loop invariant, main can be shown to be deterministic. However, this specification is unnecessarily strong. For proving determinism, it is irrelevant what exactly the final value of count is; it is only important that it is uniquely determined by the method's inputs. Proving hyperproperties using only unary specifications, however, critically depends on having exact specifications for every value returned by a called method, as well as all heap locations modified by it. Not only are such specifications difficult to infer and cumbersome to provide manually; this requirement also fundamentally removes the option of underspecifying program behavior, which is often desirable in practice. Because of these limitations, verification techniques that require precise functional specifications for proving hyperproperties often do not work well in practice, as observed by Terauchi and Aiken for the case of self-composition [211].

[211]: Terauchi et al. (2005), 'Secure Information Flow as a Safety Problem'

Proving determinism of the example program becomes much simpler if we are able to reason about two program executions at once. If both runs start with identical values for people then they will have identical values for people, i, and count when they reach the loop. Since the loop guard only depends on i and people, it will either be true for both executions or false for both. Assuming that is_smoker behaves deterministically, all three variables will again be equal in both executions at the end of the loop body. This means that the program establishes and preserves the relational loop invariant that people, i, and count have identical values in both executions, from which we can deduce the desired relational postcondition. Our modular product programs enable this modular and intuitive reasoning, as we explain next.

3.1.2. Product Construction Idea

Like other product programs, our modular k-product programs multiply the state space of the original program by creating k renamed versions of all original variables. However, unlike other product programs, they do *not* duplicate control structures like loops or method calls, while still allowing different executions to take different paths through the program.

Modular product programs achieve this as follows: The set of transitions made by the execution of a product is the union of the transitions made by the executions of the original program it represents. This means that if two executions of an if-then-else statement execute different branches, an execution of the product will execute the corresponding versions of *both* branches; however, it will be aware of the fact that each branch is taken by only one of the original executions, and the transformation of the statements *inside* each branch will ensure that the state of the other execution is not modified by executing it.

```
method main(p1, p2, people1, people2)
    returns (count1, count2)
{
  if (p1) { i1 := 0; }
  if (p2) { i2 := 0; }
  if (p1) { count1 := 0; }
  if (p2) { count2 := 0; }
  while ((p1 && i1 < |people1|) ||
        (p2 && i2 < |people2|)) {
    l1 := p1 && i1 < |people1|;
    l2 := p2 && i2 < |people2|;
    if (l1) { current1 := people1[i1]; }
    if (l2) { current2 := people2[i2]; }
    if (l1 || l2) {
      t1, t2 := is_smoker(l1, l2,
                current1, current2);
    }
    if (l1) { f1 := t1; }
    if (l2) { f2 := t2; }
    if (l1) { count1 := count1 + f1; }
    if (l2) { count2 := count2 + f2; }
    if (l1) { i1 := i1 + 1; }
    if (l2) { i2 := i2 + 1; }
  }
}
```

```
method is_smoker(p1, p2,
                 person1,
                 person2)
         returns (res1, res2)
{
  if (p1) {
    smoking1 := person1 mod 4;
  }
  if (p2) {
    smoking2 := person2 mod 4;
  }
  t1 := p1 && (smoking1 != 0);
  t2 := p2 && (smoking2 != 0);
  f1 := p1 && !(smoking1 != 0);
  f2 := p2 && !(smoking2 != 0);
  if (t1) { res1 := 1; }
  if (t2) { res2 := 1; }
  if (f1) { res1 := 0; }
  if (f2) { res2 := 0; }
}
```

Figure 3.2.: Modular 2-product of the program in Fig. 3.1 (slightly simplified). Parameters, local variables, and simple statements like assignments have been duplicated, but control flow statements have not. Instead, they are expressed by creating new sets of activation variables, and simple statements are conditional on their values. The initial activation variables p1 and p2 of the main method can be thought of as *true*.

For this purpose, modular product programs use boolean *activation variables* that store, for each execution, whether it is currently active. All activation variables are initially true. For every statement that directly changes the program state, the product performs the state change for all active executions. Control structures update which executions are active (for instance based on the loop condition) and pass this information down (into the branches of a conditional, the body of a loop, or the callee of a method call) to the level of atomic statements[1]. This representation avoids duplicating these control structures.

Figure 3.2 shows the modular 2-product of the program in Figure 3.1. Consider first the main method. Its parameters have been duplicated, there are now two copies of all variables, one for each execution. This is analogous to self-composition or existing product programs. In addition, the transformed method has two boolean parameters p1 and p2; these variables are the initial activation variables of the method. Since main is the entry point of the program, the initial activation variables can be assumed to be true (not shown).

The product will first initialize i1 and i2 to zero, like it does with i in the original program, and analogously for count1 and count2. The loop in the original program has been transformed to a single loop in the product. Its condition is true if the original loop condition is true for any active execution. This means that the loop will iterate as long as at least one execution of the original program would. Inside the loop body, the fresh activation variables l1 and l2 represent whether the corresponding executions would execute the loop body. That is, for each execution, the respective activation variable will be true if the previous activation variable (p1 or p2, respectively) is true, meaning that this execution actually reaches the loop, and the loop guard is true for that execution. All statements in the loop body are then transformed using these new activation variables. Consequently, the loop will keep iterating while at least one execution executes the loop, but as soon as the loop guard is false for any execution, its activation variable will be false and the loop

[1] In this way, the activation variables are similar to path conditions used in symbolic execution.

body will have no effect on the variables of this execution.

Conceptually, method calls are handled very similarly to loops. For the call to is_smoker in the original program, only a single call is created in the product. This call is executed if at least one activation variable is true, i.e., if at least one execution would perform the call in the original program; without the conditional, the product would perform calls that do not correspond to anything in the original executions, which could lead to non-termination. In addition to the (duplicated) arguments of the original call, the current activation variables are passed to the called method. In the transformed version of is_smoker, all statements are then made conditional on those activation variables. Therefore, like with loops, a call in the product will be performed if at least one execution would perform it in the original program, but it will have no effect on the state of the executions that are not active when the call is made.

The transformed version of is_smoker shows how conditionals are handled. We introduce four fresh activation variables t1, t2, f1, and f2, two for each execution. The first pair encodes whether the then-branch should be executed by either of the two executions; the second encodes the same for the else-branch. These activation variables are then used to transform the branches. Consequently, neither branch will have an effect for inactive executions, and exactly one branch has an effect for each active execution.

To summarize, our activation variables ensure that the sequence of state-changing statements executed by each execution is the same in the product and the original program. We achieve this without duplicating control structures or imposing restrictions on the control flow.

3.1.3. Interpretation of Relational Specifications

Since modular product programs do not duplicate calls, they provide a simple way of interpreting relational method specifications: A precondition that requires some relation between the states of different executions is required to hold before the call, and a relational postcondition can be assumed to hold afterwards. However, a relational method specification should only apply if all the executions it refers to actually perform the method call. If a call is performed by some of those executions but not all, the relational specifications are not meaningful, and thus cannot be required to hold. To encode this intuition, we transform every relational pre- or postcondition $\hat{Q}$ of the original program that refers to some subset E of all executions into an implication $(\bigwedge_{i \in E} \mathsf{p}_i) \Rightarrow \hat{Q}$. As a result, relational specifications will be trivially true if at least one execution it refers to is not active at the call site.

In our example, we give is_smoker the relational specification $\{\!\!\{\mathsf{person}^{(1)} = \mathsf{person}^{(2)}\}\!\!\}\mathsf{is\_smoker}\{\!\!\{\mathsf{res}^{(1)} = \mathsf{res}^{(2)}\}\!\!\}$, which expresses determinism. This specification will be transformed into a unary specification of the product program: $\{\mathsf{p1} \wedge \mathsf{p2} \Rightarrow \mathsf{person1} = \mathsf{person2}\}\mathsf{is\_smoker}\{\mathsf{p1} \wedge \mathsf{p2} \Rightarrow \mathsf{res1} = \mathsf{res2}\}$.

Assume for the moment that is_smoker also has a unary precondition $\mathsf{person} \geq 0$. Such a specification should hold for *every* call, and therefore for every active execution, even if other executions are inactive. Therefore,

$$
\begin{array}{llll}
(\textit{Methods}) & \mathit{Mtd} & ::= & \texttt{method}\ m(\overline{x})\ \texttt{returns}\ (\overline{y})\{\hat{s}\} \\
(\textit{FStatements}) & \dot{s} & ::= & \texttt{skip} \mid \texttt{error} \mid \texttt{magic} \\
(\textit{RTStatements}) & \hat{s} & ::= & x{:=}e \mid \hat{s};\hat{s} \mid \texttt{if}\ (e)\ \{\hat{s}\}\ \texttt{else}\ \{\hat{s}\} \\
 & & & \mid \texttt{while}\ (e)\ \{\hat{s}\} \mid \overline{x}{:=}\ m(\overline{e}) \\
 & & & \mid \texttt{assert}\ e \mid \texttt{assume}\ e \mid \texttt{havoc}\ x \\
 & & & \mid \overline{x}{:=}\texttt{frame}_{\overline{y}}(\hat{s}, \sigma) \mid \dot{s} \\
(\textit{Expressions}) & e & ::= & c \mid x \mid e \wedge e \mid e \vee e \mid \neg e \\
 & & & e \oplus e \text{ where } c \in \mathbb{Z}, \oplus \in \{+, -, \times, =, <\} \\
(\textit{Assertions}) & P & ::= & P \wedge P \mid P \Rightarrow P \mid \forall x.\, P \mid e \\
(\textit{RelExpressions}) & \hat{e} & ::= & c \mid x^{(i)} \mid \hat{e} \oplus \hat{e} \mid \hat{e} \wedge \hat{e} \mid \hat{e} \vee \hat{e} \mid \neg\hat{e} \\
(\textit{RelAssertions}) & \hat{P} & ::= & \hat{P} \wedge \hat{P} \mid \hat{P} \Rightarrow \hat{P} \mid \forall x^{(1)}, \ldots, x^{(k)}.\, \hat{P} \mid \hat{e} \\
(\textit{MixAssertions}) & \check{P} & ::= & P \wedge \hat{P}
\end{array}
$$

Figure 3.3.: Supported programming language. Relational expressions and assertions generalize unary ones to refer to the state of multiple different executions. We distinguish two kinds of statements: Runtime statements (RTStatements) $\hat{s}$ denote all statements that can occur in a trace, whereas final statements (FStatements) $\dot{s}$ denote finished computations. We write $\overline{x}$ to denote a vector of variables.

its interpretation in the product program is $(\mathsf{p1} \Rightarrow \mathsf{person1} \geq 0) \wedge (\mathsf{p2} \Rightarrow \mathsf{person2} \geq 0)$. The translation of other unary assertions is analogous.

Note that it is possible (and useful) to give a method both a relational and a unary specification; in the product this is encoded by simply conjoining the transformed versions of the unary and the relational assertions.

3.1.4. Product Program Verification

We can now prove determinism of our example using the product program. Verifying is_smoker is simple. For main, we want to prove the transformed specification $\{(\mathsf{p1} \wedge \mathsf{p2} \Rightarrow \mathsf{people1} = \mathsf{people2})\}\mathsf{main}\{(\mathsf{p1} \wedge \mathsf{p2} \Rightarrow \mathsf{count1} = \mathsf{count2})\}$. We use the relational loop invariant $\mathsf{i}^{(1)} = \mathsf{i}^{(2)} \wedge \mathsf{count}^{(1)} = \mathsf{count}^{(2)} \wedge \mathsf{people}^{(1)} = \mathsf{people}^{(2)}$, encoded as $\mathsf{p1} \wedge \mathsf{p2} \Rightarrow \mathsf{i1} = \mathsf{i2} \wedge \mathsf{count1} = \mathsf{count2} \wedge \mathsf{people1} = \mathsf{people2}$. The loop invariant holds trivially if p1 or p2 is false. Otherwise, it ensures $\mathsf{l1} = \mathsf{l2}$ and $\mathsf{current1} = \mathsf{current2}$. Using the specification of is_smoker, we obtain $\mathsf{t1} = \mathsf{t2}$, which implies that the loop invariant is preserved. The loop invariant implies the postcondition.

3.2. Preliminaries

Fig. 3.3 shows the language we use to define modular product programs. x ranges over the set of local integer variable names Var.

Program configurations have the form $\langle \hat{s}, \sigma \rangle$, where a store $\sigma \in \Sigma$ is a partial function mapping variable names to integer values. We use $\dot{s}$ to range over the *final* statements `skip`, `error`, and `magic`. A configuration is final if it has the form $\langle \dot{s}, \sigma \rangle$. Executions ending in `skip` are regular executions, whereas those ending with `error` represent failing executions caused by failing `assert` statements. Similarly, an execution halts in a `magic` state if an `assume` statement assumes something that does not hold in the current state (and the execution is therefore assumed, magically, not to exist).

Our language supports non-determinism via `havoc` statements that assign an arbitrary value to a variable; all other statements are deterministic. We distinguish between runtime statements $\hat{s}$ and program statements s (not depicted in Fig. 3.3): Program statements are runtime statements that do

$$\frac{e \Downarrow_\sigma v}{\langle x{:=}e, \sigma\rangle \rightarrow \langle \texttt{skip}, \sigma[x \mapsto v]\rangle}\ (\textsc{Assign})$$

$$\frac{}{\langle \texttt{havoc}\ x, \sigma\rangle \rightarrow \langle \texttt{skip}, \sigma[x \mapsto v]\rangle}\ (\textsc{Havoc})$$

$$\frac{e \Downarrow_\sigma \top}{\langle \texttt{assert}\ e, \sigma\rangle \rightarrow \langle \texttt{skip}, \sigma\rangle}\ (\textsc{Assert}1) \qquad \frac{e \Downarrow_\sigma \bot}{\langle \texttt{assert}\ e, \sigma\rangle \rightarrow \langle \texttt{error}, \sigma\rangle}\ (\textsc{Assert}2)$$

$$\frac{e \Downarrow_\sigma \top}{\langle \texttt{assume}\ e, \sigma\rangle \rightarrow \langle \texttt{skip}, \sigma\rangle}\ (\textsc{Assume}1) \qquad \frac{e \Downarrow_\sigma \bot}{\langle \texttt{assume}\ e, \sigma\rangle \rightarrow \langle \texttt{magic}, \sigma\rangle}\ (\textsc{Assume}2)$$

$$\frac{e \Downarrow_\sigma \top}{\langle \texttt{if}\ (e)\ \{\hat{s}_1\}\ \texttt{else}\ \{\hat{s}_2\}, \sigma\rangle \rightarrow \langle \hat{s}_1, \sigma\rangle}\ (\textsc{Cond}1)$$

$$\frac{e \Downarrow_\sigma \bot}{\langle \texttt{if}\ (e)\ \{\hat{s}_1\}\ \texttt{else}\ \{\hat{s}_2\}, \sigma\rangle \rightarrow \langle \hat{s}_2, \sigma\rangle}\ (\textsc{Cond}2)$$

$$\frac{\langle \hat{s}_1, \sigma\rangle \rightarrow \langle \hat{s}'_1, \sigma'\rangle}{\langle \hat{s}_1; \hat{s}_2, \sigma\rangle \rightarrow \langle \hat{s}'_1; \hat{s}_2, \sigma'\rangle}\ (\textsc{Seq}1) \qquad \frac{}{\langle \texttt{skip}; \hat{s}_2, \sigma\rangle \rightarrow \langle \hat{s}_2, \sigma\rangle}\ (\textsc{Seq}2)$$

$$\frac{}{\langle \texttt{error}; \hat{s}_2, \sigma\rangle \rightarrow \langle \texttt{error}, \sigma\rangle}\ (\textsc{Seq}3) \qquad \frac{}{\langle \texttt{magic}; \hat{s}_2, \sigma\rangle \rightarrow \langle \texttt{magic}, \sigma\rangle}\ (\textsc{Seq}4)$$

$$\frac{e \Downarrow_\sigma \top}{\langle \texttt{while}\ (e)\ \{\hat{s}_1\}, \sigma\rangle \rightarrow \langle \hat{s}_1; \texttt{while}\ (e)\ \{\hat{s}_1\}, \sigma\rangle}\ (\textsc{Whl}1)$$

$$\frac{e \Downarrow_\sigma \bot}{\langle \texttt{while}\ (e)\ \{\hat{s}_1\}, \sigma\rangle \rightarrow \langle \texttt{skip}, \sigma\rangle}\ (\textsc{Whl}2)$$

$$\frac{\Phi(f) = ([p_1, \dots, p_n], [r_1, \dots, r_m], s_f) \qquad e_1 \Downarrow_\sigma v_1 \qquad \dots \qquad e_n \Downarrow_\sigma v_n}{\langle \overline{x}{:=}\ f(e_1, \dots, e_n), \sigma\rangle \rightarrow \langle \overline{x}{:=}\texttt{frame}_{\overline{r}}(s_f, [][p_1 \mapsto v_1] \dots [p_n \mapsto v_n]), \sigma\rangle}\ (\textsc{Call})$$

$$\frac{\langle \hat{s}_f, \sigma_f\rangle \rightarrow \langle \hat{s}'_f, \sigma'_f\rangle}{\langle \overline{x}{:=}\texttt{frame}_{\overline{r}}(\hat{s}_f, \sigma_f), \sigma\rangle \rightarrow \langle \overline{x}{:=}\texttt{frame}_{\overline{r}}(\hat{s}'_f, \sigma'_f), \sigma\rangle}\ (\textsc{Frame}1)$$

$$\frac{r_1 \Downarrow_{\sigma_f} v_1 \qquad \dots \qquad r_m \Downarrow_{\sigma_f} v_m}{\langle x_1, \dots, x_m{:=}\texttt{frame}_{r_1, \dots, r_m}(\texttt{skip}, \sigma_f), \sigma\rangle \rightarrow \langle \texttt{skip}, \sigma[x_1 \mapsto v_1] \dots [x_m \mapsto v_m]\rangle}\ (\textsc{Frame}2)$$

$$\frac{}{\langle \overline{x}{:=}\texttt{frame}_{\overline{r}}(\texttt{error}, \sigma_f), \sigma\rangle \rightarrow \langle \texttt{error}, \sigma\rangle}\ (\textsc{Frame}3)$$

$$\frac{}{\langle \overline{x}{:=}\texttt{frame}_{\overline{r}}(\texttt{magic}, \sigma_f), \sigma\rangle \rightarrow \langle \texttt{magic}, \sigma\rangle}\ (\textsc{Frame}4)$$

Figure 3.4.: Operational semantics. We denote the empty store as [].

not syntactically contain `error`, `magic`, or `frame` statements. Programs contain only program statements; `error`, `magic`, and `frame` can occur only in the process of executing a program statement. `frame` statements are used to model stack-like behaviors for method calls: $\overline{x}{:=}\texttt{frame}_{\overline{y}}(\hat{s}, \sigma)$ executes the statement $\hat{s}$ in the framed store σ and, if it reaches a final configuration, returns the results to the original store by assigning the value of y_i in the framed store to the target variable x_i in the original store.

The small-step transition relation for program configurations has the form $\langle \hat{s}, \sigma \rangle \longrightarrow \langle \hat{s}', \sigma' \rangle$ and is defined in Fig. 3.4. We assume as given a method context Φ that maps method names to triples $(\overline{q}, \overline{r}, s_m)$, where $\overline{q}$ is a vector of input parameter variables, $\overline{r}$ is a vector of return parameter variables, and s_m is the body of the method.

We denote that expression e evaluates to value v in store σ by $e \Downarrow_\sigma v$. The inference rules for expression evaluation are standard and therefore omitted. Expression evaluation is deterministic and total, and variable names not present in the store evaluate to some default value. Programs cannot get stuck in non-final configurations, but they can diverge. All variables and expressions in our language have type integer. When evaluating conditions, any non-zero value is interpreted as true. For convenience, we write $\top$ and $\bot$ to mean any arbitrary non-zero value and zero, respectively; that is, $\top$ is not one fixed non-zero value but is used as a stand-in for any number of potentially different values. As an example, we write $e_1 \Downarrow_\sigma \top \wedge e_2 \Downarrow_\sigma \top$ as a shorthand for $\exists v_1, v_2.\, e_1 \Downarrow_\sigma v_1 \wedge e_2 \Downarrow_\sigma v_2 \wedge v_1 \neq 0 \wedge v_2 \neq 0$.

The Hoare triple $\models \{P\}s\{Q\}$ denotes that statement s, when executed in a store fulfilling the unary assertion P, will not fail, and if the execution terminates regularly, the resulting store will fulfill the unary assertion Q. Formally, $\models \{P\}s\{Q\}$ if and only if for all $\sigma, \sigma', \dot{s}$ s.t. $\sigma \models P$ and $\langle s, \sigma \rangle \longrightarrow^* \langle \dot{s}, \sigma' \rangle$, we either have $\dot{s} = \texttt{magic}$ or $(\dot{s} = \texttt{skip} \wedge \sigma' \models Q)$.

In addition to standard unary expressions and assertions, we define relational expressions and assertions. They differ from normal expressions and assertions in that they contain parameterized variable references of the form $x^{(i)}$ and are evaluated over a tuple of stores instead of a single one. A relational expression is k-relational if for all contained variable references $x^{(i)}$, we have that $1 \leq i \leq k$. The definition of k-relational assertions is analogous. The value of a variable reference $x^{(i)}$ with $1 \leq i \leq k$ in a tuple of stores $(\sigma_1, \ldots, \sigma_k)$ is $\sigma_i(x)$; the evaluation of arbitrary relational expressions and the validity of relational assertions $(\sigma_1, \ldots, \sigma_k) \models \hat{P}$ are defined accordingly.

Definition 3.2.1 *A k-relational Hoare triple $\{\!\{\hat{P}\}\!\}s\{\!\{\hat{Q}\}\!\}_k$ holds iff $\hat{P}$ and $\hat{Q}$ are k-relational assertions, and for all $\sigma_1, \ldots, \sigma_k, \sigma'_1, \ldots, \sigma'_k$, if $(\sigma_1, \ldots, \sigma_k) \models \hat{P}$ and $\forall i \in \{1, \ldots, k\}.\, \langle s, \sigma_i \rangle \longrightarrow^* \langle \texttt{skip}, \sigma'_i \rangle$, then $(\sigma'_1, \ldots, \sigma'_k) \models \hat{Q}$.*

Note that our k-relational Hoare triples allow executions to fail, unlike their unary counterparts. We write $\{\!\{\hat{P}\}\!\}s\{\!\{\hat{Q}\}\!\}$ for the most common case $\{\!\{\hat{P}\}\!\}s\{\!\{\hat{Q}\}\!\}_2$.

Mixed assertions combine unary and relational assertions and hold if the unary parts hold for every single store, and the relational parts hold for the tuple of stores as a whole. Formally, we define the validity of a mixed assertion $\check{P} = P \land \hat{P}$, where $\hat{P}$ is a k-relational assertion, as $(\sigma_1, \ldots, \sigma_k) \vDash \check{P} \Leftrightarrow (\forall i \in \{1, \ldots, k\}.\, \sigma_i \vDash P) \land (\sigma_1, \ldots, \sigma_k) \vDash \hat{P}$.

3.3. Modular k-Product Program Definition

In this section, we define the construction of modular products for arbitrary k. We will subsequently define the transformation of both relational and unary specifications to modular products.

3.3.1. Product Construction

Assume as given an injective function $\phi : (\textsc{Var}, \mathbb{N}) \longrightarrow \textsc{Var}$ that renames variables for different executions. We write $e^{(i)}$ for the renaming of expression e for execution i. We write $\mathit{fresh}(x_1, x_2, \ldots)$ to denote that the variable names $x_1, x_2, \ldots$ are fresh names that do not occur in the program and have not yet been used during the transformation. $\mathring{e}$ is used to abbreviate $e^{(1)}, \ldots, e^{(k)}$.

We denote the modular k-product of a program statement s that is parameterized by the activation variables $p^{(1)}, \ldots, p^{(k)}$ as $[\![s]\!]_k^{\mathring{p}}$. The product construction for methods is defined as

$$\begin{aligned} &[\![\textsf{\textbf{method}}\ m(x_1, \ldots, x_m)\ \textsf{\textbf{returns}}\ (y_1, \ldots, y_n)\{s\}]\!]_k \\ =\ &\textsf{\textbf{method}}\ m(\mathring{p}, \mathring{x}_1, \ldots, \mathring{x}_m)\ \textsf{\textbf{returns}}\ (\mathring{y}_1, \ldots, \mathring{y}_n)\{[\![s]\!]_k^{\mathring{p}}\} \end{aligned}$$

for fresh variable names $\mathring{p}$.

Figure 3.5 shows the product construction rules for statements, which generalize the transformation explained in Sec. 3.1. We abbreviate $\texttt{if}\,(e)\,\{s\}\,\texttt{else}\,\{\texttt{skip}\}$ as $\texttt{if}\,(e)\,\{s\}$, and use $\bigodot_{i=1}^{k} s_i$ as a shorthand for the sequential composition of k statements $s_1; \ldots; s_k$.

The core principle behind our encoding is that statements that directly change the state are duplicated for each execution and made conditional under the respective activation variables, whereas control statements are not duplicated and instead manipulate the activation variables to pass activation information to their sub-statements. This enables us to assert or assume relational assertions before and after any statement from the original program. The only state-changing statements in our language, variable assignments and havocs, are therefore transformed to a sequence of conditional assignments or havocs, one for each execution. Each one is executed only if the respective execution is currently active. Similarly, **assert** and **assume** statements should only check and assume that an assertion is true for active executions, and are therefore treated in the same way.

Duplicating conditionals would also duplicate the calls and loops in their branches. To avoid that, modular products eliminate top-level conditionals; instead, new activation variables are created and assigned the values of the current activation variables conjoined with the guard for each branch. The branches are then sequentially executed based on their respective activation variables.

A while loop is transformed to a single while loop in the product program that iterates as long as the loop guard is true for *any* active execution. Inside the loop, fresh activation variables indicate whether an execution reaches the loop and its loop condition is true. In particular, if the number of iterations of a loop differs between executions, there will be iterations where the fresh activation variables will be false for the executions that no longer iterate, but true for the others. The loop body will then modify the state of an execution only if its activation variable is true. The resulting construct affects the program state in the same way as a self-composition of the original loop would, but the fact that our product contains only a single loop enables us to use relational loop invariants instead of full functional specifications. In particular, we do not require that loops iterate the same number of times for all executions in order to use relational loop invariants; it is sufficient that the iterations that are performed by only some of the executions preserve the relational invariant.

Once the loop condition is false for all active executions, the loop in the product will no longer be executed. As a result, the loop in the product is guaranteed to terminate if the loop execution terminates in all executions of the original program.

For method calls, it is crucial that the product contains a single call for every call in the original program, in order to be able to apply relational specifications at the call site. As explained before, initial activation parameters are added to every method declaration, and all parameters are duplicated k times. method calls are therefore transformed such that the values of the current activation variables are passed, and all arguments are passed once for each execution. The return values are stored in temporary variables and subsequently assigned to the actual target variables only for those executions that actually execute the call, so that for all other executions, the target variables are not affected.

The transformation wraps the call in a conditional so that it is performed only if at least one execution is active. This prevents the transformation from introducing infinite recursion that is not present in the original program.

Note that for an inactive execution i, arbitrary argument values are passed in method calls, since the passed variables ${a_j}^{(i)}$ are not initialized. This is unproblematic because these values will not be used by the method. It is important to not evaluate ${e_j}^{(i)}$ for inactive executions, since this could lead to false alarms for languages where expression evaluation can fail. Note that expression evaluation in our language is side-effect free; for a language with effectful expression evaluation, the product construction would have to be adapted slightly. In particular, the condition expressions in loops and conditionals would have to be evaluated exactly once per execution of the loop or conditional, which could easily be achieved by assigning them to additional auxiliary variables before using them in the product.

3.3.2. Transformation of Assertions

We now define how to transform unary and relational assertions for use in a modular product.

Unary assertions such as ordinary method preconditions describe state properties that should hold for every single execution. When checking or assuming that a unary assertion holds at a specific point in the program, we need to take into account that it only makes sense to do so for executions that actually reach that program point. We can express this by making the assertion conditional on the activation variable of the respective execution; as a result, any unary assertion holds trivially for all inactive executions.

A k-relational assertion, on the other hand, describes the relation between the states of multiple executions executions. Typically, to prove a k-relational property, assertions will relate all k executions; however, some intermediate assertions (e.g., loop invariants or postconditions of auxiliary methods) may need to relate some subset of the executions to summarize their relative behavior in the case where other executions do not reach a program point. We say that an execution is *relevant* w.r.t. a relational assertion if a variable from this execution is syntactically referenced in the assertion. Checking or assuming a relational assertion at some point is meaningful only if *all* relevant executions actually reach that point. This can be expressed by making relational assertions conditional on the conjunction of the current activation variables of all relevant executions. If at least one relevant execution does not reach the assertion, it holds trivially.

Assertions can be transformed for use in a k-product as follows:

- The transformation $\lfloor P \rfloor_k^{\mathring{p}}$ of a unary assertion P is $\bigwedge_{i=1}^{k} (p^{(i)} \Rightarrow P^{(i)})$.
- The transformation $\lfloor \hat{P} \rfloor_k^{\mathring{p}}$ of a k-relational assertion $\hat{P}$ with the activation variables $p^{(1)}, \ldots, p^{(k)}$ and relevant executions $R \subseteq \{1, \ldots, k\}$ is $(\bigwedge_{i \in R} p^{(i)}) \Rightarrow \hat{P}$.

Importantly, our approach allows using *mixed* assertions and specifications, which represent conjunctions of unary and relational assertions. For example, it is common to combine a unary precondition that ensures that a method will not raise an error with a relational postcondition that states that it is deterministic.

A mixed assertion $\check{R}$ of the form $P \wedge \hat{Q}$ means that the unary assertion P holds for every single execution, and if all relevant executions are currently active, the relational assertion $\hat{Q}$ holds as well. The transformation of mixed assertions is straightforward: $\lfloor \check{R} \rfloor_k^{\mathring{p}} = \lfloor P \rfloor_k^{\mathring{p}} \wedge \lfloor \hat{Q} \rfloor_k^{\mathring{p}}$.

3.3.3. Heap-Manipulating Programs

The approach outlined so far can easily be extended to programs that work on a mutable heap, assuming that object references are opaque, i.e., they cannot be inspected or used in arithmetic. There are two distinct ways of achieving this, each of which uses a different way of creating distinct state spaces for different executions:

1. One way is to duplicate allocation (new) statements for each execution and make them conditional like assignments, thereby creating a different object for each active execution. The renaming of a field dereference $e.f$ is then defined as $e^{(i)}.f$. As a result, the heap of a k-product will consist of k partitions that do not contain references to each other, and execution i will only ever interact with objects from its partition of the heap.
2. Alternatively, one can create only a single new statement for every new in the original program, and duplicate the fields each object has. The renaming of a field dereference $e.f$ is then defined as $e^{(i)}.f^{(i)}$. This approach has the effect that newly-created objects will be considered to be equal between different executions.
   Since references allocated at the same point might not actually be equal between two program executions, such an encoding can be useful because it gives a meaning to the idea that a reference might be low when verifying non-interference properties; however, it is sound only in programming languages where references are opaque, i.e., their concrete values cannot be inspected, but can only be checked for equality. Additionally, such an encoding weakens the verified property to an *object-sensitive* version of non-interference [15, 27, 29, 33, 99, 165], where values are considered low if they are equal modulo a consistent renaming of references. In our implementation, which we will discuss later, we choose this second encoding.

The verification of modular products of heap-manipulating programs does not depend on any specific way of achieving framing. Our implementation is based on implicit dynamic frames [199], but other approaches, in particular flavors of separation logic [180], are feasible as well, provided that methods can be specified in such a way that the caller knows the heap stays unmodified for all executions whose activation variables are false. With implicit dynamic frames, this is the case, because a permission assertion **acc**$(e.f)$ in a callee's pre- or postcondition will be transformed to an assertion $p^{(i)} \Rightarrow$ **acc**$(e.f^{(i)})$ for each execution i, meaning that the caller only gives up and gets back field permissions for active executions, and can therefore frame information about the fields of all inactive executions around the call.

[199]: Smans et al. (2012), 'Implicit dynamic frames'

[180]: Reynolds (2002), 'Separation Logic: A Logic for Shared Mutable Data Structures'

Since the handling of the heap is largely orthogonal to our main technique, we will not go into further detail here, but we do support heap-manipulating programs in our implementation.

3.3.4. Discussion

One obvious property of the product construction is that it introduces a high number of branching statements into the transformed programs, which can make it expensive to reason about them using some analysis and verification techniques. However, since the product construction enables modular reasoning about loops and method calls, verifiers have to reason only about small pieces of code at a time and can check different methods independently from one another. Moreover, the ability to use relational specifications tends to make specifications simpler (and therefore easier to prove) in many cases. Additionally, the specific structure of the generated program (e.g., many branches on the same condition, many identical statements on mirrored states) makes it very efficient to reason about the generated program using some standard techniques, as we will show in Sec. 3.6.

A limitation of our approach is that, while the product construction is complete in principle, relational specifications can only be used to relate the same loop or method call in multiple executions, but not different loops or calls. As an example, for the program `if` (e) $\{m()\}$ `else` $\{m()\}$, relational specifications for m cannot be used to relate the behavior of the call in the then-branch to the one in the else-branch. Note, however, that in this case, a full functional specification of m will still be sufficient to prove any relational property of the program; modular product programs are therefore never weaker or require more specification than self-composition. Some approaches from the related field of program equivalence are able to relate different calls and loops [98, 119]. However, this ability comes at a cost (e.g., losing the ability to reason about the resulting program using separation logic, or to perform standard static analyses on it), which we discuss in more detail in Sec. 3.7.

[98]: Hawblitzel et al. (2013), 'Towards Modularly Comparing Programs Using Automated Theorem Provers'
[119]: Lahiri et al. (2013), 'Differential assertion checking'

$$
\begin{array}{lcl}
\llbracket s_1; s_2 \rrbracket_k^{\mathring{p}} & = & \llbracket s_1 \rrbracket_k^{\mathring{p}}; \llbracket s_2 \rrbracket_k^{\mathring{p}} \\
\llbracket \texttt{skip} \rrbracket_k^{\mathring{p}} & = & \texttt{skip} \\
\llbracket x{:=}e \rrbracket_k^{\mathring{p}} & = & \bigodot_{i=1}^{k} \texttt{if}\ (p^{(i)})\ \{x^{(i)}{:=}e^{(i)}\} \\
\llbracket \texttt{assert}\ e \rrbracket_k^{\mathring{p}} & = & \bigodot_{i=1}^{k} \texttt{if}\ (p^{(i)})\ \{\texttt{assert}\ e^{(i)}\} \\
\llbracket \texttt{assume}\ e \rrbracket_k^{\mathring{p}} & = & \bigodot_{i=1}^{k} \texttt{if}\ (p^{(i)})\ \{\texttt{assume}\ e^{(i)}\} \\
\llbracket \texttt{havoc}\ x \rrbracket_k^{\mathring{p}} & = & \bigodot_{i=1}^{k} \texttt{if}\ (p^{(i)})\ \{\texttt{havoc}\ x^{(i)}\} \\
\llbracket \texttt{if}\ (e)\ \{s_1\}\ \texttt{else}\ \{s_2\} \rrbracket_k^{\mathring{p}} & = & \bigodot_{i=1}^{k} ({p_1}^{(i)}{:=}p^{(i)} \wedge e^{(i)}); \\
 & & \bigodot_{i=1}^{k} ({p_2}^{(i)}{:=}p^{(i)} \wedge \neg e^{(i)}); \\
 & & \llbracket s_1 \rrbracket_k^{\mathring{p}_1}; \llbracket s_2 \rrbracket_k^{\mathring{p}_2} \\
 & & \text{where} \\
 & & \mathit{fresh}(\mathring{p}_1) \wedge \mathit{fresh}(\mathring{p}_2) \\
\llbracket \texttt{while}\ (e)\ \{s\} \rrbracket_k^{\mathring{p}} & = & \quad \texttt{while}\ (\bigvee_{i=1}^{k} (p^{(i)} \wedge e^{(i)}))\ \{ \\
 & & \qquad \bigodot_{i=1}^{k} ({p_1}^{(i)}{:=}p^{(i)} \wedge e^{(i)}); \\
 & & \qquad \llbracket s \rrbracket_k^{\mathring{p}_1} \\
 & & \quad \} \\
 & & \text{where} \\
 & & \mathit{fresh}(\mathring{p}_1) \\
\llbracket x_1, \ldots, x_n{:=}\ f(e_1, \ldots, e_m) \rrbracket_k^{\mathring{p}} & = & \quad \texttt{if}\quad (\bigvee_{i=1}^{k} p^{(i)})\ \{ \\
 & & \qquad\quad \bigodot_{i=1}^{k} \texttt{if}\ (p^{(i)})\ \{\bigodot_{j=1}^{m} ({a_j}^{(i)}{:=}{e_j}^{(i)})\}; \\
 & & \qquad\quad \bar{t}{:=}\ f(p^{(1)}, \ldots, p^{(k)}, \bar{a}); \\
 & & \qquad\quad \bigodot_{i=1}^{k} \texttt{if}\ (p^{(i)})\ \{\bigodot_{j=1}^{n} ({x_j}^{(i)}{:=}{t_j}^{(i)})\} \\
 & & \quad \} \\
 & & \text{where} \\
 & & \mathit{fresh}(\mathring{a}_1, \ldots, \mathring{a}_m) \wedge \mathit{fresh}(\mathring{t}_1, \ldots, \mathring{t}_n) \\
 & & \bar{a} = [\mathring{a}_1, \ldots, \mathring{a}_m] \\
 & & \bar{t} = [\mathring{t}_1, \ldots, \mathring{t}_n]
\end{array}
$$

Figure 3.5.: Construction rules for statement products.

3.4. Soundness and Completeness

A product construction is sound if an execution of a k-product mirrors k separate executions of the original program such that a property of a product execution reflects a hyperproperty of the original program. Similarly, the construction is complete if for any hyperproperty of the original program, the corresponding property holds for the product.

In this section, we first prove, based on the operational semantics of our language, that every execution of a k-product is simulated by k executions of the original program. In particular, if the product of a statement is executed from a store that mirrors k original stores and terminates normally then executing the original statement from any of the k original stores will also terminate normally, and the final store of the product will mirror the final original stores. Subsequently, we show the reverse, namely that any set of k executions can be simulated by a product execution. Additionally, we show that the termination behavior of modular product programs mirrors that of the underlying original executions: If a set of executions terminates, the product execution that mirrors it is guaranteed to terminate as well; conversely, normally-terminating product executions correspond to sets of terminating executions. We discuss the remaining case (the product terminates abnormally) below. Finally, we show how properties of a product program relate to hyperproperties of the original program. In particular, we prove that if a transformed relational specification holds for a product program, the original relational specification holds for the original program (*soundness*), and, conversely, that if a relational specification holds for a program, the transformed specification holds for its product program (*completeness*).

3.4.1. Preliminaries

Throughout this section, we will denote statements, stores and method mappings of product programs by $\underline{s}$, $\underline{\sigma}$, and $\underline{\Phi}$, respectively, and statements, stores and mappings of the original (non-product) executions by s, σ, and Φ.

We first define what it means for the store of a product program to mirror the stores of multiple original executions. Stores in product executions contain renamed versions of the stores of the original program executions. By $\sigma \in_i \underline{\sigma}$ we denote that $\underline{\sigma}$ contains an i-renamed version of all variables in σ and no other i-renamed program variables, and the values of those variables agree in both stores.

Definition 3.4.1 *$\sigma \in_i \underline{\sigma}$ if and only if $(\forall x \in \textsc{PVar}.\, (x^{(i)} \in dom(\underline{\sigma}) \Leftrightarrow x \in dom(\sigma)) \land (x^{(i)} \in dom(\underline{\sigma}) \Rightarrow \sigma(x) = \underline{\sigma}(x^{(i)}))$, where $\textsc{PVar}$ is the set of variable names used in the original program, and $\textsc{RPVar}$ is the set of all renamed versions of all variables in $\textsc{PVar}$; i.e., $x \in \textsc{PVar} \Leftrightarrow (\forall i.\, x^{(i)} \in \textsc{RPVar}$.)*

We assume that all fresh variable names picked during the product construction are not in RPVar, so that this definition allows $\underline{\sigma}$ to contain i-renamed auxiliary or activation variables that have no counterpart in σ.

Similarly, we use the relation $\underline{\sigma} \preccurlyeq_i^V \underline{\sigma}'$, where V is a set of variable names, to say that $\underline{\sigma}'$ contains all variables in $\underline{\sigma}$ that belong to the i-th execution and maps them to the same values, except for the variables in V:

Definition 3.4.2 $\underline{\sigma} \preccurlyeq_i^V \underline{\sigma}'$ *if and only if* $\forall x.\, x^{(i)} \notin V \Rightarrow (x^{(i)} \in dom(\underline{\sigma}) \Rightarrow x^{(i)} \in dom(\underline{\sigma}') \wedge \underline{\sigma}(x^{(i)}) = \underline{\sigma}'(x^{(i)}))$*, where* $\underline{\sigma} \preccurlyeq^V \underline{\sigma}'$ *denotes that this is true for all executions, i.e.,* $\underline{\sigma} \preccurlyeq^V \underline{\sigma}' \Leftrightarrow \forall i \in \{1, \ldots, k\}.\, \underline{\sigma} \preccurlyeq_i^V \underline{\sigma}'$.

Note that both of this relation is reflexive.

We call $\mathit{freshvars}(\underline{s})$, where $\underline{s} = [\![s]\!]_k^{\mathring{p}}$ for some $k, \mathring{p}$, and s, the set of variables used as fresh variables in the construction of $\underline{s}$. For convenience, we abbreviate $\underline{\sigma} \preccurlyeq^{\mathit{freshvars}(\underline{s})} \underline{\sigma}'$ as $\underline{\sigma} \preccurlyeq^{\underline{s}} \underline{\sigma}'$ and $\underline{\sigma} \preccurlyeq_i^{\mathit{freshvars}(\underline{s})} \underline{\sigma}'$ as $\underline{\sigma} \preccurlyeq_i^{\underline{s}} \underline{\sigma}'$. Additionally, we denote $\underline{\sigma} \preccurlyeq^{\text{RPV}\textsc{ar} \cup \mathit{freshvars}(\underline{s})} \underline{\sigma}'$ as $\underline{\sigma} \precsim^{\underline{s}} \underline{\sigma}'$ and $\underline{\sigma} \preccurlyeq_i^{\text{RPV}\textsc{ar} \cup \mathit{freshvars}(\underline{s})} \underline{\sigma}'$ as $\underline{\sigma} \precsim_i^{\underline{s}} \underline{\sigma}'$.

Finally, we denote the free variables in a mixed assertion $\check{P}$ as $\mathit{fv}(\check{P})$.

We prove soundness and completeness under the assumption that all methods referenced by our product program have been transformed to their modular products. This means that for an original program with methods Φ, the product has methods $\underline{\Phi}$ such that

$$\forall f.\, \Phi(f) = ([q_1, \ldots, q_n], [r_1, \ldots, r_m], s) \Leftrightarrow \underline{\Phi}(f) = (\mathit{args}, \mathit{rets}, [\![s]\!]_k^{\mathring{p}})$$

where $\mathit{args} = \mathring{p}, \mathring{q}_1, \ldots, \mathring{q}_n$ and $\mathit{rets} = \mathring{r}_1, \ldots, \mathring{r}_m$, and the parameters $\mathring{p}$ corresponding to activation variables are not in RPV<span style="font-variant:small-caps">ar</span>. If this is the case, we say that $\mathit{match}(\Phi, \underline{\Phi})$.

We start by showing that our notion of mirrored stores has the intended effect, namely, that evaluating an expression in a store results in the same value as evaluating a renamed expression in a store that mirrors the original store:

Lemma 3.4.1 *If* $\sigma \in_i \underline{\sigma}$ *and* $\mathit{fv}(e) \subseteq P\textsc{Var}$ *then* $e^{(i)} \Downarrow_{\underline{\sigma}} v \Leftrightarrow e \Downarrow_{\sigma} v$.

Proof. By induction on the structure of e. For the case $e = x$, the proof follows trivially from the definition of $\in_i$; for all others, it is either immediate or follows from applying the induction hypothesis to the subexpressions. □

3.4.2. Properties of Product Executions

We can now establish properties of modular product programs on the level of the operational semantics.

3.4.2.1. Normal Executions Simulate Product Executions

The first result we prove is that each execution of a product that terminates normally is simulated by a normally-terminating execution of the original statement for the executions whose activation variables are true, and the parts of the product's store belonging to executions whose activation variables are false do not change.

Theorem 3.4.2 *Assume that match*$(\Phi, \underline{\Phi})$ *and that for some sets* $A \subseteq \{1, \ldots, k\}$ *and* $I = \{1, \ldots, k\} \setminus A$ *we have that* $\forall i \in A.\, p^{(i)} \Downarrow_{\underline{\sigma}} \top \wedge \sigma_i \in_i \underline{\sigma}$ *and* $\forall i \in I.\, p^{(i)} \Downarrow_{\underline{\sigma}} \bot$. *Let* $\{p^{(1)}, \ldots, p^{(k)}\} \cap (\mathit{RPV}\textsc{ar} \cup \mathit{freshvars}(\underline{s})) = \emptyset$ *and* $\underline{s} = \llbracket s \rrbracket_k^{\mathring{p}}$ *and* $\langle \underline{s}, \underline{\sigma} \rangle \rightarrow^l \langle \texttt{skip}, \underline{\sigma}' \rangle$ *under* $\underline{\Phi}$.

Then

1. *for all* $i \in A$, $\langle s, \sigma_i \rangle \rightarrow^* \langle \texttt{skip}, \sigma_i' \rangle$ *under* Φ *for some* σ_i' *s.t.* $\sigma_i' \in_i \underline{\sigma}'$, *and* $\underline{\sigma} \precsim_i^{\underline{s}} \underline{\sigma}'$,
2. *for all* $i \in I$, $\underline{\sigma} \preccurlyeq_i^{\underline{s}} \underline{\sigma}'$.

Proof. The full proof can be found in Appendix A.1.2. The proof goes by strong induction on the length l of the derivation $\langle \llbracket s \rrbracket_k^{\mathring{p}}, \underline{\sigma} \rangle \rightarrow^l \langle \texttt{skip}, \underline{\sigma}' \rangle$. We perform a case split on the structure of s, which determines the structure of the product $\underline{s}$. The cases for assignments, havocs, assumes, and assertions are simple using Lemma 3.4.1. For other statements, we generally perform as many steps of $\underline{s}$ as necessary to get to a configuration where the statement is again a product program. We show that the product state in these configuration mirrors the states of (some of) the original executions, so that we can apply the induction hypothesis. Subsequently, we construct complete derivations of the original executions and show that the final states mirror the final state of the product. □

We also prove that products always terminate and do not modify existing state (except for fresh variables) if all activation variables are false:

Lemma 3.4.3 *If, for all* $i \in \{1, ..., k\}$, $p^{(i)} \Downarrow_{\underline{\sigma}} \bot$, *then for some* $\underline{\sigma}'$, $\langle \underline{s} = \llbracket s \rrbracket_k^{\mathring{p}}, \underline{\sigma} \rangle \rightarrow^* \langle \texttt{skip}, \underline{\sigma}' \rangle$ *and* $\underline{\sigma} \preccurlyeq^{\underline{s}} \underline{\sigma}'$.

Proof. By induction on the structure of s. For simple statements the proof is immediate. For loops, the condition of the loop in the product must be false if all activation variables are false, and the product immediately steps to `skip`. The same is true for method calls and the condition of the conditional that wraps the call in the product. The cases for sequential composition and conditionals follow immediately from applying the induction hypothesis to the substatements. □

We now show that a product execution terminating abnormally is simulated by a set of executions of which at least one terminates abnormally.

Lemma 3.4.4 *Assume that match*$(\Phi, \underline{\Phi})$ *and that for some sets* $A \subseteq \{1, \ldots, k\}$ *and* $I = \{1, \ldots, k\} \setminus A$ *we have that* $\forall i \in A.\, p^{(i)} \Downarrow_{\underline{\sigma}} \top \wedge \sigma_i \in_i \underline{\sigma}$ *and* $\forall i \in I.\, p^{(i)} \Downarrow_{\underline{\sigma}} \bot$. *If* $\langle \underline{s}, \underline{\sigma} \rangle \rightarrow^l \langle \dot{s}, \underline{\sigma}' \rangle$, *where* $\underline{s} = \llbracket s \rrbracket_k^{\mathring{p}}$ *and* $\dot{s} \neq \texttt{skip}$, *under* $\underline{\Phi}$, *then for at least one* $i_e \in A$, $\langle s, \sigma_{i_e} \rangle \rightarrow^* \langle \dot{s}, \sigma_{i_e}' \rangle$ *under* Φ *for some* σ_{i_e}'.

Proof. The full proof can be found in Appendix A.1.3. The proof goes by strong induction on l and follows roughly the same outline as the proof for Thm. 3.4.2. We again perform a case split on the structure of s. We show that the cases for assignments and havocs always terminate normally, and that product assertions and assumes terminate abnormally if at least one corresponding assert or assume terminates abnormally in an original execution (using Lemma 3.4.1). For all other statements, we perform case splits on which part of the product leads to abnormal termination. We use Thm. 3.4.2 to reason about the effects of all parts that are products and terminate normally, and apply the induction hypothesis to those that do not. □

As a corollary of the previous theorem and lemma, we observe that terminating product executions are simulated by sets of original executions that either all terminate, or of which at least one fails. An example of the latter is a program `assert` e_1`;while` (e_2) `{skip}`: If the assertion fails for one of the original executions but succeeds for all others, the other executions will diverge, whereas the product execution fails while executing the product of the assertion.

3.4.2.2. Product Executions Simulate Normal Executions

Now we prove that any set of terminating executions is simulated by a terminating execution of a modular product. The active executions of the product each mirror one of the original executions, and its inactive executions' state does not change (modulo assignments to fresh variables). We assume that we can map each of the executions to a distinct index in the range $1, \ldots, k$.

Theorem 3.4.5 *Assume that match*$(\Phi, \underline{\Phi})$ *and that for a set of indices* $A \subseteq \{1, \ldots, k\}$ *there is a derivation* $d_i = \langle s, \sigma_i \rangle \rightarrow^{l_i} \langle \mathtt{skip}, \sigma'_i \rangle$ *under* Φ *for each* $i \in A$*. Assume also that* $\sigma_i \in_i \underline{\sigma}$ *and* $p^{(i)} \Downarrow_{\underline{\sigma}} \top$ *for all* $i \in A$*, and* $p^{(i)} \Downarrow_{\underline{\sigma}} \bot$ *for any* $i \in I$*, where* $I = \{1, \ldots, k\} \setminus A$*.*

Then $\langle \underline{s}, \underline{\sigma} \rangle \rightarrow^{*} \langle \mathtt{skip}, \underline{\sigma}' \rangle$*, where* $\underline{s} = [\![s]\!]^{\mathring{p}}_{k}$*, under* $\underline{\Phi}$ *for some* $\underline{\sigma}'$ *s.t. for all* $i \in A$*,* $\sigma'_i \in_i \underline{\sigma}'$ *and* $\underline{\sigma} \precsim^{s}_{i} \underline{\sigma}'$*, and for all* $i \in I$*,* $\underline{\sigma} \preccurlyeq^{s}_{i} \underline{\sigma}'$*.*

Proof. The full proof can be found in Appendix A.1.4. The proof goes by strong induction on the sum of the lengths of derivations $l = \sum_{i \in A} l_i$. For $l = 0$ and $|A| = 0$, the conclusion follows from Lemma 3.4.3. If $l = 0$ and $|A| \neq 0$ then $s = \mathtt{skip}$ and the conclusion follows trivially. For $l > 0$ we perform a case split on the structure of s. The cases for assignments, asserts, assumes, and havocs are essentially the reverse of the corresponding cases in the proof of Thm. 3.4.2. For all other statements, we identify sets of traces that proceed in the same way and show using the induction hypothesis that the product executes subproducts that mirror the executions for those sets. □

Similarly to before, we now show that any set of terminating traces, at least one of which ends abnormally, is mirrored by a product execution that ends abnormally.

Lemma 3.4.6 *Assume that for a set of indices A s.t. $|A| = j$ and $1 \leq j \leq k$ and $A \subseteq \{1, \dots, k\}$ there is a derivation $d_i = \langle s, \sigma_i \rangle \rightarrow^{l_i} \langle \dot{s}_i, \sigma'_i \rangle$, where $\dot{s}_i \in \{\mathtt{skip}, \mathtt{error}, \mathtt{magic}\}$, under Φ for each $i \in A$. Assume given $A_s \subset A$ and $A_e = A \setminus A_s$ s.t. for all $i \in A_e$, $\dot{s}_i \neq \mathtt{skip}$, and for all $i \in A_s$, $\dot{s}_i = \mathtt{skip}$. Assume also that $\sigma_i \in_i \underline{\sigma}$ and $\underline{\sigma}(p^{(i)}) = \top$ for all $i \in A$, and $\underline{\sigma}(p^{(i)}) = \bot$ for any $i \in I$, where $I = \{1, \dots, k\} \setminus A$.*

Then $\langle \underline{s}, \underline{\sigma} \rangle \rightarrow^* \langle \dot{s}, \underline{\sigma}' \rangle$ under $\underline{\Phi}$, where $\underline{s} = \llbracket s \rrbracket^{\mathring{p}}_k$, for some $\underline{\sigma}'$ and $\dot{s}$ s.t. $\exists i \in A_e.\, \dot{s}_i = \dot{s}$.

Proof. The full proof can be found in Appendix A.1.5. The proof follows roughly the same outline as the proof for Thm. 3.4.5. We again perform a case split on the structure of s, and a further case split on which part of the original executions lead to errors. We use Thm. 3.4.5 to reason about successful executions of substatements that are products, and apply the induction hypothesis to the executions of failing product substatements. □

As a corollary, we observe that if a set of executions terminates, the corresponding product execution always terminates as well.

3.4.3. Provable Properties

We now show that since product executions mirror sets of executions of the original program, it follows that properties that hold for product programs correspond to hyperproperties of the original programs.

In particular, we prove soundness, i.e., if a transformed relational specification holds for a product program then the relational specification holds for sets of executions of the original program. Conversely, we show completeness, i.e., for any hyperproperty which holds for multiple executions of a program, the transformed property holds for its product.

First, we lift the correspondence of expression evaluation in product stores and original stores to unary assertions, and show that a product store that mirrors the original stores of all active executions fulfills a transformed assertion if and only if the original stores fulfill the original assertion.

Lemma 3.4.7 *If $A \subseteq \{1, \dots, k\}$ and $\forall i \in A.\, \sigma_i \in_i \underline{\sigma} \wedge p^{(i)} \Downarrow_{\underline{\sigma}} \top$ and $\forall i \in \{1, \dots, k\} \setminus A.\, p^{(i)} \Downarrow_{\underline{\sigma}} \bot$ and $fv(P) \subseteq PV\textsc{ar}$ then $(\bigwedge_{i \in A} \sigma_i \vDash P) \Leftrightarrow \underline{\sigma} \vDash \lfloor P \rfloor^{\mathring{p}}_k$.*

Proof. Both directions of the proof go by induction on the structure of P. Cases relating to expressions follow from Lemma 3.4.1; all others follow from using the induction hypothesis on subassertions. □

Second, we show that the same property holds for tuples of stores and mixed assertions (and therefore relational assertions as well).

Lemma 3.4.8 *If* $\forall i \in \{1, \ldots, k\}.\, \sigma_i \in_i \underline{\sigma} \land p^{(i)} \Downarrow_{\underline{\sigma}} \top$ *and* $fv(\check{P}) \subseteq PV\textsc{ar}$ *then* $(\sigma_1, \ldots, \sigma_k) \vDash \check{P} \Leftrightarrow \underline{\sigma} \vDash \lfloor \check{P} \rfloor^{\mathring{p}}_k$.

Proof. Both directions of the proof go by induction on the structure of $\check{P}$. Cases relating to unary assertions are covered by Lemma 3.4.7; cases relating to expressions follow from Lemma 3.4.1; all others follow from the induction hypothesis on subassertions. □

We can now prove the soundness theorem: If a transformed relational specification holds for a modular k-product program, then the relational specification holds for any k executions of the original program.

Theorem 3.4.9 *If* $\vDash \{\lfloor \check{P} \rfloor^{\mathring{p}}_k\} \llbracket s \rrbracket^{\mathring{p}}_k \{\lfloor \check{Q} \rfloor^{\mathring{p}}_k\}$ *then* $\vDash \{\!\!\{\check{P}\}\!\!\} s \{\!\!\{\check{Q}\}\!\!\}_k$

Proof. We have to show that for all $(\sigma_1, \ldots, \sigma_k)$ s.t. $(\sigma_1, \ldots, \sigma_k) \vDash \check{P}$, if for all $i \in \{1, \ldots, k\}$ $\langle s, \sigma_i \rangle \longrightarrow^* \langle \dot{s}_i, \sigma'_i \rangle$ for some σ'_i and $\dot{s}_i \in \{\texttt{skip}, \texttt{error}\}$, then all $\dot{s}_i = \texttt{skip}$ and $(\sigma'_1, \ldots, \sigma'_k) \vDash \check{Q}$.

Since we have $\vDash \{\lfloor \check{P} \rfloor^{\mathring{p}}_k\} \llbracket s \rrbracket^{\mathring{p}}_k \{\lfloor \check{Q} \rfloor^{\mathring{p}}_k\}$, we know that if $\underline{\sigma} \vDash \lfloor \check{P} \rfloor^{\mathring{p}}_k$ and $\langle \llbracket s \rrbracket^{\mathring{p}}_k, \underline{\sigma} \rangle \longrightarrow^* \langle \dot{s}, \underline{\sigma}' \rangle$ where $\dot{s} \in \{\texttt{skip}, \texttt{error}, \texttt{magic}\}$, then either $\dot{s} = \texttt{magic}$ or $\dot{s} = \texttt{skip} \land \underline{\sigma}' \vDash \lfloor \check{Q} \rfloor^{\mathring{p}}_k$.

We choose $\underline{\sigma}$ s.t. $p^{(i)} \Downarrow_{\underline{\sigma}} \top$ and $\sigma_i \in_i \underline{\sigma}$ for all $i \in \{1, \ldots, k\}$. By Lemma 3.4.8 we have that $\underline{\sigma} \vDash \lfloor \check{P} \rfloor^{\mathring{p}}_k$.

Assume that for all i we have $\langle s, \sigma_i \rangle \longrightarrow^* \langle \dot{s}_i, \sigma'_i \rangle$ (otherwise the proof goal is trivially true).

- If all $\dot{s}_i = \texttt{skip}$, then by Thm. 3.4.5 this implies that there is a product execution $\langle \llbracket s \rrbracket^{\mathring{p}}_k, \underline{\sigma} \rangle \longrightarrow^* \langle \texttt{skip}, \underline{\sigma}' \rangle$ for some $\underline{\sigma}'$. By Thm. 3.4.2 we then have that that $\underline{\sigma}' \vDash \lfloor \check{Q} \rfloor^{\mathring{p}}_k$ and for all $i \in \{1, \ldots, k\}$, $\underline{\sigma}'(p^{(i)}) = \top$ and $\sigma'_i \in_i \underline{\sigma}'$. By Lemma 3.4.8 we have $(\sigma'_1, \ldots, \sigma'_k) \vDash \check{Q}$.
- Otherwise: If all $\dot{s}_i \neq \texttt{error}$, we must have that some $\dot{s}_i = \texttt{magic}$, in which case we are done. If some $\dot{s}_i = \texttt{error}$ we case split on the structure of the precondition $\check{P}$:
  - If $\check{P} = \hat{P}'$ for some $\hat{P}'$: Then we trivially have that $\vDash \{\!\!\{\check{P}\}\!\!\} s \{\!\!\{\check{Q}\}\!\!\}_k$.
  - If $\check{P} = P'$ for some P': Then we can choose some $\underline{\sigma}''$ s.t. $\underline{\sigma}(p^{(i)}) = \top$ and $\sigma_i \in_i \underline{\sigma}''$ and $\underline{\sigma}(p^{(j)}) = \bot$ for all $j \neq i$. By Lemma 3.4.7 we have that $\underline{\sigma}'' \vDash \lfloor \check{P} \rfloor^{\mathring{p}}_k$. Then by Thm. 3.4.6 this implies that there is a product execution $\langle \llbracket s \rrbracket^{\mathring{p}}_k, \underline{\sigma}'' \rangle \longrightarrow^* \langle \texttt{error}, \underline{\sigma}''' \rangle$ for some $\underline{\sigma}'''$, which is impossible because $\vDash \{\lfloor \check{P} \rfloor^{\mathring{p}}_k\} \llbracket s \rrbracket^{\mathring{p}}_k \{\lfloor \check{Q} \rfloor^{\mathring{p}}_k\}$.
  - If $\check{P} = \hat{P}' \land P''$ for some $\hat{P}', P''$: Then this case is impossible for the same reason as in the previous case.

□

Similarly, we prove completeness: If some unary precondition P guarantees that a program executes without errors, and some relational specification expressing a hyperproperty holds for the program, then the transformed version of the resulting mixed specification holds for the modular product program.

The unary precondition P is necessary to guarantee that the product program does not fail in executions where some activation variables are false and a k-relational precondition is therefore not required to hold. As an example, the program $s = \texttt{assert}\ x > 0$ fulfills the relational specification $\{\!\{x^{(1)} > 0 \land x^{(2)} > 0\}\!\}s\{\!\{\mathsf{true}\}\!\}_2$; however, $\lfloor x^{(1)} > 0 \land x^{(2)} > 0\rfloor_2^{\mathring{p}} = (p^{(1)} \land p^{(2)} \Rightarrow x^{(1)} > 0 \land x^{(2)} > 0)$, and a store $[p^{(1)} \mapsto \bot, p^{(2)} \mapsto \top, x^{(1)} \mapsto 0, x^{(2)} \mapsto 0]$ therefore fulfills the transformed precondition but raises an error when the product is executed. This is not a limitation in practice, since the unary precondition that guarantees error-free execution can always be extracted from a relational specification; in the example, we can replace the relational precondition $x^{(1)} > 0 \land x^{(2)} > 0$ by a unary precondition $x > 0$.

Theorem 3.4.10 *Let $\hat{P}$ and $\hat{Q}$ be k-relational assertions, and $\models \{P\}s\{\mathsf{true}\}$ and $\models \{\!\{\hat{P}\}\!\}s\{\!\{\hat{Q}\}\!\}_k$*

Then $\models \{\lfloor P \land \hat{P}\rfloor_k^{\mathring{p}}\}\llbracket s\rrbracket_k^{\mathring{p}}\{\lfloor \hat{Q}\rfloor_k^{\mathring{p}}\}$.

Proof. We first show that for all $\underline{\sigma}$ s.t. $\underline{\sigma} \models \lfloor P \land \hat{P}\rfloor_k^{\mathring{p}}$, if $\langle \llbracket s\rrbracket_k^{\mathring{p}}, \underline{\sigma}\rangle \rightarrow^* \langle \dot{s}, \underline{\sigma}'\rangle$ for some $\dot{s}, \underline{\sigma}'$, then $\dot{s} = \texttt{magic}$ or $\dot{s} = \texttt{skip} \land \underline{\sigma}' \models \lfloor \hat{Q}\rfloor_k^{\mathring{p}}$.

Assume that $\underline{\sigma} \models \lfloor P \land \hat{P}\rfloor_k^{\mathring{p}}$ and $\langle \llbracket s\rrbracket_k^{\mathring{p}}, \underline{\sigma}\rangle \rightarrow^* \langle \dot{s}, \underline{\sigma}'\rangle$ (otherwise the goal is trivially true).

- If $\dot{s} \neq \texttt{skip}$ then we get by Lemma 3.4.4 that for some $i \in \{1, \ldots, k\}$, $\underline{\sigma}(p^{(i)}) = \top$, and for any σ_i s.t. $\sigma_i \in_i \underline{\sigma}$ we have $\langle s, \sigma_i\rangle \rightarrow^* \langle \dot{s}, \sigma_i'\rangle$ for some σ_i'.
    - If $\dot{s} = \texttt{magic}$ then the specification holds trivially.
    - If $\dot{s} = \texttt{error}$ then we choose some such σ_i and by Lemma 3.4.7 we get that $\sigma_i \models P$. But since $\models \{P\}s\{\mathsf{true}\}$ we have that if $\langle s, \sigma_i\rangle \rightarrow^* \langle \dot{s}_i', \sigma_i'\rangle$ then $\dot{s}_i' \neq \texttt{error}$. Therefore we cannot have that $\langle s, \sigma_i\rangle \rightarrow^* \langle \texttt{error}, \sigma_i'\rangle$, and this case is impossible.
- If $\dot{s} = \texttt{skip}$, then we have by Thm. 3.4.2 that for all i s.t. $\underline{\sigma}(p^{(i)}) = \top$, $\langle s, \sigma_i\rangle \rightarrow^* \langle \texttt{skip}, \sigma_i'\rangle$ for some σ_i' s.t. $\sigma_i' \in_i \underline{\sigma}'$ and $\underline{\sigma}'(p^{(i)}) = \top$.
    - If $\forall i \in \{1, \ldots, k\}.\ \underline{\sigma}(p^{(i)}) = \top$, then we have that also $\forall i \in \{1, \ldots, k\}.\ \underline{\sigma}'(p^{(i)}) = \top$. Then by Lemma 3.4.8, we have that $\underline{\sigma}' \models \lfloor \hat{Q}\rfloor_k^{\mathring{p}}$.
    - If $\exists i \in \{1, \ldots, k\}.\ \underline{\sigma}(p^{(i)}) = \bot$, then we have that for said i, $\underline{\sigma}'(p^{(i)}) = \bot$. Since $\lfloor \hat{Q}\rfloor_k^{\mathring{p}} = (\bigwedge_{i=1}^{k} p^{(i)}) \Rightarrow \hat{Q}$, we trivially have that $\underline{\sigma}' \models \lfloor \hat{Q}\rfloor_k^{\mathring{p}}$.

Therefore we have $\models \{\lfloor P \land \hat{P}\rfloor_k^{\mathring{p}}\}\llbracket s\rrbracket_k^{\mathring{p}}\{\lfloor \hat{Q}\rfloor_k^{\mathring{p}}\}$. □

3.5. Modular Verification of Non-Interference

In this section, we demonstrate the expressiveness of modular product programs by showing how they can be used to verify non-interference, an important hyperproperty which can be used to express both information flow security and integrity. To simplify the presentation, we will describe our approach exclusively in terms of information flow security throughout; the methodology for showing integrity properties is exactly the same but uses different labels. We first concentrate on secure information flow in the classical sense [221], and later demonstrate how the ability to check relational assertions at any point in the program can be exploited to prove advanced properties like the absence of timing and termination channels, and to encode declassification.

[221]: Volpano et al. (1996), 'A Sound Type System for Secure Flow Analysis'

3.5.1. Non-Interference

Properties like secure information flow that describe that some (secret) program inputs do not influence (public) program outputs can be expressed as a 2-safety hyperproperty of a program called *non-interference*. Non-interference states that, if a program is run twice, with the public (*low-sensitivity*, often just called *low*) inputs being equal in both runs but the secret (*high-sensitivity*, or *high*) inputs possibly being different, the public outputs of the program must be equal in both runs [25]. This property guarantees that the high inputs do not influence the low outputs.

[25]: Barthe et al. (2011), 'Secure information flow by self-composition'

We can formalize non-interference as follows:

Definition 3.5.1 *A program s with a set of input variables I and output variables O, of which some subsets $I_l \subseteq I$ and $O_l \subseteq O$ are low, satisfies noninterference iff for all σ_1, σ_2 and σ'_1, σ'_2, if $\forall x \in I_l.\, \sigma_1(x) = \sigma_2(x)$ and $\langle s, \sigma_1 \rangle \rightarrow^* \langle \texttt{skip}, \sigma'_1 \rangle$ and $\langle s, \sigma_2 \rangle \rightarrow^* \langle \texttt{skip}, \sigma'_2 \rangle$ then $\forall x \in O_l . \sigma'_1(x) = \sigma'_2(x)$.*

As stated above, the same definition can be applied to the setting of integrity properties if the labels are switched: In this setting, the *low-integrity* inputs must not influence the *high-integrity* outputs.

Since our definition of non-interference describes a hyperproperty, we can verify it using modular product programs:

Theorem 3.5.1 *A program s with a set of input variables I and output variables O, of which some subsets $I_l \subseteq I$ and $O_l \subseteq O$ are low, satisfies non-interference under a unary precondition P if $\models \{\lfloor P \rfloor_2^{\mathring{p}} \land (\bigwedge_{x \in I_l} x^{(1)} = x^{(2)})\} [\![s]\!]_2^{\mathring{p}} \{\forall x \in O_l .\, x^{(1)} = x^{(2)}\}$*

Proof. Since non-interference can be expressed using a 2-relational specification, the theorem follows directly from Theorem 3.4.9. □

An expanded notion of secure information flow considers observable *events* in addition to regular program outputs [89]. An event is a statement that has an effect that is visible to an outside observer, but may not necessarily affect the program state. The most important examples of events are output operations like printing a string to the console or sending a message over a network. Programs that cause events can be considered information flow secure only if the sequence of produced

[89]: Giffhorn et al. (2015), 'A new algorithm for low-deterministic security'

events is not influenced by high data. One way to verify this using our approach is to track the sequence of produced events in a ghost variable and verify that its value never depends on high data. This approach requires substantial amounts of additional specifications.

Modular product programs offer an alternative approach for preventing leaks via events, since they allow formulating assertions about the relation between the activation variables of different executions. In particular, if a given event has the precondition that all activation variables are equal when the event statement is reached then this event will either be executed by both executions or be skipped by both executions. As a result, the sequence of events produced by a program will be equal in all executions.

3.5.2. Information Flow Specifications

The relational specifications required for modularly proving non-interference with the previously described approach have a specific pattern: they can contain functional specifications meant to be valid for both executions (e.g., to make sure both executions run without errors), they may require that some information is low, which is equivalent to the two renamings of the same expression being equal, and, in addition, they may assert that the control flow at a specific program point is low.

We therefore introduce modular *information flow specifications*, which can express properties required for proving secure information flow but are transparent w.r.t. the encoding or the verification methodology, i.e., they allow expressing that a given event or value must not be secret without knowledge of the encoding of this fact into an assertion about two different program executions. We define information flow specifications as follows:

$$\begin{array}{lcl} (\mathit{SIFAssertions}) \quad \tilde{P} & ::= & \tilde{P} \wedge \tilde{P} \mid e \mid \mathit{low}(e) \mid \mathit{lowEvent} \mid \forall x.\, \tilde{P} \\ & & \mid e \Rightarrow e \mid e \Rightarrow \mathit{low}(e) \mid \mathit{low}(e) \Rightarrow \mathit{low}(e) \\ & & \mid e \Rightarrow \mathit{lowEvent} \end{array}$$

$\mathit{low}(e)$ specifies that the value of the expression e is not influenced by high data. Note that e can be any expression and is not limited to variable references; this reflects the fact that our approach can label secrecy in a more fine-grained way than, e.g., a type system; one can, for example, declare to be public whether a number is odd while keeping its value secret.

lowEvent specifies that high data must not influence if and how often the current program point is reached by an execution, which is a sufficient precondition of any statement that causes an observable event. In particular, if a method outputs an expression e, the precondition $\mathit{lowEvent} \wedge \mathit{low}(e)$ guarantees that no high information will be leaked via this method.

Information flow specifications can express complex properties. $e_1 \Rightarrow \mathit{low}(e_2)$, for example, expresses that if e_1 is true, e_2 must not depend on high data; this is sometimes called *value-dependent sensitivity* [160]. $e_1 \Rightarrow \mathit{lowEvent}$ says the same about the current control flow. A possible use case of these assertions is the precondition of a library function that

[160]: Murray et al. (2018), 'COVERN: A Logic for Compositional Verification of Information Flow Control'

$$\begin{array}{lcl}
\lceil e \rceil^{\mathring{p}} & = & (p^{(1)} \Rightarrow e^{(1)}) \land (p^{(2)} \Rightarrow e^{(2)}) \\
\lceil low(e) \rceil^{\mathring{p}} & = & p^{(1)} \land p^{(2)} \Rightarrow e^{(1)} = e^{(2)} \\
\lceil lowEvent \rceil^{\mathring{p}} & = & p^{(1)} = p^{(2)} \\
\lceil \tilde{P}_1 \land \tilde{P}_2 \rceil^{\mathring{p}} & = & \lceil \tilde{P}_1 \rceil^{\mathring{p}} \land \lceil \tilde{P}_2 \rceil^{\mathring{p}} \\
\lceil e_1 \Rightarrow e_2 \rceil^{\mathring{p}} & = & (p^{(1)} \land {e_1}^{(1)} \Rightarrow {e_2}^{(1)}) \land (p^{(2)} \land {e_1}^{(2)} \Rightarrow {e_2}^{(2)}) \\
\lceil e_1 \Rightarrow low(e_2) \rceil^{\mathring{p}} & = & p^{(1)} \land {e_1}^{(1)} \land p^{(2)} \land {e_1}^{(2)} \Rightarrow {e_2}^{(1)} = {e_2}^{(2)} \\
\lceil e_1 \Rightarrow lowEvent \rceil^{\mathring{p}} & = & (p^{(1)} \land e^{(1)} \lor p^{(2)} \land e^{(2)}) \Rightarrow p^{(1)} = {p_2}^{(2)} \\
\lceil low(e_1) \Rightarrow low(e_2) \rceil^{\mathring{p}} & = & p^{(1)} \land p^{(2)} \land {e_1}^{(1)} = {e_1}^{(2)} \Rightarrow {e_2}^{(1)} = {e_2}^{(2)} \\
\lceil \forall x.\, \tilde{P} \rceil^{\mathring{p}} & = & \forall x^{(1)}, x^{(2)}.\, x^{(1)} = x^{(2)} \Rightarrow \lceil \tilde{P} \rceil^{\mathring{p}}
\end{array}$$

Figure 3.6.: Translation of information flow specifications. The intuition behind the translation of universal quantifiers is that we quantify over a single variable x but give it two different names $x^{(1)}$ and $x^{(2)}$ so that we can transform the body of the quantifier like any other assertion.

prints e_2 to a low-observable channel if e_1 is true, and to a secure channel otherwise.

In addition to the two primitives *low*() and *lowEvent*, information flow specifications can also contain ordinary functional specifications for single executions. These can be used as a fallback if said primitives are not expressive enough. As an example, for proving that the statement `if` (e) $\{x.m()\}$ `else` $\{x.n()\}$ ends with x being low, it may be necessary to give a description of the functional behavior of m and n. Similarly, if both m and n perform the same output, requiring the calls to be a *lowEvent* (which is sufficient but generally not necessary) is too strong a requirement, and their functional contracts must model the output explicitly instead.

Note that high-ness of some expression is not modeled by its renamings being definitely unequal, but by leaving underspecified whether they are equal or not, meaning that high-ness is simply the absence of the knowledge of low-ness. As a result, it is never necessary (or possible) to specify explicitly that an expression is high. This approach (which is also used in self-composition) is analogous to the way type systems encode security levels, where low is typically a subtype of high.

The encoding $\lceil \tilde{P} \rceil^{\mathring{p}}$ of an information flow assertion $\tilde{P}$ under the activation variables $p^{(1)}$ and $p^{(2)}$ is defined in Figure 3.6. For the example in Figure 3.1, a possible, very precise information flow specification could say that the results of main are low if the first two bits of all entries in people is low. We can write this as $\{low(|\mathsf{people}|) \land \forall i \in \{0, \ldots, |\mathsf{people}| - 1\}.\, low(\mathsf{people[i]\ mod\ 4})\}\mathsf{main}\{low(\mathsf{count})\}$. In the product, this will be translated to

$$\left\{ \begin{array}{c} \mathsf{p1} \land \mathsf{p2} \Rightarrow (|\mathsf{people1}| = |\mathsf{people2}| \land \\ \forall i \in \{0, \ldots, |\mathsf{people1}| - 1\}. \\ (\mathsf{people1[i]\ mod\ 4}) = (\mathsf{people2[i]\ mod\ 4})) \end{array} \right\}$$
$$\mathsf{main}$$
$$\{\mathsf{p1} \land \mathsf{p2} \Rightarrow \mathsf{count1} = \mathsf{count2}\}$$

In this scenario, the loop in main could have the simple invariant $low(\mathsf{i}) \land low(\mathsf{count})$, and the method is_smoker could have the contract $\{true\}\mathsf{is\_smoker}\{(low(\mathsf{person\ mod\ 4}) \Rightarrow low(\mathsf{res}))\}$. This contract follows a useful pattern where, instead of requiring an input to be low and promising that an output will be low for all calls, the output is described as *conditionally* low based on the level of the input, which is more permissive for callers.

```
method check(password, input)
      returns (result)
{
  result := |password| == |input|;
  i := 0;
  while (i < |password| & & result) {
     result := result & & password[i] == input[i];
     i := i + 1;
  }
}
```

Figure 3.7.: Password check example: Leaking secret data is desired.

The example shows that the information required for proving secure information flow can in many cases be expressed concisely, without requiring any knowledge about the methodology used for verification. Modular product programs therefore enable the verification of the information flow security of main based solely on modular, relational specifications, and without depending on functional specifications.

3.5.3. Secure Information Flow with Arbitrary Security Lattices

The definition of secure information flow used in Definition 3.5.1 is a special case in which there are exactly two possible classifications of data, high and low. In the more general case, classifications come from an arbitrary lattice $\langle \mathcal{L}, \sqsubseteq \rangle$ of security levels s.t. for some $l_1, l_2 \in \mathcal{L}$, information from an input with level l_1 may influence an output with level l_2 only if $l_1 \sqsubseteq l_2$. Instead of the specification $low(e)$, information flow assertions can therefore have the form $levelBelow(e, l)$, meaning that the security level of expression e is at most l.

It is well-known that techniques for verifying information flow security with two levels can conceptually be used to verify programs with arbitrary finite security lattices [164] by splitting the verification task into $|\mathcal{L}|$ different verification tasks, one for each element of $\mathcal{L}$. Instead, we propose to combine all these verification tasks into a single task by using a symbolic value for l, i.e., declaring an unconstrained global constant representing l. Specifications can then be translated as follows:

[164]: Naumann (2006), 'From Coupling Relations to Mated Invariants for Checking Information Flow'

$$levelBelow(e, l') \mathrel{\hat{=}} l' \sqsubseteq l \Rightarrow e^{(1)} = e^{(2)}$$

Since no information about l is known, verification will only succeed if all assertions can be proven for all possible values of l, which is equivalent to proving them separately for each possible value of l. This approach is similar to the one used in information flow type systems and logics, which paramaterize their judgements by a free attacker level.

3.5.4. Declassification

In practice, non-interference is too strong a property for many use cases. Often, some leakage of secret data is required for a program to work correctly. Consider the case of a password check (see Figure 3.7): A secret internal password is compared to a non-secret user input. While the password itself must not be leaked, the information whether the user input matches the password should influence the public outcome of the program, which is forbidden by non-interference.

To incorporate this intention, the part of the secret information that should be leaked by the program can be *declassified* [188], e.g., via a declassification statement `declassify` e in the style of delimited information release [186] that declares an arbitrary expression e to be low. With modular products (as with self-composition [211]), declassification can be encoded via a simple assumption stating that, if the declassification is executed in both executions, the expression is equal in both executions:

[188]: Sabelfeld et al. (2005), 'Dimensions and Principles of Declassification'

[186]: Sabelfeld et al. (2003), 'A Model for Delimited Information Release'

[211]: Terauchi et al. (2005), 'Secure Information Flow as a Safety Problem'

$$[\![\texttt{declassify } e]\!]_2^{\mathring{p}} = \texttt{assume } p^{(1)} \land p^{(2)} \Rightarrow e^{(1)} = e^{(2)}$$

Importantly, an assumption of this form can never contradict current knowledge (i.e., assume *false*) if the information flow specifications from Sec. 3.5.2 are used to specify the program. Recall that high-ness is encoded as the absence of the knowledge that an expression is equal in both executions, *not* by the knowledge that they are different. In fact, the format of information flow specifications makes it impossible to specify that an expression is definitely not equal in both executions, since $low(e)$ can only occur in positive positions, i.e., not on the left side of an implication; the only exception, $low(e_1) \Rightarrow low(e_2)$, implies that ${e_1}^{(1)} \neq {e_1}^{(2)}$ only under the condition that one can already prove ${e_2}^{(1)} \neq {e_2}^{(2)}$, which as just argued can never happen. As a result, there is no danger that assuming equality will contradict current knowledge, since any such assumption (and any combination of such assumptions) will always be fulfilled if the state of both executions represented in the product program is completely identical. Nevertheless, assumptions outside of method contracts can obfuscate what property was actually proved. In order to get better formal guarantees, one could for example generate declassification statements based on some declassification policy. The exact process of doing this, however, is orthogonal our approach. As in the information flow specifications, the declassified expression can be arbitrarily complex, so that it is for example possible to declassify the parity of a number while keeping all other information about it secret.

The example in Fig. 3.7 becomes valid if we add `declassify` **result** at the end of the method, or `declassify` *equal*(password, input) at some earlier point. The latter would arguably be safer because it specifies exactly the information that is intended to be leaked, and would therefore prevent accidentally leaking more if the implementation of the checking loop was faulty.

This kind of declassification has the following interesting properties: First, it is *imperative*, meaning that the declassified information may be leaked (e.g., via a print statement) after the execution of the declassification statement, but not before. Second, it is *semantic*, meaning that the declassification affects the value of the declassified expression as opposed to, e.g., syntactically the declassified variable. As a result, it will be allowed to leak any expression whose value contains the same (or a part of the) secret information which was declassified, e.g., the expression $f(e)$ if f is a deterministic function and e has been declassified.

While declassification statements by themselves are quite primitive, they can be used to implement more complex declassification policies by

```
method main(h: Int)
{
  while (h != 0) {
    h := h - 1;
  }
}
```

```
method main(h: Int)
{
  i := 0;
  while (i < h) {
    i := i + 1
  }
  print(0)
}
```

Figure 3.8.: Programs with a termination channel (left), and a timing channel (right). In both cases, h is high.

making the statements conditional, i.e., by declassifying information under the condition that some policy is fulfilled.

3.5.5. Preventing Termination Channels

In Definition 3.5.1, we have considered only terminating program executions. In practice, however, termination is a possible side-channel that can leak secret information to an outside observer. Figure 3.8 (left) shows an example of a program that verifies under the methodology presented so far, but leaks information about the secret input h to an observer: If h is initially negative, the program will enter an endless loop. Anyone who can observe the termination behavior of the program can therefore conclude if h was negative or not.

To prevent leaking information via a termination side channel, it is necessary to verify that the termination of a program depends only on public data. We will show that modular product programs are expressive enough to encode and check this property. We will focus on preventing non-termination caused by infinite loops here; preventing infinite recursion works analogously. In particular, we want to prove that if a loop iterates forever in one execution, any other execution with the same low inputs will also reach this loop and iterate forever. More precisely, this means that

(A) if a loop does not terminate, then whether or not an execution reaches that loop must not depend on high data.
(B) whether a loop that is reached by both executions terminates must not depend on high data.

We propose to verify these properties by requiring additional specifications that state, for every loop, an *exact* condition under which it terminates, i.e., a boolean expression such that the loop terminates if and only if the expression is true. For Figure 3.8 (left) the condition is $h \geq 0$. We also require a ranking function for the cases when the termination condition is true. We can then prove the following:

1. If the termination condition of a loop evaluates to false, then any two executions with identical low inputs either both reach the loop or both do not reach the loop (i.e., reaching the loop is a low event). This guarantees property (A) above.
2. For loops executed by both executions, the loop's termination condition is low. This guarantees property (B) under the assumption that the termination condition is exact.
3. The termination condition is sound, i.e., every loop terminates if its termination condition is true. We prove this by showing that if

```
$term(w, c)$  =  $cond$:=$e_c$;
                 assert $\neg cond \Rightarrow lowEvent$;            // checks (a)
                 assert $low(cond)$;                           // checks (b)
                 assert $cond \Rightarrow 0 \leq e_r$;               // checks (c)
                 assert $c \Rightarrow cond$;                     // checks (e)
                 assert $\neg cond \Rightarrow e$;                  // checks (d)
                 while ($e$) do {
                    if ($cond$) {$rank$:=$e_r$};
                    $term(s, cond)$;
                    assert $cond \Rightarrow 0 \leq e_r \land e_r < rank$   // checks (c)
                    assert $\neg cond \Rightarrow e$;               // checks (d)
                 }
```

Figure 3.9.: Program instrumentation for termination leak prevention. We abbreviate `while` (e) `terminates`(e_c, e_r) $\{s\}$ as w. If $[\![term(s, false)]\!]_2^{\mathring{p}}$ verifies under some precondition, s does not have a termination side channel under this precondition. Statements of the form `assert` $\tilde{P}$ are to be interpreted as asserting the encoding `assert` $\lceil \tilde{P} \rceil^{\mathring{p}}$ in the product program.

the termination condition is true, we can prove the termination of the loop using the supplied ranking function.

4. The termination condition is complete, i.e., every loop terminates only if its termination condition is true. We prove this by showing that if the condition is false, the loop condition will always remain true. This check, along with the previous proof obligation, ensures that the termination condition is exact.
5. Every statement in a loop body terminates if the loop's termination condition is true, i.e., the loop's termination condition implies the termination conditions of all statements in its body.

We introduce an annotated while loop `while` (e) `terminates`(e_c, e_r) $\{s\}$, where e_c is the exact termination condition and e_r is the ranking function, i.e., an integer expression whose value decreases with every loop iteration but never becomes negative if the termination condition is true. Based on these annotations, we present a program instrumentation $term(s, c)$ that inserts the checks outlined above for every while loop in s. c is the termination condition of the outside scope, i.e., for the instrumentation of a nested loop, it is the termination condition e_c of the outer loop. The instrumentation is defined for annotated while loops in Figure 3.9; for all other statements, it does not make any changes except instrumenting all subnodes. The instrumentation guarantees that if the product of the instrumented program verifies, the original program does not leak secret information via a termination channel. We slightly abuse the notation and use `assert` statements with information flow assertions (as defined in Sec. 3.5.2) instead of expressions; the intended meaning is that the product of such a statement is a statement that asserts the encoded version of said assertion (which can be written as an expression). Again, we make use of the fact that modular products allow checking relational assertions at arbitrary program points and formulating assertions about the control flow.

We now prove that if an instrumented statement verifies under some 2-relational precondition then any two runs from a pair of states fulfilling that precondition will either both terminate or both loop forever.

Theorem 3.5.2 *Assume that* $\underline{s} = [\![term(s, false)]\!]_2^{\mathring{p}}$ *and* $\models \{\lfloor \check{P} \rfloor_2^{\mathring{p}}\}\underline{s}\{true\}$ *and* $(\sigma_1, \sigma_2) \models \check{P}$ *and* s *does not terminate with* `magic` *from* σ_1 *and* σ_2 *and* s

does not contain calls to (mutually) recursive methods.

Then either s always terminates from both σ_1 and σ_2 or s never terminates from both σ_1 and σ_2.

Proof. The full proof can be found in Appendix A.1.6. We first prove lemmas that state that if the product of an instrumented loop does not fail from a state, single executions of this loop from states that mirror the product state terminate (1) if and (2) only if their termination condition is true. We show (1) by a standard termination proof using the provided ranking function, and (2) by showing that the loop condition never becomes false. We then extend these lemmas to general instrumented statements. The main proof then goes by induction on the structure of s; we show that if the product of an instrumented statement does not fail from a state, pairs of executions from states that mirror the product state either both terminate or both diverge. □

Note that, while this approach is sound, it is incomplete, since it requires that the termination of every single loop by itself does not leak information. It will therefore reject, for example, a program consisting two consecutive loops, if the first terminates depending on secret data, and the second never terminates at all. Assuming that an attacker can observe only the termination of the program as a whole, no information is leaked since the program never terminates, and the program should be accepted. This incompleteness is similar to the one that occurs when specifying method invocations to be low events (discussed in Sec. 3.5.2).

3.5.6. Preventing Timing Channels

A program has a *timing channel* if high input data influences the program's execution time, meaning that an attacker who can observe the time the program executes can gain information about those secrets. Timing channels can occur in combination with observable events; the time at which an event occurs may depend on a secret even if the overall execution time of a program does not. Consider the example in Figure 3.8 (right). Assuming main receives a positive secret h, both the print statement and the end of the program execution will be reached later for larger values of h.

Using modular product programs, we can verify the absence of timing side channels in two ways:

1. In languages where all basic operations take constant time, timing channels can be avoided by proving that all branch conditions are low: If high data does not influence which statements are executed, and the execution time of individual statements also does not depend on the data they use, then high data cannot influence execution time overall.
2. Alternatively, we can prove the absence of timing channels by adding ghost state to the program that tracks the time passed since the program has started; this could, for example, be achieved via a simple step counting mechanism, or by tracking the sequence of previously executed bytecode statements. This ghost state is updated separately for both executions. We can then assert anywhere

in the program that the passed time does not depend on high data in the same way we do it for program variables. In particular, we can enforce that the passed time is equal whenever an observable event occurs, and we can enable users to write relational specifications that compare the time passed in both executions of a loop or a method.

3.6. Implementation and Evaluation

We have implemented both the general modular product program transformation and our approach for non-interference in the Viper verification infrastructure [159] and applied it to a number of example programs from the literature. [2] Both the implementation and examples are available at `http://viper.ethz.ch/modularproducts/`.

[159]: Müller et al. (2016), 'Viper: A Verification Infrastructure for Permission-Based Reasoning'

[2] In the next chapter, we will describe how we extend our implementation to Nagini.

3.6.1. Implementation in Viper

Our implementation supports an extended version of the Viper language that adds the following features:

1. Expressions of the form rel(x,i) that represent a reference $x^{(i)}$ to variable x from execution i and can be used to express general relational specifications
2. The assertions $low(e)$ and *lowEvent* for information flow specifications
3. A declassify statement
4. Variations of the existing method declarations and while loops that include the termination annotations shown in Sec. 3.5.5

The implementation transforms a program in this extended language into a modular k-product for arbitrary k in the original language, which can then be verified by the (unmodified) Viper backend verifiers, using either symbolic execution (SE) or verification condition generation (VGC). Error messages are automatically translated back to the original program.

In the resulting language, users can use unary, relational, and mixed assertions in method pre- and postconditions as well as loop invariants, and can therefore specify any hyperproperty that is expressible in the framework we presented so far.

Alternatively, if $k = 2$, specifications can be provided as information flow specifications (see Sec. 3.5.2) such that users need no knowledge about the transformation or the methodology behind information flow verification.

Declassification is implemented as described in Sec. 3.5.4. Our implementation optionally verifies the absence of timing channels; the metric chosen for tracking execution time is simple step-counting.

For languages with opaque object references, secure information flow can require proving that pointers are low, i.e., equal up to a consistent renaming of addresses. To avoid the complexity of reasoning about such a renaming, we choose the second approach of modeling the heap described in Sec. 3.3.3: Our implementation creates a single new statement for every new in the original program, but duplicates the fields each object has. As a result, if both executions execute the same new statement, the newly created object will be considered low afterwards (but the values of its fields might still be high).

For supporting unbounded heap data structures like lists or trees, we use Viper's existing support of iterated separating conjunctions, i.e., *quantified* permission assertions of the form $\textbf{forall} x \in s.\, \textbf{acc}(x.f)$, which represent the permissions to the f fields for all references in set s [158]. Note that it is also possible to support recursive predicates (we will show how in the next chapter), but we do not use them in our evaluation here.

[158]: Müller et al. (2016), 'Automatic Verification of Iterated Separating Conjunctions Using Symbolic Execution'

3.6.2. Evaluation: Secure Information Flow

We have evaluated our approach for verifying non-interference in the context of information flow security by verifying a number of examples in the extended Viper language using our implementation. The examples are listed in Table 3.1 and include all code snippets shown in this chapter as well as a number of examples from the literature [8, 16, 23, 51, 56, 89, 117, 164, 204, 211]. They combine complex language features like mutable state on the heap, arrays and method calls, as well as timing and termination channels, declassification, and non-trivial information flows (e.g., flows whose legality depends on semantic information not available in a standard information flow type system). We manually added pre- and postconditions as well as loop invariants; for those examples that have forbidden flows and therefore should not verify, we also added a legal version that declassifies the leaked information.

[8]: Antonopoulos et al. (2017), 'Decomposition instead of self-composition for proving the absence of timing channels'
[16]: Banerjee et al. (2002), 'Secure Information Flow and Pointer Confinement in a Java-like Language'
[23]: Barthe et al. (2011), 'Relational Verification Using Product Programs'
[51]: Costanzo et al. (2014), 'A Separation Logic for Enforcing Declarative Information Flow Control Policies'
[56]: Darvas et al. (2005), 'A Theorem Proving Approach to Analysis of Secure Information Flow'
[89]: Giffhorn et al. (2015), 'A new algorithm for low-deterministic security'
[117]: Küsters et al. (2015), 'A Hybrid Approach for Proving Noninterference of Java Programs'
[164]: Naumann (2006), 'From Coupling Relations to Mated Invariants for Checking Information Flow'
[204]: Smith (2007), 'Principles of Secure Information Flow Analysis'
[211]: Terauchi et al. (2005), 'Secure Information Flow as a Safety Problem'

3.6.2.1. Qualitative Evaluation

Our implementation returns the correct result for all examples. In all cases but one, our approach allows us to express all information flow related assertions, i.e., method specifications and loop invariants, purely as relational specifications in terms of *low* and *lowEvent* assertions (see Table 3.1). For all these examples, we completely avoid the need to specify the functional behavior of the program.

The only exception is an example that, depending on a high input, executes different loops with identical behavior, and for which we need to prove that the execution time is low. In this case we have to provide invariants for both loops that exactly specify their execution time in order to prove that the overall execution time after the conditional is low. Nevertheless, the specification of the method containing the loop is again expressed with a relational specification using only *low*. For all other examples, unary specifications were only needed to verify the absence of runtime errors (e.g., out-of-bounds array accesses), which Viper verifies by default. Consequently, a verified program cannot leak low data through such errors, which is typically not guaranteed by type systems or static analyses.

Table 3.1.: Evaluated examples for secure information flow. We show the used language features (Ev. = observable events, Hp. = mutable heap, Arr. = arrays, Decl. = declassification, Call = method calls) and proved properties (Ter. = absence of termination channels, Tm. = absence of timing channels), lines of code including specifications (LOC), overall lines used for specifications (Ann), unary specifications for safety (SF), relational specifications for non-interference (NI), specifications for termination (TM), and functional specifications required for non-interference (F). Note that some lines contain specifications belonging to multiple categories. Columns T_{SE} and T_{VCG} show the running times of the verifiers for the SE backend and the VCG backend, respectively, in seconds.

File	Ev.	Hp.	Arr.	Decl.	Call	Ter.	Tm.	LOC	Ann/SF/NI/TM/F	T_{VCG}	T_{SE}
antopolous1 [8]							x	25	7/3/3/0/2	0.78	1.10
antopolous2 [8]				x			x	61	14/0/14/0/0	0.72	0.91
banerjee [16]		x		x	x			76	17/11/6/0/0	1.02	0.61
constanzo [51]	x		x					22	7/2/5/0/0	0.67	0.28
darvas [56]		x		x				33	12/8/4/0/0	0.67	0.35
example			x		x			31	7/1/6/0/0	0.73	0.59
example_decl			x	x				19	5/2/3/0/0	0.72	0.77
example_term				x		x		31	8/4/2/2/0	0.77	0.43
example_time	x			x	x		x	32	9/0/9/0/0	0.70	0.38
joana_1_tl [89]	x			x	x			28	1/0/1/0/0	0.62	0.23
joana_2_bl [89]	x				x			18	2/0/2/0/0	0.63	0.25
joana_2_t [89]	x							15	1/0/1/0/0	0.62	0.20
joana_3_bl [89]	x			x	x	x		47	5/1/2/2/0	0.77	0.47
joana_3_br [89]	x			x	x	x		43	8/0/2/6/0	0.83	0.60
joana_3_tl [89]	x				x	x		33	8/2/2/4/0	0.75	0.53
joana_3_tr [89]	x			x	x	x		35	8/4/2/2/0	0.76	0.51
joana_13_l [89]					x			12	1/0/1/0/0	0.62	0.24
kusters [117]		x			x			29	9/6/3/0/0	0.64	0.44
naumann [164]		x	x					20	6/3/6/0/0	0.81	0.88
product [23]		x	x		x			65	30/21/21/0/0	5.47	15.73
smith [204]			x	x				43	12/6/8/0/0	0.87	0.89
terauchi1 [211]								14	2/0/2/0/0	0.62	0.26
terauchi2 [211]				x	x			21	4/0/4/0/0	0.63	0.30
terauchi3 [211]								24	5/1/4/0/0	0.66	0.40

3.6.2.2. Performance

For all but one example, the runtime (averaged over 10 runs on a Lenovo ThinkPad T450s with an Intel Core i7-5600U CPU running at 2.6 GHz base and 3.2 GHz turbo frequency with 12GB RAM on Ubuntu) with both the Symbolic Execution (SE) and the Verification Condition Generation (VCG) verifiers is under or around one second (see Table 3.1). The one exception, which makes extensive use of unbounded heap data structures, takes ca. five seconds when verified using VCG, and 15 in the SE verifier. This is likely a result of inefficiencies in our encoding: As noted before, the created product has a high number of branching instructions, and some properties have to be proved more than once, two issues which have a much larger performance impact for SE than for VCG. We believe that it is feasible to remove much of this overhead by optimizing the encoding; we describe some possible optimizations in the next chapter and leave the rest as future work.

3.6.3. Evaluation: Other Hyperproperties

For the second part of our evaluation, we evaluated our approach and implementation for other hyperproperties and compared them to Descartes, a specialized tool by Sousa and Dillig that automates a relational logic and handles loops by guessing and checking possible invariants [206], as well as Synonym, a version of Descartes by Pick et al. with additional optimizations to speed up the verification process [175].

We considered 34 Java implementations of comparators, taken from the evaluation of Descartes. The examples were originally taken from posts on websites like *Stackoverflow*, where developers asked questions about buggy comparators and other users responded with proposed fixes. Additionally, we considered all eight correct *expanded* examples from the

`https://github.com/marcelosousa/descartes`

[206]: Sousa et al. (2016), 'Cartesian Hoare logic for verifying k-safety properties'

`https://github.com/lmpick/synonym`

[175]: Pick et al. (2018), 'Exploiting Synchrony and Symmetry in Relational Verification'

```
public class AInt
    implements Comparator<AInt>{
  int length;
  int get(int pos) { ... }

  public int compare(AInt o1, AInt o2){
    int index, aentry, bentry;
    index = 0;
    while ((index < o1.length) &&
          (index < o2.length)) {
      aentry = o1.get(index);
      bentry = o2.get(index);
      if (aentry < bentry) {
        return −1;
      }
      if (aentry > bentry) {
        return 1;
      }
      index++;
    }
    return 0;
  }
}
```

```
public class AInt
    implements Comparator<AInt>{
  int length;
  int get(int pos) { ... }

  public int compare(AInt o1, AInt o2){
    int index, aentry, bentry;
    index = 0;
    while ((index < o1.length) &&
          (index < o2.length)) {
      aentry = o1.get(index);
      bentry = o2.get(index);
      if (aentry < bentry) {
        return −1;
      }
      if (aentry > bentry) {
        return 1;
      }
      index++;
    }
    if (o1.length < o2.length) {
      return −1;
    }
    if (o1.length > o2.length) {
      return 1;
    }
    return 0;
  }
}
```

Figure 3.10.: Example of a faulty comparator implementation (left) and a fixed version (right) [206]. The left implementation fulfills properties P1 and P2 but violates property P3, the right version fulfills all three properties.

evaluation of Synonym; these are variations of correct comparators from Descartes that were artificially made more complex by Pick et al. Instead of comparing two objects, they take three objects and decide which one is the greatest according to the defined comparison metric. These implementations are generally both longer than the original versions (the longest, PokerHand, is 297 LOC long in the original Java version) and contain more branching, and can thus be used to test the scalability of our verification approach. Since the examples are derived from real-world code, they have a number of challenging properties: Many of them call the compare methods of other classes, many have deeply nested conditional structures, and many loop over some data structure on both objects to compare them.

Comparators have a single method compare that has to fulfill, among others, the following hyperproperties:

- P1: $\forall x, y.\, sgn(\text{compare}(x, y)) = -sgn(\text{compare}(y, x))$
- P2: $\forall x, y, z.\, \text{compare}(x, y) > 0 \wedge \text{compare}(y, z) > 0 \Longrightarrow \text{compare}(x, z) > 0$
- P3: $\forall x, y, z.\, \text{compare}(x, y) = 0 \Longrightarrow sgn(\text{compare}(x, z)) = sgn(\text{compare}(y, z))$

where *sgn* denotes the sign of an integer.

For the modified pick methods, we want to check the following hyperproperties (retaining the naming by Pick et al.):

- P13: $\forall x, y, z.\, pick(x, y, z) = pick(y, x, z)$
- P14: $\forall x, y, z.\, pick(x, y, z) = pick(y, x, z) \wedge pick(x, y, z) = pick(z, y, x)$

Since all implementations have the signatures int compare(T x, T y) and int pick(T x, T y, T z), respectively, where T is the class whose objects are being compared, these properties can be expressed as follows in our specification language (using *res* to refer to the result of the method):

- P1: $\{\{x^{(1)} = y^{(2)} \land y^{(1)} = x^{(2)}\}\}\mathsf{compare}\{\{sgn(res^{(1)}) = -sgn(res^{(2)})\}\}_2$
- P2: $\{\{y^{(1)} = x^{(2)} \land x^{(3)} = x^{(1)} \land y^{(3)} = y^{(2)} \land res^{(1)} > 0 \land res^{(2)} > 0)\}\}\mathsf{compare}\{\{res^{(3)} > 0\}\}_3$
- P3: $\{\{x^{(1)} = x^{(2)} \land y^{(1)} = x^{(3)} \land y^{(3)} = y^{(2)} \land res^{(1)} = 0\}\}\mathsf{compare}\{\{sgn(res^{(2)}) = sgn(res^{(3)})\}\}_3$
- P13: $\{\{true\}\}\mathsf{pick}\{\{(x^{(1)} = y^{(2)} \land x^{(2)} = y^{(1)} \land z^{(1)} = z^{(2)}) \Rightarrow res^{(1)} = res^{(2)}\}\}_s$
- P14: $\{\{true\}\}\mathsf{pick}\{\{((x^{(1)} = y^{(2)} \land x^{(2)} = y^{(1)} \land z^{(1)} = z^{(2)}) \lor (x^{(1)} = z^{(2)} \land y^{(1)} = y^{(2)} \land z^{(1)} = x^{(2)})) \Rightarrow res^{(1)} = res^{(2)}\}\}_s$

For each comparator example, Sousa and Dillig provide a version of the comparator that was initially reported as broken, as well as (both successful and incorrect) attempts at fixed versions. Like them, we check all three hyperproperties for all versions of the examples. Fig. 3.10 shows an example of one of the simpler faulty comparator implementations from the data set as well as a fixed version. For the modified examples by Pick et al., we verify both properties only for the correct implementations.

Our encoding is based on the examples as encoded by Sousa and Dillig and Pick et al., since the original versions of some of the examples are no longer available. In particular, this means that we adopt some simplifications performed by them; however, where we could find the original versions, we tried to stay as close to them as possible. As an example, in several cases, Sousa and Dillig exchanged loops that iterate over a list by loops with a statically fixed number of iterations, and with the current element in each iteration being the result of calling an unspecified function; we reverted this change.

Since the examples were originally written in Java, they contain return statements, which do not exist in the Viper language. We manually emulate them in the standard way by declaring an output parameter res as well as a boolean flag returned, which is initially not set. return statements are then encoded by assigning the returned expression to the result variable, setting the returned flag, and making subsequent statements conditional on returned not being set. Additionally, we add the conjunct ¬returned to the guards of loops whose bodies contain return statements.

As before, we manually added pre- and postconditions as well as loop invariants to all examples.

3.6.3.1. Qualitative Evaluation

Our implementation returns the correct results (i.e., verifies correct implementations and shows errors for incorrect ones) for all of the original comparator examples (see Table 3.2). In that, it behaves identically to Descartes. For the modified examples, our implementation can correctly verify all examples with the VCG backend, but fails to verify four cases with the SE backend due to timeouts. Synonym times out on one example and is unable to infer sufficient invariants in two other cases, and Descartes times out seven times and does not find sufficient invariants in one case. Note, however, that the feature set of the different tools is quite different and comparing the performance can therefore only give an indication of their respective scalability; we discuss the differences in detail in the next section.

In all cases, we were able to use purely relational specifications for other compare methods called by the comparators to be verified. This is vital, since doing otherwise would require information about the precise behavior and therefore the contents and internal data structures of referenced classes (some of which are presumably quite complex), which clearly violates information hiding. By using relational specifications, we were able to prove the desired hyperproperties of each comparator only assuming that the same hyperproperties hold for called comparators of other classes.

For loop invariants, we followed a mixed approach for specifications. For proving P1, we had to use only simple, relational loop invariants. For P2 and P3, where $k = 3$, we opted to use unary loop invariants (i.e., full functional specifications of the loop behavior) for some loops with simple functional behavior. The main reason for this choice was that the proof would otherwise have required invariants that describe both the relative behavior of all three executions if they all execute the loop, *and* the relative behavior of any pair of executions if two executions reach the loop and one does not. This is therefore an example where some of the used specifications are relational but not all executions are relevant for them.

As a result, relational loop invariants would in these cases have been more verbose and arguably more complex. Additionally, we subjectively found it non-trivial to find the required relational loop invariants in some cases: Unlike in the case of secure information flow, where relational invariants translate straightforwardly to high-level concepts like some data being low, finding relational invariants for properties 2 and 3 actually required thinking in terms of multiple executions and was thus more complex. We proved properties P13 and P14 on the modified examples also using a combination of relational of unary specifications for similar reasons.

Nevertheless, we benefited from the ability to use relational loop invariants in several cases. In particular, in examples where other comparators are called inside loop bodies, using functional specifications in the invariant is not possible without also requiring them for the called comparator, which, as pointed out before, is undesirable and not modular. In one example, a loop called another comparator under a condition and performed some simple computation by itself under a different condition; in this case we found it easiest to use a mixed specification that describes the behavior of the comparator in relational terms and the alternative functional behavior in unary terms.

3.6.3.2. Performance

Table 3.2 shows the timings measured to verify all original comparator examples using Viper's VCG backend, its SE backend, and the Descartes tool by Sousa and Dillig. The times for the modified examples are shown in Table 3.3. All times were measured under the same conditions as the timings for secure information flow. In particular, our timings for Descartes and Synonym were measured using the version of the tools currently on Github; in virtually all cases, they are similar to the timings reported by Sousa and Dillig [206] and Pick et al. [175], respectively.

[206]: Sousa et al. (2016), 'Cartesian Hoare logic for verifying k-safety properties'

[175]: Pick et al. (2018), 'Exploiting Synchrony and Symmetry in Relational Verification'

Table 3.2.: Evaluated examples of Java comparators. We show the verification outcomes for all three hyperproperties for the original version(s) and the fixed versions (marked with †) of each example. Columns T_{VCG}, T_{SE}, and T_{DC} show the running times of the verifiers for the VCG backend, the SE backend, and Descartes, respectively, in seconds.

	P1				P2				P3			
	V	T_{VCG}	T_{SE}	T_{DC}	V	T_{VCG}	T_{SE}	T_{DC}	V	T_{VCG}	T_{SE}	T_{DC}
ArrayInt	✓	1.22	1.23	0.06	✓	1.65	21.11	0.06	✗	1.79	11.87	0.11
ArrayInt †	✓	1.25	1.52	0.11	✓	1.91	23.21	0.16	✓	3.43	23.43	0.11
CatBPos	✗	1.25	1.09	0.26	✗	1.50	15.39	5.85	✗	1.52	7.90	1.71
Chromosome	✓	1.18	0.91	0.11	✗	1.22	2.48	0.15	✗	1.31	1.08	0.22
Chromosome †	✓	1.25	1.20	0.06	✓	1.61	22.41	2.02	✓	1.39	9.48	0.28
CollItem	✗	1.19	0.43	0.06	✗	1.27	2.21	0.11	✗	1.27	1.85	0.16
CollItem †	✓	1.15	1.94	0.06	✓	1.22	26.89	0.16	✓	1.25	27.13	0.13
Contact	✓	1.38	5.91	0.11	✗	2.30	15.37	0.78	✗	1.92	2.41	1.34
ContainerV1	✗	1.15	0.30	0.06	✗	1.17	1.08	0.03	✗	1.25	0.88	0.06
ContainerV2	✗	1.15	0.31	0.06	✗	1.21	1.34	0.03	✗	1.22	1.02	0.06
Container †	✓	1.19	0.94	0.16	✓	1.31	4.17	2.32	✓	1.25	2.99	0.94
DataPoint	✗	1.30	0.78	0.21	✗	1.63	16.20	0.67	✗	1.61	38.80	0.42
FileItem	✓	1.14	0.63	0.03	✓	1.37	4.58	0.03	✗	1.23	0.72	0.11
FileItem †	✓	1.20	1.52	0.06	✓	1.34	11.29	0.06	✓	1.30	9.44	0.06
IsoSpriteV1	✗	1.16	0.33	0.06	✗	1.19	1.13	0.03	✗	1.21	1.01	0.06
IsoSpriteV2	✗	1.35	0.58	0.31	✗	2.13	32.94	1.22	✓	2.13	168.62	0.12
Match	✗	1.16	0.47	0.04	✓	1.14	5.33	0.03	✗	1.19	1.72	0.06
Match †	✓	1.17	0.96	0.03	✓	1.20	7.16	0.06	✓	1.22	7.29	0.06
NameComparator	✗	1.24	0.74	0.07	✓	1.77	17.17	0.11	✓	1.70	16.44	0.10
NameComparator †	✓	1.31	1.63	0.10	✓	1.66	24.10	0.11	✓	1.66	24.31	0.16
Node	✓	1.13	0.77	0.03	✓	1.16	5.13	0.03	✗	1.20	1.37	0.07
Node †	✓	1.13	0.77	0.03	✓	1.19	4.90	0.06	✓	1.21	5.01	0.06
NzbFile	✗	1.39	0.56	0.11	✓	1.96	121.60	0.21	✓	1.76	108.31	0.11
NzbFile †	✓	1.35	3.05	0.16	✓	2.39	71.57	0.21	✓	2.38	65.09	0.42
PokerHand	✓	1.56	5.83	0.26	✗	9.19	118.18	0.37	✗	8.67	120.81	0.99
PokerHand †	✓	1.62	6.94	0.27	✓	9.15	237.42	0.37	✓	9.09	233.72	0.66
Solution	✓	1.17	0.86	0.16	✓	1.22	4.29	0.37	✗	1.24	1.25	0.52
Solution †	✓	1.16	0.96	0.31	✓	1.29	4.84	0.97	✓	1.23	4.53	0.92
TextPosition	✓	1.24	1.84	0.01	✗	1.37	7.77	0.01	✗	1.43	6.87	0.01
TextPosition †	✓	1.28	1.99	0.11	✓	1.33	26.29	0.37	✓	1.42	24.98	0.16
Time	✗	1.15	0.43	0.11	✓	1.34	6.23	0.32	✓	1.21	3.80	0.02
Time †	✓	1.11	0.45	0.06	✓	1.28	3.92	0.26	✓	1.19	1.90	0.16
Word	✗	1.30	0.61	0.23	✗	1.60	3.73	4.43	✓	1.65	34.68	0.03
Word †	✓	1.26	1.54	0.11	✓	1.65	23.93	0.17	✓	1.77	24.41	0.11

We first observe that our implementation generally achieves good performance when using the VCG backend. The average verification time for the original comparators is under two seconds for all three properties. With maximal verification times of under ten seconds even for the most complex comparator examples and 18 seconds for the most complex modified example, we conclude that our technique can be used to verify hyperproperties of realistic code in very reasonable time.

When using our tool with Viper's SE backend, we observe considerably worse performance than with VCG. While the average verification times for the original comparators are comparable for the first property, the SE backend performs much worse for the two 3-safety hyperproperties, particularly for the examples that have deeply nested conditionals, where the worst case is an example taking almost four minutes to verify. For the larger modified examples, performance becomes considerably worse; two of the examples do not verify within one hour. This confirms our earlier suspicion that the performance with SE (which considers each branch separately) is impaired by the large amount of branching in modular product programs: Since our product transformation introduces many branches into the program (more for $k = 3$ than for $k = 2$), the SE backend has to consider a large number of possible paths through the program. While the SE backend struggles with this, the VCG backend is able to exploit the fact that many of those branches are on the same or related conditions, and the actual number of feasible paths through the program is therefore much smaller than it seems.

We can now compare the performance of our implementation to that of Descartes and Synonym. While this leads to some interesting conclusions,

Table 3.3.: Evaluated modified examples. All examples fulfill both properties and verify successfully in every configuration for which a time is shown. Columns T_{VCG}, T_{SE}, T_{DC}, and T_{SY} show the running times of the verifiers for the VCG backend, the SE backend, Descartes, and Synonym, respectively, in seconds. "TO" indicates that the test times out after one hour, "-" means that the tool reports that it could not infer sufficient invariants.

	P13				P14			
	T_{VCG}	T_{SE}	T_{DC}	T_{SY}	T_{VCG}	T_{SE}	T_{DC}	T_{SY}
ArrayInt-pick3-simple †	3.49	18.08	2.16	0.82	3.48	22.60	TO	292.90
ArrayInt-pick3 †	3.53	51.95	2.03	1.01	3.64	58.95	TO	210.71
Chromosome-pick3-simple †	3.52	55.65	1.12	0.68	3.47	78.28	TO	177.69
Chromosome-pick3 †	3.66	28.12	3.15	1.97	3.63	34.02	TO	980.77
PokerHand-pick3-part1 †	4.61	202.41	6.96	2.86	5.19	188.74	TO	1548.27
PokerHand-pick3-part2 †	7.14	1082.14	11.90	5.18	9.73	1125.56	TO	-
PokerHand-pick3 †	12.05	TO	20.63	8.63	18.07	TO	-	-
Solution-pick3 †	4.79	TO	84.92	19.01	5.28	TO	TO	TO

we stress that the feature sets supported by the tools are quite different. Viper is a mature, general purpose verification tool with built-in support for mutable heap data structures and several basic datatypes; it performs several additional tasks like type checking, well-definedness checking of specifications, and proving memory safety, all of which are not performed by Descartes and Synonym. The latter, on the other hand, are specialized prototype implementations that do not support heap mutation or data types beyond integers, but, unlike Viper, are able to automatically infer loop invariants via a guess-and-check-approach.

Compared to Descartes and Synonym, our implementation using the VCG backend is generally slower for small examples (the original comparators) but faster for the more complex modified examples; in fact, all verifiers except for VCG time out on at least one example. This is likely at least partly due to the aforementioned differences between the tools; while Viper has a higher constant overhead for each program due to startup time and additional checks, the additional work needed by Descartes and Synonym for guessing and checking loop invariants becomes more of a factor for more complex examples. SE has a lower constant overhead than VCG and is more competitive than VCG on small examples (though still slower than Descartes and Synonym) but scales much worse in most cases and is thus the slowest tool for most (though not all) of the more complex examples.

In the VCG backend, each verification task requires starting a .NET runtime for running the Boogie [18] verifier in the background, which leads to some constant amount of overhead.

We conclude that while our technique creates programs that lead to bad performance with some verification techniques (SE), it can be combined with other standard verification techniques (in particular, VCG) to verify arbitrary hyperproperties of real-world code in reasonable time. Compared to existing more automated tools, our technique delivers better performance on very complex examples when used with VCG. In particular, since our approach allows modular (and therefore independent) verification of different methods, we expect it to scale to large programs.

One advantage of using a program transformation approach for verifying hyperproperties is that one can freely choose which tool to use to reason about the resulting program, based on the desired level of automation and required language features. In contrast, adding mutable heap support to Descartes would require changes to the underlying logic itself.

3.7. Related Work

The notion of k-safety hyperproperties was originally introduced by Clarkson and Schneider [46]. Here, we focus on statically proving hyperproperties for imperative and object-oriented programs; more work exists for testing or monitoring hyperproperties like non-interference at runtime, or for reasoning about hyperproperties in different programming paradigms.

[46]: Clarkson et al. (2010), 'Hyperproperties'

Relational logics such as relational Hoare logic [30], relational separation logic [231] and others [2, 26] allow reasoning directly about relational properties of two different program executions. Beringer [32] extends relational Hoare logic with rules for dissonant loops that allow reasoning in a similar way as our product construction. Unlike our approach, relational logics usually allow reasoning about the executions of two *different* programs; as a result, they do not give special support for two executions of the same program calling the same method with a relational specification.

[30]: Benton (2004), 'Simple relational correctness proofs for static analyses and program transformations'

[231]: Yang (2007), 'Relational separation logic'

[2]: Aguirre et al. (2017), 'A relational logic for higher-order programs'

[26]: Barthe et al. (2009), 'Formal certification of code-based cryptographic proofs'

[32]: Beringer (2011), 'Relational Decomposition'

Banerjee et al. [17] introduced biprograms, which allow explicitly expressing alignment between executions and using relational specifications to reason about aligned calls; however, this approach requires that methods with relational specifications are always called by both executions, which is for instance not the case if a call occurs under a high guard in secure information flow verification. We handle such cases by interpreting relational specifications as trivially true; one can then still resort to functional specifications to complete the proof. Their work also does not allow mixed specifications, which are easily supported in our product programs.

[17]: Banerjee et al. (2016), 'Relational Logic with Framing and Hypotheses'

Relational program logics are generally difficult to automate. Recent work by Sousa and Dillig [206] presents a logic that can be applied automatically by an algorithm that implicitly constructs different product programs that align *some* identical statements, but does not fully support relational specifications. Pick et al. [175] use similar ideas but exploit symmetries in the verification problem and further align the execution of loops for improved performance. Both approaches require dedicated tool support, whereas our modular product programs can be verified using off-the-shelf tools. More recently, Banerjee et al. [14] present a relational logic for biprograms that supports modular reasoning for a rich language including classes, takes into account concepts like encapsulation, and has been automated in a prototype tool based on Why3 [82].

[206]: Sousa et al. (2016), 'Cartesian Hoare logic for verifying k-safety properties'

[175]: Pick et al. (2018), 'Exploiting Synchrony and Symmetry in Relational Verification'

[14]: Banerjee et al. (2022), 'A Relational Program Logic with Data Abstraction and Dynamic Framing'

[82]: Filliâtre et al. (2013), 'Why3 - Where Programs Meet Provers'

The approach of reducing hyperproperties to ordinary trace properties was introduced by self-composition [25]. While self-composition is theoretically complete, it does not allow modular reasoning with relational specifications; this distinction has recently been examined and formalized by Nagasamudram and Naumann [163]. The resulting problem of having to fully specify program behavior was pointed out by Terauchi and Aiken [211]; since then, there have been a number of different attempts to solve this problem by allowing (parts of) programs to execute in lock-step. Terauchi and Aiken [211] did this for secure information flow by relying on information from a type system; other similar approaches exist [164].

[25]: Barthe et al. (2011), 'Secure information flow by self-composition'

[163]: Nagasamudram et al. (2021), 'Alignment Completeness for Relational Hoare Logics'

[211]: Terauchi et al. (2005), 'Secure Information Flow as a Safety Problem'

[211]: Terauchi et al. (2005), 'Secure Information Flow as a Safety Problem'

[164]: Naumann (2006), 'From Coupling Relations to Mated Invariants for Checking Information Flow'

Product programs [23, 24] allow different interleavings of program executions. The initial product program approach [23] would in principle

[23]: Barthe et al. (2011), 'Relational Verification Using Product Programs'

[24]: Barthe et al. (2013), 'Beyond 2-Safety: Asymmetric Product Programs for Relational Program Verification'

[23]: Barthe et al. (2011), 'Relational Verification Using Product Programs'

allow the use of relational specifications for method calls, but only under the restriction that both program executions *always* follow the same control flow. The generalized approach [24] allows combining different programs and arbitrary numbers of executions. This product construction is non-deterministic and usually interactive; in some (but not all) cases, it allows avoiding duplicated calls and loops and thereby using relational specifications. However, the construction requires manual work by the programmer, and whether the product can be constructed in such a way that call is not duplicated depends on the used specification, meaning that the product construction and verification are intertwined and a new product has to be constructed when specifications change. In contrast, our product construction is fully deterministic and automatic, allows arbitrary control flows while still being able to use relational specifications for all loops and calls, and therefore avoids the issue of requiring full functional specifications.

[24]: Barthe et al. (2013), 'Beyond 2-Safety: Asymmetric Product Programs for Relational Program Verification'

While modular product programs always align the same program points in different executions (e.g., relational loop invariants always relate the state after the same iteration of the same loop in different executions), both Shemer et al. [197] and Farzan and Vandikas [78] recently proposed techniques for automatically finding different alignments that allow proving a specific hyperproperty using simpler relational invariants. While the details differ, both techniques, when given a program, a hyperproperty to be proves, and a set of assertions that may be used in the proof, iteratively refine an alignment of different executions using counterexamples until they either find an alignment that provably fulfills the given property or they conclude that no proof is possible with the given set of assertions.

[197]: Shemer et al. (2019), 'Property Directed Self Composition'

[78]: Farzan et al. (2019), 'Automated Hypersafety Verification'

Techniques for proving program equivalence, whose goal is to prove relational properties about pairs of programs, can be used to prove hyperproperties of a single program as well. In particular, Hawblitzel et al. [98] and Lahiri et al. [119] propose two different techniques for modularly proving and using *mutual summaries* that relate the behavior of two different methods. Compared to our work, both techniques have the advantage of being able to relate arbitrary pairs of method calls. In the former approach [98], this is achieved by axiomatically assuming mutual summaries for any pair of calls and subsequently generating verification conditions that check the mutual summary of one pair of methods assuming these axioms for all calls. In contrast to modular product programs, this approach does not allow checking relational *preconditions*, and it does not work with standard static analysis tools, since those typically cannot make use of the axioms that are used to assume relational postconditions. In the latter approach [119], mutual summaries are used via a product construction that stores the local and global state before and after each method call in auxiliary variables, sequentially composes the bodies of both different methods, and subsequently assumes or asserts relational properties about the stored information for each pair of related calls. As a result, the technique is limited in the presence of non-termination (since no relational properties are checked if one of the programs does not terminate) and, for example, cannot support reasoning about termination channels. Importantly, both of the aforementioned approaches require that dependencies on global variables are statically known and finite (since their values need to be mentioned in axioms

[98]: Hawblitzel et al. (2013), 'Towards Modularly Comparing Programs Using Automated Theorem Provers'

[119]: Lahiri et al. (2013), 'Differential assertion checking'

[98]: Hawblitzel et al. (2013), 'Towards Modularly Comparing Programs Using Automated Theorem Provers'

[119]: Lahiri et al. (2013), 'Differential assertion checking'

or stored for each call, respectively) and therefore do not easily extend to programs with dynamic heap allocation and to separation logics, unlike modular product programs. Additionally, both approaches rely on eliminating loops by transforming them to tail recursion. This is always possible but can require additional specifications from the user to carry information about local variables into method calls inserted by the transformation. Felsing et al. [80] present a weakest precondition calculus for showing (conditional) program equivalence as well as an encoding into Horn clauses for automation. Their approach to loop verification is based on similar ideas as our loop encoding and they also support mutual function summaries. In contrast to our approach, Felsing et al. consider only terminating programs in a language without arrays or a mutable heap, and their approach requires dedicated tool support. Similar ideas for loop fusion have been used for other purposes, e.g., to simplify loop invariants in a single program and for program optimization [107].

[80]: Felsing et al. (2014), 'Automating regression verification'

[107]: Imanishi et al. (2018), 'A guess-and-assume approach to loop fusion for program verification'

The equivalence checking approach by Hawblitzel et al. [98] also supports checking *relative termination* under some relational precondition, which is similar but not identical to the absence of termination channels; it asserts that if one execution terminates, the other *can* terminate as well. The presented approach requires fewer annotations than ours, in particular, it does not require ranking functions, but as a result it is less precise and does not allow any loops or method calls under high conditions. Similarly, Elenbogen et al. [70] provide an algorithm for proving *mutual termination*, which can be used for proving the absence of termination channels, via a product construction. It, too, is less precise than our approach but requires less user input.

[98]: Hawblitzel et al. (2013), 'Towards Modularly Comparing Programs Using Automated Theorem Provers'

[70]: Elenbogen et al. (2015), 'Proving mutual termination'

Reducing control flow to straight-line code with conditional statements, also called *predicated execution*, is sometimes performed as a performance optimization to avoid branching. The principle is also employed in modern GPUs, which execute multiple threads at once in a SIMD fashion, and implement diverging control flow by deactivating some threads while a branch they do not take is executed. Betts et al. [34] exploit this and present an encoding similar to our product construction to prove trace properties of (concurrently executed) GPU kernels, but do not explore its application to hyperproperties of sequential programs. Collingbourne et al. [49] build on this work to present an encoding for GPU kernels with arbitrary reducible control flow graphs; the main ideas behind this extended encoding could likely also be used to create modular product programs based on control flow graphs.

[34]: Betts et al. (2012), 'GPUVerify: a verifier for GPU kernels'

[49]: Collingbourne et al. (2013), 'Interleaving and Lock-Step Semantics for Analysis and Verification of GPU Kernels'

Considerable work has been invested into proving specific hyperproperties like secure information flow. One popular approach is the use of type systems [204]; while those are modular and offer good performance, they overapproximate possible program behaviors and are therefore less precise than approaches using logics. In particular, they require labeling any single value as either high or low, and do not allow distinctions like the one we made for the example in Fig. 3.1, where only the first bits of a sequence of integers were low. In addition, type systems typically struggle to prevent information leaks via side channels like termination or program aborts. There have been attempts to create type systems that handle some of these limitations (e.g. [57]).

[204]: Smith (2007), 'Principles of Secure Information Flow Analysis'

[57]: Deng et al. (2004), 'Lenient Array Operations for Practical Secure Information Flow'

Static analyses [8, 89] enable fully-automatic reasoning. They are typically

[8]: Antonopoulos et al. (2017), 'Decomposition instead of self-composition for proving the absence of timing channels'

[89]: Giffhorn et al. (2015), 'A new algorithm for low-deterministic security'

not modular and, similarly to type systems, need to abstract semantic information, which can lead to false positives. They strike a trade-off different from our solution, which requires specifications, but enables precise, modular reasoning.

A number of logic-based approaches to proving specific hyperproperties exist. As an example, Darvas et al. use dynamic logic for proving noninterference [56]; this approach offers some automation, but requires user interaction for most realistic programs. Leino et al. [131] verify determinism up to equivalence using self-composition, which suffers from the drawbacks explained above. Prabawa et al. [179] present a logic for proving secure information flow. While their approach, like ours, enables modular verification, it can only track secrecy on a more coarse-grained level, does not take into account semantic information, and will therefore produce more false alarms.

[56]: Darvas et al. (2005), 'A Theorem Proving Approach to Analysis of Secure Information Flow'

[131]: Leino et al. (2008), 'Verification of Equivalent-Results Methods'

[179]: Prabawa et al. (2018), 'A Logical System for Modular Information Flow Verification'

The approach of declassifying data via special statements or expressions has been introduced by Sabelfeld and Myers [186]. An extended version of JML by Scheben and Schmitt [192] supports declassification expressions as parts of method contracts that can be verified using self-composition. Other kinds of declassification have been studied extensively; Sabelfeld and Sands [188] provide a good overview. Li and Zdancewic [136] introduce downgrading policies that describe which information can be declassified and, similar to our approach, can do so for arbitrary expressions. Costanzo and Shao [51] define a similar system that allows users to define program preconditions to describe which parts of the secret data may be released. Our approach allows for similar specifications that describe that some aspects of some data are low (and everything else is high by default).

[186]: Sabelfeld et al. (2003), 'A Model for Delimited Information Release'

[192]: Scheben et al. (2011), 'Verification of Information Flow Properties of Java Programs without Approximations'

[188]: Sabelfeld et al. (2005), 'Dimensions and Principles of Declassification'

[136]: Li et al. (2005), 'Downgrading policies and relaxed noninterference'

[51]: Costanzo et al. (2014), 'A Separation Logic for Enforcing Declarative Information Flow Control Policies'

Since their publication, modular product programs have been used to automatically *infer* relational specifications and prove hyperproperties in different ways. Pick et al. [176] use Syntax-Guided Synthesis (SyGuS) [6] to infer information flow specifications, exploiting that they often have simple syntactic shapes. They enumerate possible candidate specifications, including ones that include quantifiers to specify precise properties of arrays, and check them using a CHC solver on an encoding of the original program into its modular product program. Knabenhans [116] uses abstract interpretation [52] and relational numerical domains [53, 153] on modular product programs to infer relational properties of the original program. His analysis uses trace partitioning [145] to analyze traces with different values of activation variables separately; this solves the problem that the control flow of a modular product program essentially merges the control flow of different executions, including the case where no statements at all are executed (when all activation variables are false). An analysis that considers all possible control flows at once has to continually join the states of all possible paths and will be unable to gain any information. Building on this analysis, Blarer [36] uses modular product programs to statically analyze GPU kernels for potential performance problems, exploiting the close relationship between modular product programs and predicated execution used in GPUs mentioned above.

[176]: Pick et al. (2020), 'Automating Modular Verification of Secure Information Flow'

[6]: Alur et al. (2013), 'Syntax-guided synthesis'

[116]: Knabenhans (2018), 'Automatic Inference of Hyperproperties'

[52]: Cousot et al. (1977), 'Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints'

[53]: Cousot et al. (1978), 'Automatic Discovery of Linear Restraints Among Variables of a Program'

[153]: Miné (2006), 'The octagon abstract domain'

[145]: Mauborgne et al. (2005), 'Trace Partitioning in Abstract Interpretation Based Static Analyzers'

[36]: Blarer (2019), 'Static Analysis of GPU Kernel Performance Hyperproperties'

3.8. Conclusion

We have presented modular product programs, a novel form of product programs that enable modular reasoning about k-safety hyperproperties using relational specifications with off-the-shelf verifiers. We have proved our approach sound and complete for arbitrary values of k. We showed that modular products are expressive enough to handle advanced aspects of secure information flow verification. They can prove the absence of termination and timing side channels and encode declassification. Our implementation shows that our technique works in practice on a number of challenging examples from both the literature and real user code, and exhibits good performance even without optimizations.

Product Programs in IVL-Based Verifiers 4.

As explained in Chapter 1, in recent years, many automated and expressive verification tools have been developed for a wide range of real-world programming languages. However, most of these tools are limited to trace properties (properties of single program traces) and cannot prove hyperproperties such as non-interference. Building new verifiers with support for non-interference from scratch can take years when targeting substantial subsets of real-world programming languages.

In principle, existing program verifiers can be used to verify hyperproperties by reducing them to trace properties via self-composition [25] or product programs [23, 24] like the product construction shown in the previous chapter. However, this approach does not easily extend to real-world programming languages: Many product constructions [23, 119], including modular product programs, have been defined only for simple languages that lack many language features found in real world languages (e.g., dynamic method binding and concurrency); others require manual work for the product construction [24]. While there is one product construction we are aware of that applies to concurrent programs [78], this construction does not allow for thread-modular verification and therefore likely does not scale to large programs; nor does self-composition, which can handle arbitrary programming languages in principle, but allows neither thread-modular nor method-modular verification. As a result, applying product constructions to programs written in complex languages would require defining and implementing new and complex product constructions for every new verifier.

In this chapter, which is based on the CAV 2021 paper "Product Programs in the Wild: Retrofitting Program Verifiers to Check Information Flow Security" [66], we address the problem of retroactively enabling an existing program verifier to check non-interference by using an existing product construction. To do this, we leverage the fact that most automatic deductive verifiers share a similar architecture explained in Chapters 1 and 2: They consist of a custom frontend, which encodes a source program into an intermediate verification language (IVL), and a reusable backend, which verifies the IVL program using generic proof search engines. Boogie [18], Viper [159], and Why3 [82] are examples of such IVLs, which power a large number of program verifiers; for instance Boogie is used by Dafny [127], VCC [48], Spec# [130], and GPUVerify [34], Why3 [82] by Frama-C [54] and Krakatoa [81], and Viper [159] by VerCors [38], Prusti [12], Gobra [228], and of course Nagini. The ubiquitiy of this architecture offers a chance to retrofit existing verifiers to check non-interference by performing the product construction on the level of the IVL (an approach that is already used by SymDiff [118] for the related problem of program equivalence). The resulting architecture, which allows one to reuse both the frontend and the backend of the existing verifier, is shown in Figure 4.1.

Performing the product construction on the IVL level has three major advantages over a product construction on the source program: (1) It cleanly separates the encoding of the source language (which tends to

[25]: Barthe et al. (2011), 'Secure information flow by self-composition'

[23]: Barthe et al. (2011), 'Relational Verification Using Product Programs'

[24]: Barthe et al. (2013), 'Beyond 2-Safety: Asymmetric Product Programs for Relational Program Verification'

[23]: Barthe et al. (2011), 'Relational Verification Using Product Programs'

[119]: Lahiri et al. (2013), 'Differential assertion checking'

[24]: Barthe et al. (2013), 'Beyond 2-Safety: Asymmetric Product Programs for Relational Program Verification'

[78]: Farzan et al. (2019), 'Automated Hypersafety Verification'

[66]: Eilers et al. (2021), 'Product Programs in the Wild: Retrofitting Program Verifiers to Check Information Flow Security'

[18]: Barnett et al. (2005), 'Boogie: A Modular Reusable Verifier for Object-Oriented Programs'

[159]: Müller et al. (2016), 'Viper: A Verification Infrastructure for Permission-Based Reasoning'

[82]: Filliâtre et al. (2013), 'Why3 - Where Programs Meet Provers'

[127]: Leino (2010), 'Dafny: An Automatic Program Verifier for Functional Correctness'

[48]: Cohen et al. (2010), 'Local Verification of Global Invariants in Concurrent Programs'

[130]: Leino et al. (2008), 'Using the Spec# Language, Methodology, and Tools to Write Bug-Free Programs'

[34]: Betts et al. (2012), 'GPUVerify: a verifier for GPU kernels'

[82]: Filliâtre et al. (2013), 'Why3 - Where Programs Meet Provers'

[54]: Cuoq et al. (2012), 'Frama-C - A Software Analysis Perspective'

[81]: Filliâtre et al. (2007), 'The Why/Krakatoa/Caduceus Platform for Deductive Program Verification'

[159]: Müller et al. (2016), 'Viper: A Verification Infrastructure for Permission-Based Reasoning'

[38]: Blom et al. (2014), 'The VerCors Tool for Verification of Concurrent Programs'

[12]: Astrauskas et al. (2019), 'Leveraging Rust types for modular specification and verification'

[228]: Wolf et al. (2021), 'Gobra: Modular Specification and Verification of Go Programs'

[118]: Lahiri et al. (2012), 'SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs'

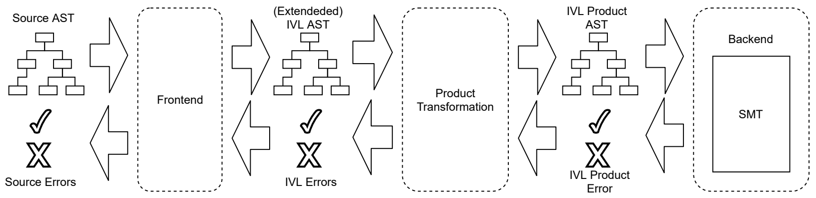


Figure 4.1.: Proposed architecture for information flow verifiers. The existing encoding from source to IVL (frontend) as well as the proof search (backend) can be reused. The product construction needs to support only the (relatively small) IVL and can be reused across different verifiers.

be complex for full-fledged languages) from the product construction. (2) The product construction is much simpler, since IVLs are small, sequential languages; as a result, existing product definitions can be used. (3) The product construction can be reused across all verifiers built on the same IVL. Overall, this architecture therefore has the potential to enable non-interference verification in existing verifiers with substantially less effort than building a new tool from scratch.

Even though this approach has strong advantages, there are several open questions that must be addressed to make it useful and widely applicable:

1. Soundness: Given an IVL encoding and a product construction that are individually sound, is the resulting combination always sound as well?
2. Concurrency: There is a substantial number of verifiers that verify concurrent source programs by encoding them into (sequential) IVLs. Can we soundly verify non-interference of concurrent programs based on a product program of the sequential IVL encoding?
3. Performance: Product constructions are known to cause a performance penalty for verification. Does this overhead prevent the construction of useful verification tools in practice?

In this chapter, we answer these three questions. We focus our investigation on modular product programs, exploiting their usefulness for modular verification demonstrated in the previous chapter, as well as their ability to precisely express complex information flow properties including termination-sensitive non-interference, value-dependent sensitivity [160], and declassification. We will, however, not re-explain these concepts or their encoding in modular product programs, and will focus our presentation on ordinary termination-insensitive non-interference. As in the previous chapter, we will describe our approach in terms of information flow security throughout, but non-interference in general is also relevant to proving integrity properties. To summarize, we make the following contributions:

[160]: Murray et al. (2018), ‘COVERN: A Logic for Compositional Verification of Information Flow Control’

- We show that the combination of sound IVL encodings and sound product constructions can indeed be unsound. We identify a novel condition on IVL encodings that ensures the soundness of the overall workflow (Sec. 4.1).
- We show real-world examples of both sound and unsound encodings, and show how to adjust unsound encodings on the example of a commonly-used encoding for dynamically-bound method calls (Sec. 4.2).
- We show for the common case of data race free programs using locks that it is possible to verify different non-interference properties as well as related properties like observational determinism

for concurrent programs using sequential product programs. Furthermore, we demonstrate that existing criteria for possibilistic non-interference are insufficient in our setting; we provide alternative criteria that are sound and show how to encode them in a product program (Sec. 4.3).

- We implement the approach for Nagini, thus completing its ability to verify the program properties we outlined in Chapter 1 that are relevant for proving security properties of software systems. We evaluate the performance impact of the product construction and show that, while worse than a custom-made information flow verifier [72], performance is acceptable for real-world use, and its expressiveness matches the state of the art (Sec. 4.4).

[72]: Ernst et al. (2019), 'SecCSL: Security Concurrent Separation Logic'

These results demonstrate that the proposed approach can indeed be used to retrofit an existing verifier to *soundly* check non-interference, even for concurrent programs. The resulting version of Nagini, whose extension to non-interference verification took only a fraction of the effort required for the development of a new verifier, can compete with custom-made tools in its expressiveness at an acceptable performance cost.

The remainder of this chapter is organized as follows: Sec. 4.1 explains the proposed verifier architecture in more detail, shows its potential soundness issue, and derives a soundness criterion on the IVL encoding. In Sec. 4.2, we demonstrate the practical relevance of both the soundness issue and the soundness criterion on two real-world examples. Sec. 4.3 shows how the proposed architecture can be used to verify different non-interference properties for concurrent source language programs. We describe and evaluate our implementation in Nagini and Viper in Sec. 4.4, compare to related work in Sec. 4.5 and conclude in Sec. 4.6.

Throughout the chapter, we will re-use definitions and notation from the previous chapter, and we will re-use the color-coding of code listings from Chapter 2, i.e., we will mark source language code (for which we use Python syntax) in blue and IVL code (for which we again use a simplified Viper dialect) in red.

4.1. Sound Products of IVL Encodings

In this section, we address the first question from the introduction, namely, whether we can always soundly combine an existing encoding into an IVL with a sound product construction. We first describe the proposed architecture in greater detail. Then we show a potential soundness issue and define a sufficient criterion on the IVL encoding for the entire approach to be sound.

4.1.1. Proposed Architecture

The architecture proposed in the introduction (Fig. 4.1) enables the construction of information flow aware verifiers with relatively little effort, by reusing most of the frontend encoding of the source language to an IVL as well as the entire backend proof search. The only major change

Figure 4.2.: Example of an encoding that is unsound in our setting. The program on the left can be encoded into a conditional statement (identical to the source program, modulo language syntax) or to the program on the right; the latter leads to unsoundness if P is a relational predicate.

```
def foo(x: int):
  if x > 7:
    y = 5
  else:
    y = 7
  assert P(y)
```

```
method foo(x: Int)
{
  assert x > 7 ? P(5) : P(7)
}
```

that is necessary is that the frontend and the IVL have to be extended to allow for the use of information flow assertions in specifications. Crucially, the frontend does not have to know their meaning; it can treat relational source-level assertions like $low(e)$ or *lowEvent* like ordinary unary predicates and simply translate them to their counterparts $low(e)$ and *lowEvent* on the IVL level. IVL-level relational assertions will then be translated to ordinary assertions during the product transformation.

In the remainder of this chapter, we will generally assume that the existing IVL encoding is used unchanged, and point out when changes need to be made.

4.1.2. Soundness Issue

Surprisingly, combining a sound encoding from source language to IVL with a sound IVL-level product construction may result in a verification technique that is *unsound* in the presence of relational specifications. Consider the source program in Fig. 4.2 (left), where P is some predicate.

A frontend could encode the body of foo into an identical (modulo syntax) conditional statement on the IVL level (assuming the IVL provides conditionals, assignments, and assert statements). Alternatively, it could produce the encoding shown in Fig. 4.2 (right), which directly asserts a sufficient precondition of the source program. If P is a unary predicate, both encodings are sound: If they verify, the original program is correct. However, if P(y) is a relational predicate, for instance, $low(\text{y})$, then the encoding on the right is *unsound*: $low(5)$ and $low(7)$ are trivially true (since $5 = 5$ and $7 = 7$), so the assertion in the encoded program trivially passes, yet the original program is clearly incorrect: If x is greater than 7 in one execution but less in the other, y will have different values in both executions, and will therefore not be low.

The underlying reason is that the encoding on the right does not encode the exact behavior of the source program; it encodes a verification condition computed by the frontend that is sound if assertions are unary, but may not be sound for relational assertions.

We will now (1) formalize this intuition and derive a sufficient condition for the soundness of an encoding in this approach, and (2) show an example of this problem occurring in real frontends, and describe how it can be solved.

4.1.3. Soundness Criterion

We write Σ and S for states and statements of the source language, and σ and s for states and statements of the IVL. States may contain, for example, a mutable heap and a variable store. For simplicity, we assume that both source and IVL statements contain a statement `skip` that represents a finished computation. We also assume that there is a small-step transition relation $\rightarrow$ for both languages, and that the standard notion of Hoare triple validity $\vDash \{P\}s\{Q\}$ is defined for the IVL. We let P and Q range over (source and IVL level) assertions from a standard assertion language extended with $low(e)$ and $lowEvent$, and assume a standard definition of assertion validity for pairs of states.

We define an encoding to be a triple $\langle \alpha, \cong, \beta \rangle$, where α is an encoding function from source statements to statements of the target language (i.e., the IVL), β similarly encodes assertions to the target language, and $\cong$ relates source language states to corresponding target language states.

We first define the desired relational soundness property, which expresses that if an encoded Hoare triple holds for the encoded program, then the original hyperproperty holds for all pairs of executions of the source program:

Definition 4.1.1 $\langle \alpha, \cong, \beta \rangle$ *is* relationally sound *iff, for all* $S, \Sigma_1, \Sigma_2, \Sigma_1', \Sigma_2'$, P, Q, *if* $\vDash \{\lceil \beta(P) \rceil^{\mathring{p}}\} [\![\alpha(S)]\!]^{\mathring{p}} \{\lceil \beta(Q) \rceil^{\mathring{p}}\}$ *and* $\Sigma_1, \Sigma_2 \vDash P$ *and* $\langle S, \Sigma_1 \rangle \rightarrow^* \langle \mathtt{skip}, \Sigma_1' \rangle$ *and* $\langle S, \Sigma_2 \rangle \rightarrow^* \langle \mathtt{skip}, \Sigma_2' \rangle$, *then* $\Sigma_1', \Sigma_2' \vDash Q$.

Product programs represent the operational behavior of two program executions by the operational behavior of a single product program execution. The unsoundness shown before is caused by the fact that the encoding into the IVL does not reflect the operational behavior of the conditional statement (replacing it by an assertion of a sufficient precondition) and, thus, the resulting product does not soundly reflect two executions of the source program.

We call an encoding that preserves the operational behavior of the source program *operational*: It encodes every step of the source program into some number of steps of the target program so that matching initial states result in matching end states. Similarly, it encodes specifications from the source level into target-level specifications that hold in matching states. We can formalize this intuition by requiring that the source and target programs are connected by the simulation relation $\cong$:

Definition 4.1.2 $\langle \alpha, \cong, \beta \rangle$ *is an* operational *encoding if: (1) for all* $\Sigma, \Sigma', \sigma, S, S'$, *if* $\langle S, \Sigma \rangle \rightarrow \langle S', \Sigma' \rangle$ *and* $\Sigma \cong \sigma$, *then* $\langle \alpha(S), \sigma \rangle \rightarrow^* \langle \alpha(S'), \sigma' \rangle$ *for some* σ' *s.t.* $\Sigma' \cong \sigma'$, *and (2) if* $\Sigma \cong \sigma$ *then* $\Sigma \vDash P$ *iff* $\sigma \vDash \beta(P)$.

Note that this notion allows the encoding to overapproximate the behaviors of the source program, i.e., admit steps that are not possible on the source level, but not vice versa. This is crucial, since abstraction and overapproximation are a part of many typical frontend encodings, as explained in Sec. 2.1.3.

For the example in Fig. 4.2, it is easy to see that this criterion is fulfilled by the left encoding: the source and IVL programs are identical (modulo syntax), matching states are identical states (modulo state encodings), and the behavior of both programs is identical. The encoding on the right, however, is *not* operational: While the left program modifies the state, the right program never performs any state modification at all.

We now show that operationality is sufficient for relational soundness:

Theorem 4.1.1 *If $\langle \alpha, \cong, \beta \rangle$ is operational then it is relationally sound.*

Proof. Assume that $\langle S, \Sigma_1 \rangle \longrightarrow^* \langle \texttt{skip}, \Sigma_1' \rangle$ and $\langle S, \Sigma_2 \rangle \longrightarrow^* \langle \texttt{skip}, \Sigma_2' \rangle$ and $\Sigma_1, \Sigma_2 \vDash P$ and $\vDash \{\lceil \beta(P) \rceil^{\mathring{p}}\} [\![\alpha(S)]\!]^{\mathring{p}} \{\lceil \beta(Q) \rceil^{\mathring{p}}\}$. Let $\Sigma_1 \cong \sigma_1$ and $\Sigma_2 \cong \sigma_2$. We have to show that $\Sigma_1', \Sigma_2' \vDash Q$.

By operationality (1), we have that $\langle \alpha(S), \sigma_1 \rangle \longrightarrow^* \langle \alpha(\texttt{skip}), \sigma_1' \rangle$ and $\langle \alpha(S), \sigma_2 \rangle \longrightarrow^* \langle \alpha(\texttt{skip}), \sigma_2' \rangle$ for some σ_1', σ_2' s.t. $\Sigma_1' \cong \sigma_1'$ and $\Sigma_2' \cong \sigma_2'$. By Thm. 3.4.5, we have $\langle [\![\alpha(S)]\!]^{\mathring{p}}, \sigma_1 | \sigma_2 \rangle \longrightarrow^* \langle [\![\alpha(\texttt{skip})]\!]^{\mathring{p}}, \sigma_1' | \sigma_2' \rangle$, where $\sigma_i | \sigma_j$ is a state containing the renamed combination of σ_i and σ_j. By operationality (2), we know $\sigma_1, \sigma_2 \vDash \beta(P)$ and therefore $\sigma_1 | \sigma_2 \vDash \lceil \beta(P) \rceil^{\mathring{p}}$, so because the Hoare triple is valid, we know $\sigma_1' | \sigma_2' \vDash \lceil \beta(Q) \rceil^{\mathring{p}}$ and therefore $\sigma_1', \sigma_2' \vDash \beta(Q)$. By operationality (2), we therefore know $\Sigma_1', \Sigma_2' \vDash Q$. □

Note that operationality is a sufficient but not necessary condition; encodings of verification conditions may be sound for relational verification as well. The main advantage of applying the operationality criterion instead of directly reasoning about relational soundness is that, since operationality represents the simple notion that the IVL program performs equivalent steps and equivalent state changes to the source program, it is intuitive and easy to check whether a given encoding is operational.

Some real-world encodings are not operational, but generate the same proof obligations as a possible operational encoding (or stronger ones). We call these encodings *operational-equivalent* and define this notion as follows:

Definition 4.1.3 *An encoding $\langle \alpha, \cong, \beta \rangle$ is* operational-equivalent *in some sound Hoare logic if there is some operational encoding $\langle \alpha', \cong', \beta' \rangle$ s.t. for all S, P, Q, if in the Hoare logic $\vdash \{\lceil \beta(P) \rceil^{\mathring{p}}\} [\![\alpha(S)]\!]^{\mathring{p}} \{\lceil \beta(Q) \rceil^{\mathring{p}}\}$ then also $\vdash \{\lceil \beta'(P) \rceil^{\mathring{p}}\} [\![\alpha'(S)]\!]^{\mathring{p}} \{\lceil \beta'(Q) \rceil^{\mathring{p}}\}$.*

An example of an operational-equivalent encoding is the encoding of a statically-bound call as an assert (or exhale in a permission logic) of the call's precondition and a subsequent assume (or inhale) of its postcondition, as shown in Sec. 2.1.3: This encoding is not operational (since the encoded program does not jump to the called implementation), but in typical IVLs, call statements are verified by asserting the callee's precondition and assuming its postcondition. That is, from the verifier's perspective, there is no difference between an encoding containing a call and an encoding containing an assert-assume-pair (or, in a permission logic, an exhale-inhale-pair): If the latter verifies, the former would have also verified. As a result, the soundness of an operational-equivalent encoding follows directly from the soundness of the operational encoding it is equivalent to:

Theorem 4.1.2 *If $\langle \alpha, \cong, \beta \rangle$ is operational-equivalent, then it is relationally sound.*

Proof. If $\vdash \{\lceil\beta(P)\rceil^{\mathring{p}}\}[\![\alpha(s)]\!]^{\mathring{p}}\{\lceil\beta(Q)\rceil^{\mathring{p}}\}$, then, by the definition of operational-equivalence, also $\vdash \{\lceil\beta'(P)\rceil^{\mathring{p}}\}[\![\alpha'(s)]\!]^{\mathring{p}}\{\lceil\beta'(Q)\rceil^{\mathring{p}}\}$ for some operational encoding $\langle\alpha', \cong', \beta'\rangle$. By the soundness of the Hoare logic, we have that $\vDash \{\lceil\beta'(P)\rceil^{\mathring{p}}\}[\![\alpha'(s)]\!]^{\mathring{p}}\{\lceil\beta'(Q)\rceil^{\mathring{p}}\}$. From here, the proof continues like the proof for Thm. 4.1.1. □

Note that, for example, the encoding shown in Fig. 4.2 (right) is not operational-equivalent, since (assuming that P(y) is *low*(y)) it generates *weaker* proof obligations than any operational encoding: the encoded program trivially verifies, whereas any operational encoding would also require proving that the values y could have in two executions that take *different* branches are equal at the end (which is not the case).

In the next section, we will show an example of a more complex real-world encoding whose relational soundness can be derived from the fact that it is operational-equivalent.

4.2. Practical Relevance

In this section, we discuss two examples of real-world encoding patterns that violate the operationality criterion. For the first, we show that it is indeed unsound, and propose an alternative sound encoding. For the second, we show how the existence of an equivalent operational encoding can be used to argue that the encoding is sound despite not being operational.

As mentioned in Sec. 2.1.3, in most existing frontends, the encoding of virtually all source language constructs is operational; the main appeal of IVLs is, after all, that frontends *do not* have to compute verification conditions, but can instead "compile" input programs into an IVL without worrying about the verification process itself. However, many frontends still use non-operational encodings at least for *some* language constructs. Examples for this are VCC's encoding of local blocks, Dafny's encoding of calls on traits, Prusti's encoding for loops, and Nagini's encoding of dynamically-bound calls, which we have described in Chapter 2 and which we will discuss in detail in the next subsection. Additionally, as we will discuss in Sec. 4.3, all encodings of concurrent source languages into sequential IVLs necessarily have some non-operational elements.

Where non-operational encodings are used, this is often intentional to enable modular verification, since operational encodings for some language constructs are inherently non-modular (see the following example). In practice, one can therefore use the operationality criterion to quickly check that the existing encoding is sound for the vast majority of source language statements, and subsequently check the few remaining ones for relational soundness in detail.

4.2.1. Example 1: Dynamically-Bound Calls

We now show a real example of an unsound encoding of dynamically-bound calls that violates the operationality criterion, and show how to derive a sound alternative.

```
class A:
  def foo(self) -> int:
    # ensures low(result)
    return 0
```

```
class B(A):
  def foo(self) -> int:
    # ensures low(result)
    return 1
```

Figure 4.3.: Example of a problematic method override. B.foo overrides A.foo and has a compatible specification, but the implementations return different values.

Statically-bound method calls, i.e., calls whose exact target is fixed at compile time, can be encoded as method calls on the IVL level, which yields an operational encoding if the operational semantics of the IVL treats calls analogously to the source semantics. As mentioned above, the IVL verifier might later reason about calls in terms of pre- and postconditions instead of actually performing a call, but this transformation is not relevant here as long as the product program is constructed *before* such a desugaring step (and if it is, as we have shown above, an argument can be made for operational-equivalence).

However, the same approach does not work for dynamically-bound calls, i.e., calls whose target is chosen at runtime based on the type of the call's receiver. Since the implementation to be executed is generally not known during modular verification, it is not possible to encode dynamically-bound calls as method calls with the usual operational semantics (and existing IVLs do not offer dynamically-bound calls). Therefore, dynamically-bound calls are typically (e.g., in Dafny and, as shown in Chapter 2, in Nagini) directly encoded using the method specification of the static call target. Additional, separate proof obligations enforce that all overrides of a method respect *behavioral subtyping* [138], i.e., live up to the specification of the overridden method.

[138]: Liskov et al. (1994), 'A Behavioral Notion of Subtyping'

Consider method A.foo in Fig. 4.3 (left), which returns a constant integer and guarantees in its postcondition that the result is low. A dynamically-bound call a.foo(), where a has the static type A, will be encoded as an assertion (or exhale) of the (here, trivial) precondition of A.foo, followed by an assumption (or inhale) of the postcondition (we ignore side effects here for simplicity).

This encoding is sound if foo has a purely unary specification, without any relational parts. However, it does *not* fulfill our operationality criterion: The semantics of the source program performs a call to an implementation of foo (selected based on the dynamic type of a), whereas the IVL encoding directly encodes the proof obligations (similarly to the example from Fig. 4.2). It is also not operational-equivalent, since conditional calls to different overrides generate different proof obligations than a single, unconditional call to a single method.

Since the encoding is not operational, we have to check whether it is still relationally sound. Method B.foo in Fig. 4.3 (right), which overrides A.foo, shows that it is not. B.foo's contract is identical with that of A.foo, so behavioral subtyping holds trivially. B.foo's implementation satisfies the contract because it also returns a constant (but, importantly, a different one). Now, if a client calls a.foo() and, depending on a secret, the dynamic type of a is either A or B, then, depending on the secret, the result will be either 0 or 1. With the standard encoding of dynamically-bound calls outlined above, however, the client will assume the postcondition of A.foo and will therefore incorrectly conclude that the returned result is low.

To avoid this unsoundness while retaining the ability to use relational specifications[1], the problematic encoding must be replaced, either with

[1] One could, of course, forbid the use of relational specifications in some places to trivially avoid the unsoundness; this, however, is typically not desirable in practice.

an operational one, or with a different non-operational encoding that is sound for relational specifications. Note that, depending on the case, this second option may be non-trivial or may only be possible in restricted settings; after all, finding modular verification approaches for properties or programs that are not inherently modular has been an important research topic in verification in the past decades.

For our example, the former option is not desirable: An operational encoding for dynamically-bound calls would essentially have to case split on the dynamic type of the receiver and invoke the appropriate override. Since such an encoding is inherently non-modular (all possible overrides need to be known), we follow the alternative option: we give an example of a non-operational, but sound encoding.

For our new encoding we exploit the fact that the standard encoding is unsound only if the two executions of the program resolve the dynamically-bound call to two different implementations, that is, if the dynamic types of the receiver differ in the two executions. We reflect this observation by adjusting the encoding of pre- and postconditions as follows: (1) If the postcondition of a method guarantees some relational property (e.g., that an expression is low), we assume this at the call site *only if* the dynamic type of the receiver is low, that is, the calls in the two program executions are resolved to the same implementation. (2) Similarly, if a precondition requires that the call is a low event, we enforce that the receiver type is low in addition to the usual criterion for low events. Low events typically perform observable behavior such as I/O; it is therefore important that the *same* observable behavior is produced, independent of the receiver type. The meaning of low-assertions in preconditions remains unchanged, because the requirement of a method to receive low arguments is independent of the invoked implementation and must, thus, not be weakened. *lowEvent*-assertions are generally not allowed in postconditions, where they add no expressiveness.

For a specification language that contains no relational specification constructs except for *low*(e) and *lowEvent* (otherwise, the encoding of general relational assertions would mirror that of *low*(e)), we can encode this adjustment as follows:

$$\begin{aligned}\lceil \mathit{low}(e) \rceil^{\mathring{p}}_{\mathit{post}_r} &= (p^{(1)} = p^{(2)} \land \mathit{type}(r^{(1)}) = \mathit{type}(r^{(2)})) \Rightarrow e^{(1)} = e^{(2)} \\ \lceil \mathit{lowEvent} \rceil^{\mathring{p}}_{\mathit{pre}_r} &= p^{(1)} = p^{(2)} \land \mathit{type}(r^{(1)}) = \mathit{type}(r^{(2)})\end{aligned}$$

where *type*(e) represents the dynamic type of expression e, $\lceil P \rceil^{\mathring{p}}_{\mathit{post}_r}$ is the encoding of P in the postcondition of a call with receiver r, and $\lceil P \rceil^{\mathring{p}}_{\mathit{pre}_r}$ represents the same for the precondition. We leave the remaining encoding untouched, meaning that we can summarize the resulting encoding as follows:

1. We keep the existing check for behavioral subtyping for all overrides; this prevents, for example, that `A.foo` is overridden with a method that simply returns a secret value and therefore leaks information into the result.
2. We keep the existing encoding of dynamically-bound calls as an assert followed by an assume (or an exhale followed by an inhale),

but interpret $low(e)$ in preconditions and $lowEvent$ in postconditions as shown above.

In the example above, this encoding lets the caller assume that the result is low only if it can prove that the dynamic type of `a` is low.

The adjusted encoding is indeed sound:

Theorem 4.2.1 *Let S_c be of the form x:=$r.m()$, where r has static type A, and let $pre_{A.m}$ and $post_{A.m}$ be the pre- and postcondition of $A.m$. Assume that the implementation of* `A.m` *and its overrides fulfill their specifications and satisfy behavioral subtyping, and that all specifications contain no relational specification constructs except for $low(e)$ and $lowEvent$. Then the described encoding of S_c is relationally sound.*

Proof. Assume that $\langle S_c, \Sigma_1\rangle \rightarrow^* \langle \texttt{skip}, \Sigma'_1\rangle$ and $\langle S_c, \Sigma_2\rangle \rightarrow^* \langle \texttt{skip}, \Sigma'_2\rangle$ and $\Sigma_1, \Sigma_2 \vDash P$.

Also assume that $\vDash \{\lceil \beta(P) \rceil^{\mathring{p}}\} [\![\alpha(S_c)]\!]^{\mathring{p}} \{\lceil \beta(Q) \rceil^{\mathring{p}}\}$. Let $\Sigma_1 \cong \sigma_1$ and $\Sigma_2 \cong \sigma_2$. We have to show that $\Sigma'_1, \Sigma'_2 \vDash Q$.

W.l.o.g. we can assume that no information is framed around the call. If the encoding verifies, since the initial state must fulfill the initial assertion, we know that $\lceil \beta(P) \rceil^{\mathring{p}} \Rightarrow \lceil pre_{A.m}(r) \rceil^{\mathring{p}}_{pre_r}$. Similarly, we know that $\lceil post_{A.m}(r, x) \rceil^{\mathring{p}}_{post_r} \Rightarrow \lceil \beta(Q) \rceil^{\mathring{p}}$. Because of behavioral subtyping, for any subclass B of A, we know that overrides of $A.m$ must be such that $pre_{A.m} \Rightarrow pre_{B.m}$ and $post_{B.m} \Rightarrow post_{A.m}$. We perform a case split:

- If the dynamic type of r is equal in both source executions, both calls step to some statement S_B, representing the implementation of some override $B.m$ (where B might be identical to A). Since the implementation of $B.m$ fulfills its specification and we have $\langle S_B, \Sigma_1\rangle \rightarrow^* \langle \texttt{skip}, \Sigma'_1\rangle$ and $\langle S_B, \Sigma_2\rangle \rightarrow^* \langle \texttt{skip}, \Sigma'_2\rangle$ and $\Sigma_1, \Sigma_2 \vDash pre_{B.m}$, we have $\Sigma'_1, \Sigma'_2 \vDash post_{B.m}$ Since $post_{B.m} \Rightarrow post_{A.m}$ and $post_{A.m} \Rightarrow Q$, we are done.
- If the dynamic type of r is different in both source executions, the calls step to different implementations S_1 and S_2. $\lceil post_{A.m}(r, x) \rceil^{\mathring{p}}_{post_r}$ cannot contain relational information in this case, since all occurrences of $low(e)$ are conditional under the dynamic types being identical in both executions. Since $post_{A.m} \Rightarrow Q$, Q also cannot contain relational information. It therefore suffices to show that both S_1 and S_2 separately fulfill the Q, which follows from the fact that they fulfill their own specification and behavioral subtyping as in the previous case.

□

Note that this encoding is incomplete, since it is not aware that two different receiver types can lead to the same implementation being called (e.g., if one type inherits from the second and does not override the called method). Alternative encodings could explicitly represent this possibility. Conversely, one could approximate further (while remaining sound) by requiring the receiver *values* to be low, not just their types, in encodings that do not model dynamic types.

```
def main():
  # requires P
  # ensures Q
  parallel:
    thread 1:
      # requires P1
      # ensures Q1
      s1
    thread 2:
      # requires P2
      # ensures Q2
      s2
```

Figure 4.4.: Example program using a parallel block. The **parallel** statement introduces a parallel block which consists of multiple threads, each of which have their own pre- and postcondition.

```
method main()
  requires P
  ensures Q
{
  parallel_block()
}

method parallel_block()
  requires P1 * P2
  ensures Q1 * Q2
{
  assume false
}
```

```
method thread_1()
  requires P1
  ensures Q1
{
  s1
}

method thread_2()
  requires P2
  ensures Q2
{
  s2
}
```

Figure 4.5.: (Simplified version of) VerCors' non-operational encoding of the program in Figure 4.4.

As a final note, one can also argue that the adapted encoding is indeed correct because it results in the same proof obligations as a possible (non-modular) operational encoding that conditionally calls different method implementation depending on the receiver type. We will show how such an argument can be structured in the next subsection.

4.2.2. Example 2: Parallel Blocks in VerCors

We now show an example of a real-world encoding which is not operational, but which is operational-equivalent: The encoding of parallel blocks used by VerCors [38]. Using this encoding, we show how an argument for operational-equivalence can be made.

[38]: Blom et al. (2014), 'The VerCors Tool for Verification of Concurrent Programs'

Note that this example serves *only* to illustrate the concept of operational-equivalence, it is *not* an encoding that can then directly be combined with a sequential product construction, since the example uses concurrency and we are assuming a concurrent IVL here. More precisely, we will assume an IVL that has **fork** and **join** statements to spawn and join new threads similar to Chalice [133], and will show that VerCors' existing encoding, while not operational, is equivalent to a possible operational encoding in such a language. In a later section, we will then explain how a sequential product construction can be used to reason about concurrent source programs.

[133]: Leino et al. (2009), 'Verification of Concurrent Programs with Chalice'

Figure 4.4 shows an example program using such a parallel block (written using imaginary Python syntax; VerCors' internal language has parallel blocks as native language constructs). A parallel block consists of some number of threads which, as the name suggests, execute in parallel. Each thread has its own specification. In our example, we have two threads that each execute a statement and have some specification which we leave abstract.

```
method parallel_block()
  requires P1 ∗ P2
  ensures Q1 ∗ Q2
{
  var t1: Thread
  var t2: Thread
  t1 := fork thread_1()
  t2 := fork thread_2()
  join t1
  join t2
}
```

Figure 4.6.: Modified method parallel_block which would make VerCors' encoding operational and imposes the same proof obligations as the original encoding.

Figure 4.5 shows a simplified version of VerCors' encoding of the source program shown before. This encoding consists of several parts:

1. The main method itself, which, instead of the parallel block in the source program, contains a single method call.
2. The parallel_block method, whose precondition is the separating conjunction of the preconditions of all threads in the block, and whose postcondition is similarly the separating conjunction of the postconditions of all threads. This method's body is simply assume false, meaning that this method does not generate any proof obligations for the verifier.
3. The thread_1 method has the specification of the first thread in the block, and contains the body of this first thread.
4. Similarly, method thread_2 has the specification and the body of the second thread in the block.

This encoding is obviously not operational: When main executes the parallel block in the source program, the encoded program instead calls a method whose body immediately assumes false, meaning that in a typical operational semantics like the one shown in the previous chapter, the program would step to a magic state. Meanwhile, the two methods containing the code of the two threads are never executed from the main method, neither in parallel nor sequentially.

This encoding is a part of a class of encodings which, instead of encoding each statement in the source program to a single (potentially composite) statement in the IVL, creates several unconnected pieces of code in the IVL that check different verification conditions.

The intention behind this encoding is as follows: The two thread methods require the verifier to prove that the respective thread bodies are both correct with respect to their respective thread specifications[2]. The main method, when calling method parallel_block, requires the verifier to prove that the separating conjunction of all thread preconditions is fulfilled at the point when the parallel block is executed (by exhaling it), and allows it to inhale the separating conjunction of all thread postconditions holds after the parallel block. Essentially, the method call here does not really represent a method call at all, but is used to encode an exhale-inhale-pair that is a proof obligation which must be checked by the verifier.

However, this encoding can easily be extended to one that is operational: Figure 4.6 shows an alternative version of method parallel_block (written, as stated before, in a concurrent IVL which we will assume behaves like Chalice). In this alternative method implementation, we have added a body which actually executes both threads in parallel, by forking both of the thread methods and subsequently joining the threads. With this

[2] This encoding does not explicitly model the fact that these threads will execute in parallel, i.e., interleaved, in the source program; we explained why this is sound in permission logics like CSL [169] in Chapter 2, and will come back to this point in the next section.

alternative method, the entire encoding becomes operational: main, via its call to parallel_block, starts executing both thread methods in parallel, and the thread methods' bodies are (the encoded versions of) the statements executed by the threads in the original parallel block. After both threads have finished executing, they are joined by the main thread, and the execution can continue after the parallel block.

In addition to making the encoding operational, this adapted encoding also combines the previously disconnected elements of the encoding by introducing connections in the form of fork and join statements. For the original encoding to be operational-equivalent, the adapted encoding must also generate the same proof obligations as the original encoding (or weaker ones). This is indeed the case: The body of the adapted method parallel_block is constructed in such a way that it will *always* verify. The reasoning for this is as follows: As explained in Sec. 2.2.5.4 in Chapter 2, a fork statement is verified in permission logics, including Chalice, by exhaling the precondition of the forked method, and a join statement by inhaling its postcondition. However, the precondition of parallel_block gives it the separating conjunction of the preconditions of all threads, which means that it is always possible to sequentially exhale the individual thread preconditions. Similarly, after sequentially inhaling all thread postconditions (when verifying the join statements), the postcondition of parallel_block, the separating conjunction of all thread postconditions, will always be fulfilled.

As a result, method parallel_block in the adapted (operational) encoding always verifies by construction, and thus we can argue that the adapted encoding always verifies if the original encoding verifies (which does not impose any proof obligations in parallel_block). Therefore, the original encoding is operational-equivalent, and thus relationally sound.

As a final note, this pattern of combining disconnected parts of an encoding to create an operational encoding can also be applied to argue for the correctness of the adapted encoding of dynamically-bound calls shown in the previous subsection. We leave this as an exercise to the reader.

4.3. Product Programs and Concurrency

It has long been recognized that verification of information flow security for concurrent programs is especially challenging. The reason is that one needs to reason about a pair of executions that may have different thread interleavings (depending on the type of scheduler), and that these different interleavings can introduce additional information flows [187].

[187]: Sabelfeld et al. (2000), 'Probabilistic Noninterference for Multi-Threaded Programs'

Many product constructions, like modular product programs, explicitly target only sequential languages and thus do not support concurrent programs at all. While self-composition can in principle be applied to any programs, including concurrent ones, this would generally force the verifier to consider all combinations of possible thread interleavings of the two executions (in addition to the modularity issue also present for sequential programs), which leads to a state space explosion and thus makes verification infeasible in practice.

A naive product construction for concurrent programs would have to faithfully represent all combinations of potential thread interleavings, which makes verification infeasible in practice. To the best of our knowledge, there is currently only one existing product construction that explicitly targets concurrent programs, a technique by Farzan and Vandikas, which is implemented in the tool Weaver [78]. This technique avoids reasoning about all possible combinations of thread schedules by (automatically) creating a reduced product that represents all *relevant* classes of schedule combinations. However, reasoning about all relevant differences in interleavings is still required, and thus this approach is not thread-modular.

[78]: Farzan et al. (2019), 'Automated Hypersafety Verification'

We are aware of only one existing tool that automatically verifies information flow security in a thread-modular way, SecC, which automates SecCSL, a concurrent separation logic for information flow security proofs [72].

[72]: Ernst et al. (2019), 'SecCSL: Security Concurrent Separation Logic'

As explained in Chapter 2, many existing verifiers for trace properties avoid reasoning about different thread interleavings by employing a program logic (such as concurrent separation logic [169], on which SecCSL is based) that essentially reduces verification to sequential reasoning and allows concurrent verification problems to be encoded into sequential IVLs. Examples for such verifiers include VerCors and Nagini (using the Viper IVL), as well as Chalice [133], VCC, and Spec# (using the Boogie IVL).

[169]: O'Hearn (2007), 'Resources, concurrency, and local reasoning'

[133]: Leino et al. (2009), 'Verification of Concurrent Programs with Chalice'

In this section, we show how to use IVL-level product programs to extend such verifiers to handle information flow properties. We first describe in general terms how existing IVL encodings for concurrent languages work, and subsequently show how we can use similar principles to apply an IVL-based product construction, and which additional proof obligations we must fulfill to ensure that no flows exist as a result of concurrency.

It is well-known that concurrent programs can have different additional information flows depending on the type of scheduler [187]. To accommodate for this, we will present several solutions that give different guarantees when used with specific kinds of schedulers: We present detailed solutions for proving both *possibilistic* and *probabilistic* non-interference, and sketch an approach for proving *observational determinism*.

[187]: Sabelfeld et al. (2000), 'Probabilistic Noninterference for Multi-Threaded Programs'

Our goal is to describe a technique that applies to a wide range of source languages, IVLs, proof techniques, and encodings. Therefore, we focus on the high-level concepts, instead of formalizing them for one specific setting.

4.3.1. Concurrent IVL Encodings

Existing thread-modular encodings from concurrent source languages to IVLs do not model the exact behavior of the original language, in particular, the aforementioned thread interleavings (i.e., the parts of these encodings that concern concurrency are non-operational). Instead, they encode a verification condition that ensures that the original program is correct for *every possible* thread interleaving.

While the exact proof techniques differ between frontends, and can be based for example on CSL [169] (like in Nagini) or ownership [48, 90, 108,

[169]: O'Hearn (2007), 'Resources, concurrency, and local reasoning'

```
enc(l.acquire())  =  // gain access to protected memory;
                     assume Inv(l)
enc(l.release())  =  assert Inv(l);
                     // lose access to protected memory
```

Figure 4.7.: Standard IVL encoding of lock operations. *Inv*(l) denotes the invariant constraining the memory protected by lock l.

109], they generally follow a common pattern [132]: They prove that the source program is data race free, which ensures that thread interactions need to be considered *only* at well-defined synchronization points, for instance, upon acquiring or releasing a lock. The code between such interaction points can be considered to execute without interference from other threads, and thus can be reasoned about as if it were sequential.

[48]: Cohen et al. (2010), 'Local Verification of Global Invariants in Concurrent Programs'

[90]: Goubault et al. (2018), 'Concurrent Specifications Beyond Linearizability'

[108]: Jacobs et al. (2011), 'Expressive modular fine-grained concurrency specification'

[109]: Jacobs et al. (2005), 'Safe Concurrency for Aggregate Objects with Invariants'

[132]: Leino et al. (2009), 'A Basis for Verifying Multi-threaded Programs'

We focus on locks here, but other synchronization primitives are handled analogously. Program logics based on CSL or ownership systems formally connect a lock and the heap locations it protects, such that these locations may be accessed only while holding the respective lock. In addition, they associate locks with an invariant that constrains the values of the heap locations it protects. When acquiring a lock, a thread may assume that this lock invariant holds, and when releasing a lock, it has to prove that the invariant is re-established. A frontend can encode this into an IVL as depicted in Fig. 4.7. As an example, in a CSL-like setting like the one used by Nagini, gaining and losing access to the memory protected by a lock is expressed by inhaling and exhaling permissions in the lock invariant, respectively.

Our solution for information flow verification in concurrent programs follows the same basic approach: We exploit that code between lock operations can be considered to execute without interference, and that we can therefore use ordinary sequential product programs to reason about this code. To capture the thread interactions at synchronization points, we extend lock invariants to contain relational assertions (which can prescribe that some values protected by the lock are low), and add additional checks around lock operations to ensure that they do not give rise to unwanted information flow.

4.3.2. Possibilistic Non-Interference with Non-Deterministic Scheduling

For concurrent programs executed in a language or execution environment with a non-deterministic scheduler, standard non-interference is too strict a property: Since two executions can have different thread schedules, their low outputs can differ not only because of different high inputs, but because of scheduling differences, which are benign and obviously do not constitute a breach of information flow security[3]. One way of approaching this problem is to instead verify *possibilistic non-interference* [205], which enforces that high information does not influence the *possible* values of low outputs, i.e., if some combination of low output values is reachable from an initial state, then the same combination of low output values must still be reachable using *some* possible thread schedule after arbitrarily changing the high inputs. Possibilistic non-interference can be defined as follows:

[205]: Smith et al. (1998), 'Secure Information Flow in a Multi-Threaded Imperative Language'

[3] This is true under the assumption, which we make throughout this section, that the scheduling behavior does not depend on (potentially high) program state.

Definition 4.3.1 *A program s with a set of input variables I and output variables O, of which some subsets $I_l \subseteq I$ and $O_l \subseteq O$ are low, satisfies possibilistic non-interference iff for all σ_1, σ_2 and σ_1', if $\forall x \in I_l.\, \sigma_1(x) = \sigma_2(x)$ and $\langle s, \sigma_1\rangle \rightarrow^* \langle \texttt{skip}, \sigma_1'\rangle$ then $\langle s, \sigma_2\rangle \rightarrow^* \langle \texttt{skip}, \sigma_2'\rangle$ for some σ_2' s.t. $\forall x \in O_l.\sigma_1'(x) = \sigma_2'(x)$.*

Note that this property is sufficient only in a setting with purely nondeterministic thread scheduling without any underlying probability distribution, as it would otherwise allow high inputs to influence the *probability* of different outputs; we discuss a stronger notion of non-interference that addresses this problem in the next subsection.

Since we build on a proof technique that ensures data race freedom, we can see each program trace as a sequence of local operations and lock operations by specific threads, where (1) every local operation depends only on previous (local or lock) operations of the same thread, and (2) every lock operation depends only on the previous local operations of the same thread and all previous lock operations (of arbitrary threads). As a result, we can (akin to partial order reduction) rearrange segments freely as long as we retain the overall order of lock operations and the order of operations of every specific thread; in particular, we can rearrange a trace so that it consists of a number of *segments*, such that in each segment, one thread executes any number of local operations and then one lock operation.

Based on this observation, we impose proof obligations that ensure the following property: For every program trace with some schedule and some high and low inputs, and for arbitrary alternative high inputs, there exists a second trace with the same low but the alternative high inputs such that: (1) Both traces include the same lock operations performed by the same threads, in the same order, and (2) at each lock operation, the lock's invariant holds; in particular, the relational assertions of the lock invariant correctly relate the state protected by the lock in both traces.

To enforce this property, we devise four proof obligations that can be checked thread-locally:

1. Every lock operation o is a low event, i.e., if a thread executes o in the first execution, it will also execute o in the second execution.
2. Termination of the local code *before* the lock operation does not depend on secret data; i.e., if lock operation o is reached in the first trace, it will also be reached in the second trace.
3. o operates on the same lock in both executions, i.e., the lock is low.
4. If o releases the lock, i.e., makes a new lock state public, this lock state fulfills the relational invariant, meaning that heap operations meant to be low are identical in both executions after the lock operation.

Note that, even though the lock operations of both traces are closely aligned, the traces' local operations may differ. For instance, a thread may branch on a high guard as long as no lock operation is performed before the control flow re-joins.

$$
\begin{array}{lcl}
\mathit{poss}(\textsf{l.acquire()}) & = & \texttt{assert}\ \mathit{lowEvent};\\
 & & \texttt{assert}\ \mathit{low}(l);\\
 & & \textsf{// gain access to protected memory;}\\
 & & \texttt{assume}\ \mathit{Inv}(\textsf{l})\\
\mathit{poss}(\textsf{l.release()}) & = & \texttt{assert}\ \mathit{lowEvent};\\
 & & \texttt{assert}\ \mathit{low}(l);\\
 & & \texttt{assert}\ \mathit{Inv}(\textsf{l})\\
 & & \textsf{// lose access to protected memory;}\\
\mathit{poss}(\texttt{while}\ (e)\ \{s\}) & = & \texttt{assert}\ \mathit{low}(e);\\
 & & \texttt{assert}\ \mathit{lowEvent};\\
 & & \texttt{while}\ (e)\ \{\mathit{poss}(s); \texttt{assert}\ \mathit{low}(e)\}
\end{array}
$$

Figure 4.8.: Statement encoding for possibilistic information flow security. For loops, we check that the loop guard is low, ensuring that termination is also low.

4.3.2.1. Encoding

The aforementioned four properties can be checked as part of the encoding of lock operations. We adjust the encoding from Figure 4.7 for possibilistic non-interference as shown in Figure 4.8. For thread acquire and release, the assertions of *lowEvent* and *low*(l) directly ensure properties (1) and (3). Assuming and asserting the lock invariant works as in the standard IVL encoding for concurrent programs, but now this invariant can be relational, ensuring property (4). The condition on while loops is used to ensure property (2), which can be done simply by asserting that the loop condition is low for every loop in the program and reaching a (potentially non-terminating) loop is a low event; we assume, for simplicity, that there is no infinite recursion. Note that the latter is a simpler option than the encoding for termination-sensitive non-interference presented in the previous chapter, but said encoding may be used instead; we will discuss the relative advantages and disadvantages later.

We now show that the above checks are sufficient to satisfy Def. 4.3.1. In our proof, we will assume a similar language as in the previous chapter for the purpose of this proof, but without assume and havoc statements, since the former do not occur in actual code and the latter would be non-deterministic, whereas we assume that thread scheduling is the only possible source of non-determinism.

First, we prove a lemma stating that the encoding ensures that high data does not influence termination, and similarly, that high data does not influence whether a lock operation is executed:

Lemma 4.3.1 *If s is a sequential, deterministic command, $\sigma_1, \sigma_2 \vDash P$ and $\vDash \{\lceil P \rceil^{\mathring{p}}\} [\![\mathit{poss}(s)]\!]_2^{\mathring{p}} \{\lceil Q \rceil^{\mathring{p}}\}$ and $\langle \sigma_1, s \rangle \rightarrow^{l_1} \langle \sigma_1', s' \rangle$, s.t. s' is either* `skip` *or a statement that would execute a lock operation in its next step, and no lock operation has been executed in any of the l_1 steps, then $\langle \sigma_2, s \rangle \rightarrow^{l_2} \langle \sigma_2', s' \rangle$ for some σ_2' and l_2.*

Proof. The proof goes by induction on the structure of s.

If s is a basic statement, the proof is immediate if s' is `skip`, since then s cannot be a lock operations by assumption, and s therefore always terminates, and cannot fail by Thm. 3.4.2. In the case where s' would execute a lock operation, we must have $s' = s$ and $l_1 = 0$, and the second zero-step execution is trivial.

If s is a conditional, its condition e either evaluates to the same value in both executions, in which case both step to the same substatement s'' and we can apply the induction hypothesis to s''. Otherwise, the condition evaluates to different values, the two executions step to different statements s_1 and s_2, and thus the activation variables of the branches in the product program have different values for both executions from here on out. As a result, both statements s_i can never step to a statement s''_i that is a loop or a lock operation, since $poss(s''_i)$ asserts that the activation variables have equal values (*lowEvent*), and this assertion would fail, which is impossible because of the given Hoare triple. Therefore s cannot lead to a lock operation, and s' must instead be `skip`. Since any loop-free statement s must necessarily terminate, the second execution of s will terminate as well.

If s is a sequential composition $s_1; s_2$, we have two cases: Either s_1 leads to a lock operation s'' from σ_1, meaning that s' has the form $s''; s_2$. Then, by the induction hypothesis, s_2 also leads to s'' from σ_2, and we are done. Alternatively, s_1 terminates and s_2 steps to s' from σ_1. Then we apply the induction hypothesis twice to show s_1 also terminates from σ_2, and subsequently s_2 steps to s' again.

Finally, if s is a loop, the assertion $low(e)$ in $poss(s)$ ensures that the loop condition e evaluates to the same value in both executions. If it is false, both executions terminate immediately and s' must be `skip`. Otherwise they must both execute the loop body s'' at least once. We do a proof by induction on the number j of complete executions of s'' in the first execution *before* the loops either terminates or an iteration steps to the lock operation s', and show that for any j, if j executions of the loop body s'' terminate in the first execution, then they also do so in the second execution, and the value of e after these j loop executions will be equal in both executions. The case for $j = 0$ is trivial, since we start in a state where e has the same truth value on both executions, as already stated. In the inductive step, we know from initial the induction hypothesis that since s'' terminates in the first execution, it does so in the second. Additionally, since $poss(s)$ ensures that e is again low at the end of the loop body, we know that e has the same truth value in both executions. By the induction hypothesis, the remaining $j - 1$ executions will also terminate and lead to states with equal values of e. Then, if e is false in both executions, both executions terminate and s' must be skip. Otherwise, the next execution of a loop body must lead to a lock operation s' in the first operation, and by the original induction hypothesis, it also does so in the second. □

Now we can prove our main theorem, the soundness of our encoding. We do this for programs that, at the top level, are a parallel composition of multiple threads. Without defining properties of program states in any more detail, we assume that a program state σ can be split up into several partial states (e.g., partial heaps and parts of the store), and use the operator $\uplus$ to combine states. That is, if σ consists of the partial states σ_1 and σ_2, we write $\sigma = \sigma_1 \uplus \sigma_2$. Similarly, we assume that there is an operator $*$ that allows combining assertions in such a way that if two states σ_1 and σ_2 individually fulfill the assertions P_1 and P_2, the combined state $\sigma_1 \uplus \sigma_2$ fulfills the assertion $P_1 * P_2$. This property is obviously fulfilled by the separating conjunction in permission logics, which is why we use the same notation, but other specification techniques may have similar concepts.

Theorem 4.3.2 *Let $s = s^1 || \dots || s^n$ and $\sigma_1 = \sigma_1^1 \uplus \cdots \uplus \sigma_1^n \uplus \sigma_1^L$ and $\sigma_2 = \sigma_2^1 \uplus \cdots \uplus \sigma_2^n \uplus \sigma_2^L$ and $P = P^1 * \cdots * P^n$ and $Q = Q^1 * \cdots * Q^n$, s.t. $\sigma_1^i, \sigma_2^i \vDash P^i$ for all $i \in \{1, \dots, n\}$. Let $\sigma_1^L, \sigma_2^L \vDash Invs$, where $Invs$ are the (relational) invariants of a set of locks. If $\vDash \{\lceil P^i \rceil^{\mathring{p}}\} [\![poss(s^i)]\!]_2^{\mathring{p}} \{\lceil Q^i \rceil^{\mathring{p}}\}$ for all i and $\langle \sigma_1, s \rangle \rightarrow^* \langle \sigma_1', \texttt{skip} \rangle$ in trace τ_1, then there is a trace τ_2 of the form $\langle \sigma_2, s \rangle \rightarrow^* \langle \sigma_2', \texttt{skip} \rangle$ for some σ_2' s.t. $\sigma_1', \sigma_2' \vDash Q$.*

Proof. As described before, all steps in a trace will either be thread-local steps or lock operations (lock releases or acquires), and we partition each thread's steps into segments that consist of some number of local steps and end with a lock operation. The proof goes by induction on the number j of segments in τ_1. If $j = 0$, then threads do not interact at all and execute completely independently. In this case, by Lemma 4.3.1, each thread's termination in τ_1 implies its termination in τ_2, and by Thm. 3.4.9, all postconditions will be fulfilled.

Otherwise, we will show that it is possible to construct τ_2 such that it has the same sequence of lock operations as τ_1, and the (relational) lock invariants hold between both executions at each point of matching lock operations. We consider the segment that ends with the first lock operation in τ_1, which we assume is performed by some thread i, and which must therefore have the form $\tau_1' = \langle \sigma_1^i, s^i \rangle \rightarrow^* \langle \sigma_1'^i, s'^i \rangle \rightarrow \langle \sigma_1''^i, s''^i \rangle$, for some $s'^i, s''^i, \sigma_1'^i, \sigma_1''^i$ s.t. s'^i performs a lock operation. In the trace, this segment is of course interleaved with steps from other threads, but (as we argued before) since we assume that our encoding ensures data race freedom, the segment would execute identically if all its steps were performed without interruption, as in τ_1'. We construct τ_2 s.t. it exclusively schedules thread i until it performs the same thread interaction on the same lock. By Lemma 4.3.1, we get that $\langle \sigma_2^i, s^i \rangle \rightarrow^* \langle \sigma_2'^i, s'^i \rangle$. Then, we make a case distinction on the kind of lock operation:

- If s'^i performs a release, then $poss(s'^i)$ ensures that $\sigma_1'^i, \sigma_2'^i \vDash low(l) * Inv(l) * R$ for some R, where $Inv(l)$ is the invariant of the acquired lock l. That is, the relational invariant of the lock holds, and both threads release the same lock l. We then split $\sigma_1'^i$ s.t. $\sigma_1'^i = \sigma_1''^i \uplus \sigma_1^l$, and split $\sigma_2'^i$ analogously, s.t. $\sigma_1^l, \sigma_2^l \vDash Inv(l)$ and $\sigma_1''^i, \sigma_2''^i \vDash R$. σ_1^l is added to the lock state σ_1^L in the next step, and analogously for the second execution. That is, we get that $\langle \sigma_2'^i, s'^i \rangle \rightarrow^* \langle \sigma_2''^i, s''^i \rangle$ and have ensured that the lock invariant holds for the released lock.
- If s'^i performs an acquire, then $poss(s'^i)$ ensures that $\sigma_1'^i, \sigma_2'^i \vDash low(l) * R$ for some R, so both threads acquire the same lock l. Additionally, since the lock states σ_1^L and σ_2^L fulfill all lock invariants, we know that they contain some parts σ_1^l and σ_2^l s.t. $\sigma_1^l, \sigma_2^l \vDash Inv(l)$ and $\sigma_1''^i = \sigma_1'^i \uplus \sigma_1^l$. Now the second trace can perform a step $\langle \sigma_2'^i, s'^i \rangle \rightarrow^* \langle \sigma_2''^i, s''^i \rangle$, where $\sigma_2''^i = \sigma_2'^i \uplus \sigma_2^l$, and it is safe to assume $\sigma_1''^i, \sigma_2''^i \vDash R * Inv(l)$.

Since all lock invariants are fulfilled at this point, and we have $\vDash$

```
def main(secret: bool) -> None:
  c = Cell()
  l = CellLock(c)
  l.acquire()
  c.val = 4
  if secret:
    l.release()
    l.acquire()
  c.val = 5
  c.release()
```

Figure 4.9.: Possibilistic information flow violation via a secret-dependent lock release. The Cell state 4 is visible to other threads only if secret is True.

$\{\lceil R \rceil^{\mathring{p}}\}[\![poss(s''^{i})]\!]_2^{\mathring{p}}\{\lceil Q^i \rceil^{\mathring{p}}\}$, we can now apply the induction hypothesis on the remaining $j-1$ segments. □

4.3.2.2. Discussion

With our verification technique, the product construction on the IVL level does not need to be aware of concurrency in *any* way; applying the standard sequential product construction to the updated encoding is sufficient to ensure possibilistic non-interference in concurrent programs.

To the best of our knowledge, we are the first to consider possibilistic information flow in a setting with locks, and therefore the first to propose that the order of lock operations must be constrained (proof obligations 1 and 3). The example in Fig. 4.9 demonstrates that proof obligation 1 is indeed necessary to prevent unwanted information flow: The CellLock protects the val field of a Cell object, which is intended to be low. The code unconditionally sets the field to two constants (first to 4, then to 5), which should be allowed since the constants are low. However, whether the lock is *released* while the cell has value 4 depends on a secret. As a result, when a different thread acquires the lock and sees that the value is 4, this leaks that the secret *must* have been true, which violates possibilistic non-interference.

Another example that illustrates the requirement to ensure that high data does not influence *which* lock a lock operation accesses (proof obligation 3) can be found in Fig. 4.10. Here, two locks are created, and thread 1 acquires the first one. Thread 2 acquires, depending on the secret, either the same lock or a different one. This influences the possible results of the program: If both threads acquire the same lock, then the print statements of one thread cannot be interleaved with those of the other, otherwise they can. As a result, if the attacker observes the pattern 1212 (or any other interleaving of 1s and 2s), they know with certainty that the two threads acquired different locks and secret must therefore be False.

The necessity to prevent termination differences in a concurrent setting (i.e., proof obligation 2) has been recognized before, e.g. in work on security type systems [205]. Fig. 4.11 shows an example of a program that leaks information because of secret-dependent non-termination. For convenience, we have omitted locks in this example, but we assume that all variables accessed in more than one thread (i.e., go_1, go_2, proceed, and leak) are protected by a lock and that variables proceed and leak are marked as low by the lock invariant. In the example, thread 0 controls, by setting the shared variables go_1 and go_2 (both initially False), whether or not the other threads set the value of the shared variable leak to a

[205]: Smith et al. (1998), 'Secure Information Flow in a Multi-Threaded Imperative Language'

```
def thread1(l: Lock) -> None:
  # requires lowEvent
  l.acquire()
  print(1)
  print(1)
  l.release()

def thread2(l: Lock) -> None:
  # requires lowEvent
  l.acquire()
  print(2)
  print(2)
  l.release()
```

```
def main(secret: bool) -> None:
  l1 = Lock()
  l2 = Lock()
  if secret:
    l = l1
  else:
    l = l2
  fork thread1(l1)
  fork thread2(l)
```

Figure 4.10.: Possibilistic information flow violation through locks. If secret is true, both threads acquire the same lock, and their critical sections cannot be interleaved.

```
def thread_0(secret: bool):
  if secret:
    go_1 = True
  else:
    go_2 = True
  while not proceed:
    pass
  print(leak)
```

```
def thread1():
  while not go_1:
    pass
  leak = True
  proceed = True

def thread2():
  while not go_2:
    pass
  leak = False
  proceed = True
```

Figure 4.11.: Example of a possibilistic information flow violation through thread interactions. The value of secret is leaked through variable leak.

constant, and it does so based on variable secret. The go variables are assigned under a high condition, but the assignments to variable leak are not. However, whether or not they can ever be executed depends on the values of the high go variables, because those determine whether the loops before the respective assignments terminate or not.

As we stated before, to prevent this kind of information flow, it is necessary to show that loop termination does not depend on high values, which we do by forbidding loops with high guards. However, this criterion is unnecessarily strict in some cases. Consider the example in Fig. 4.12, which shows an alternative implementation of thread 2. It also contains a loop with a high guard that is followed by an assignment to a shared variable (leak). However, even though the guard is high, this loop always terminates, and there is therefore always a possible schedule that will execute the assignment to leak after the loop, which means that there is no possibilistic non-interference violation.

To allow such loops, one can use a different criterion that forbids only loops whose *termination* depends on a high value. As a result, one could allow loops for which secret data influences the number of iterations but does not influence termination, as is the case in Fig. 4.12. We can easily implement this (and we did, in fact, do this in our implementation in Nagini) by instead using the technique for preventing termination channels in loops described in Sec. 3.5.5 in the previous chapter: We require the user to specify, for each loop, an exact condition under which its execution terminates (in the example, this condition is simply True). We then (1) use standard techniques to prove that the loop terminates iff the condition is true when the loop is reached, and (2) prove that this termination condition is low.

Note, however, that this verification technique is not strictly more precise than the one presented in this chapter, since it requires that a condition under which a loop terminates can be expressed in thread-local terms

```
def thread2(secret: int):
    while secret > 0:
        secret -= 1
    leak = False
    proceed = True
```

Figure 4.12.: Example of a loop with a high loop condition whose termination is low.

```
def thread1(l: Lock, c: Cell):
    ctr = 0
    for i in range(100):
        ctr += 1
    l.acquire()
    c.val = 1
    l.release()
```

```
def thread2(l: Lock, c: Cell, secret: int):
    ctr = 0
    for i in range(secret):
        ctr += 1
    l.acquire()
    c.val = 2
    l.release()
```

Figure 4.13.: Example of probabilistic information flow. With a non-deterministic scheduler, secret does not influence the set of possible outputs, but a greater secret leads to higher probability of seeing a final cell value of 2.

before the loop executes and that loop termination under this condition can be proved locally, which is not always the case. For example, one could write a safe version of the code in Figure 4.11 where thread 0 *always* sets go_1 and go_2 to True and therefore no information is leaked: In this example, loop termination in threads 1 and 2 depends on the actions of an outside thread, and no local termination proof is possible. However, with the encoding shown in Figure 4.8, this example can be proved correct, since both go_1 and go_2 can be marked as low in the lock invariant, and therefore both loop conditions are low. Therefore, both ways of dealing with loop termination can be used in conjunction, and none of them always subsumes the other.

4.3.3. Probabilistic Non-Interference with Probabilistic Scheduling

Possibilistic non-interference is generally regarded as insufficient in the presence of schedulers that follow a known probability distribution, since in this scenario, programs can leak high information through the *probability* of seeing a certain result. Fig. 4.13 illustrates the problem: The final value of c.val can be either 1 or 2, that is, possibilistic non-interference holds. However, with most schedulers, a final value of 2 is much more likely for greater secret values than for lower values because the assignment of 1 is more likely to happen before the assignment of 2.

A stronger notion of non-interference that forbids such leaks is *probabilistic* non-interference [222], which requires that two executions from low-equivalent initial states will produce the same low outputs with the same probabilities.

[222]: Volpano et al. (1998), 'Probabilistic Noninterference in a Concurrent Language'

Definition 4.3.2 *A program s with a set of input variables I and output variables O, of which some subsets $I_l \subseteq I$ and $O_l \subseteq O$ are low, satisfies probabilistic non-interference iff for all σ_1, σ_2 and σ_1', if $\forall x \in I_l.\, \sigma_1(x) = \sigma_2(x)$ and $\langle s, \sigma_1\rangle \rightarrow^* \langle \mathtt{skip}, \sigma_1'\rangle$ with probability p then $\langle s, \sigma_2\rangle \rightarrow^* \langle \mathtt{skip}, \sigma_2'\rangle$ with probability p for some σ_2' s.t. $\forall x \in O_l. \sigma_1'(x) = \sigma_2'(x)$.*

The information flow in Fig. 4.13 is caused by secret data influencing the timing of thread 2, which in turn may affect the relative order of modifications of shared variables. This problem is sometimes called an

$$
\begin{array}{lcl}
\mathit{prob}(\mathsf{l.acquire()}) & = & \texttt{assert}\ \mathit{low}(l); \\
 & & \text{// gain access to protected memory;} \\
 & & \texttt{assume}\ \mathit{Inv}(\mathsf{l}) \\
\mathit{prob}(\mathsf{l.release()}) & = & \texttt{assert}\ \mathit{low}(l); \\
 & & \texttt{assert}\ \mathit{Inv}(\mathsf{l}) \\
 & & \text{// lose access to protected memory;} \\
\mathit{prob}(\texttt{while}\ (e)\ \{s\}) & = & \texttt{assert}\ \mathit{low}(e); \\
 & & \texttt{while}\ (e)\ \{\mathit{prob}(s); \texttt{assert}\ \mathit{low}(e)\} \\
\mathit{prob}(\texttt{if}\ (e)\ \{s_1\}\ \texttt{else}\ \{s_2\}) & = & \texttt{assert}\ \mathit{low}(e); \\
 & & \texttt{if}\ (e)\ \{\mathit{prob}(s_1)\}\ \texttt{else}\ \{\mathit{prob}(s_2)\} \\
\mathit{prob}(r.m()) & = & \texttt{assert}\ \mathit{low}(\mathit{type}(r)); \\
 & & r.m()
\end{array}
$$

Figure 4.14.: Statement encoding for probabilistic information flow security.

internal timing channel, since the final value of the shared variables can leak information about the relative timing of threads.

Of course, preventing high data from influencing execution time is difficult in the general case, but becomes more viable depending on the programming language and execution environment. As discussed in Sec. 3.5.6 in the previous chapter, one can for example choose to explicitly track timing as ghost state, and assert that the current execution time is low at critical points. Alternatively, if the execution time of all basic statements does not depend on their data, it suffices to prevent branching on high values, since this ensures that high data does not influence which statements will be executed. This assumption is frequently made in the literature about information flow security for concurrent systems (e.g. [72, 160, 201], although it is not necessarily true on typical existing computers. This basic system can, however, be extended to account for different sources of timing differences: For example, if (in addition to branches) memory accesses influence execution timing, which is usually the case in reality because of cache effects, one can add additional proof obligations to ensure that the same memory locations are accessed, independently of secret data. We will, however, stick with the existing literature and assume, for the sake of simplicity, that preventing high branches is sufficient.

[72]: Ernst et al. (2019), 'SecCSL: Security Concurrent Separation Logic'
[160]: Murray et al. (2018), 'COVERN: A Logic for Compositional Verification of Information Flow Control'
[201]: Smith (2001), 'A New Type System for Secure Information Flow'

That is, to prevent secrets from influencing the timing of operations, we additionally assert that every branch condition in the program is low, meaning that the two executions will always follow the same code path, which leads to the adjusted encoding in Fig. 4.14. Note that the check that branch conditions are low must also be performed for any implicit branches; e.g., with the encoding of dynamically-bound calls shown before, we must now assert that the type of the receiver of every such call is low. Also note that since we enforce that branches are low, the *lowEvent* conditions we showed in the possibilistic encoding will be trivially fulfilled and can be omitted here. However, we still need to assert that acquired and released lock references are low.

With this adjusted encoding, probabilistic non-interference can be verified using simple assertions in the IVL encoding and subsequently performing a standard product construction on the IVL level.

We again formalize the correctness of our encoding, using the same model as before, except that executions are now associated with probabilities.

First, we prove that our encoding ensures that high data does not influence a program's control flow, i.e., in verified programs, each thread performs the same step in both executions:

Lemma 4.3.3 *If s is a sequential, deterministic command that does not contain lock operations, $\models \{\lceil P\rceil^{\mathring{p}}\}[\![\mathit{prob}(s)]\!]_2^{\mathring{p}}\{\lceil Q\rceil^{\mathring{p}}\}$ and $\sigma_1, \sigma_2 \models P$ and $\langle\sigma_1, s\rangle \longrightarrow \langle\sigma_1', s'\rangle$, then $\langle\sigma_2, s\rangle \longrightarrow \langle\sigma_2', s'\rangle$ for some σ_2'.*

Proof. The proof goes by induction on the structure of s. If s is a basic statement, the proof is trivial, since all basic statements only step to `skip` except assertions, and errors are ruled out by the Hoare triple and Thm. 3.4.2. If s is a conditional, then since $\mathit{prob}(s)$ asserts that its condition e is low, both executions step to the same branch s'', and we are done. If s is a loop, by the same argument, the loop condition e has the same value in both executions, and either both executions finish or both execute the same loop body. Finally, if s is a sequential composition $s_1; s_2$, then either s_1 is `skip` and both executions step to s_2, or by the induction hypothesis s_1 will step to some s_1' in both executions, and therefore s steps to $s_1'; s_2$ in both executions as well, and we are done. □

Building on this, we now prove our soundness theorem:

Theorem 4.3.4 *Let $s = s^1 || \ldots || s^n$ and $\sigma_1 = \sigma_1^1 \uplus \cdots \uplus \sigma_1^n \uplus \sigma_1^L$ and $\sigma_2 = \sigma_2^1 \uplus \cdots \uplus \sigma_2^n \uplus \sigma_2^L$ and $P = P^1 * \cdots * P^n$ and $Q = Q^1 * \cdots * Q^n$, s.t. $\sigma_1^i, \sigma_2^i \models P^i$ for all $i \in \{1, \ldots, n\}$. Let $\sigma_1^L, \sigma_2^L \models \mathit{Invs}$, where Invs are the (relational) invariants of a set of locks. If $\models \{\lceil P^i\rceil^{\mathring{p}}\}[\![\mathit{prob}(s^i)]\!]_2^{\mathring{p}}\{\lceil Q^i\rceil^{\mathring{p}}\}$ and $\langle\sigma_1, s\rangle \longrightarrow^l \langle\sigma_1', \texttt{skip}\rangle$ with probability p in trace τ_1, then there is a trace τ_2 of the form $\langle\sigma_2, s\rangle \longrightarrow^l \langle\sigma_2', \texttt{skip}\rangle$ with probability p for some σ_2' s.t. $\sigma_1', \sigma_2' \models Q$.*

Proof. The probability p of τ_1 is the product of each of its steps, i.e., $p = p_1 \times \cdots \times p_l$. The proof goes by induction on the number l of steps of τ_1. We will assume that joining terminated threads takes one execution step, so if $l = 1$, then all thread statements s^i must be `skip` so that the program terminates immediately, and we must have $P \Rightarrow Q$ and are done; $l = 0$ is impossible. Otherwise, in the first step of τ_1, the scheduler selects some thread i with probability p_1, and the thread executes $\langle\sigma_1^i, s^i\rangle \longrightarrow \langle\sigma_1'^i, s'^i\rangle$ for some $\sigma_1'^i$ and s'^i. Since the thread scheduler does not depend on high data, it will also select thread i with probability p_1 from state $\langle\sigma_2, s\rangle$. By Lemma 4.3.3, we have that $\langle\sigma_2^i, s^i\rangle \longrightarrow \langle\sigma_2'^i, s'^i\rangle$ for some $\sigma_2'^i$. As in the proof of Thm. 4.3.2, we know that the assertions in $\mathit{prob}(s^i)$ ensure that all lock invariants are preserved if this step executes a lock operation. Again, we have $\models \{\lceil R^i\rceil^{\mathring{p}}\}[\![\mathit{prob}(s'^i)]\!]_2^{\mathring{p}}\{\lceil Q^i\rceil^{\mathring{p}}\}$ for some R^i s.t. $\sigma_1'^i, \sigma_2'^i \models R^i$, and we can apply the induction hypothesis on the remaining $l - 1$ steps. □

As a corollary, we can show that the same encoding guarantees standard non-interference in the presence of a deterministic thread scheduler: In this setting, at every point in the program, the scheduler will choose a specific thread with probability 1 in the first execution. As a result, for any second execution with different high inputs, there will be an

$$\mathit{lowLoc}(x{:=}e) \;=\; \texttt{assert}\ \mathit{lowEvent};\quad \texttt{assert}\ \mathit{low}(e);\quad x{:=}e$$

Figure 4.15.: Encoding of writes to low locations, where x is a low local variable. Writes to low fields would be encoded analogously; assignments to non-low locations do not have to be modified.

execution with probability 1 that has the same public outputs. Since both executions have probability 1, there are no other possible executions from the same initial states, so the programs are deterministic and satisfy ordinary non-interference. Note that, throughout the remainder of this chapter, we will nevertheless refer to our encoding exclusively as an encoding for probabilistic non-interference to avoid confusion.

4.3.4. Observational Determinism

A scheduler-independent definition of information flow security that is often applied to concurrent programs is *observational determinism* [106, 233]. Observational determinism is defined for a setting where an attacker can not only observe the low outputs of a program, but the values of a set of memory locations declared to be low *throughout its entire execution*. In this setting, observational determinism requires that the sequence of values the attacker observes in a given low memory location is not influenced by high data, modulo stuttering and prefixing [233].

[106]: Huisman et al. (2006), 'A Temporal Logic Characterisation of Observational Determinism'

[233]: Zdancewic et al. (2003), 'Observational Determinism for Concurrent Program Security'

[233]: Zdancewic et al. (2003), 'Observational Determinism for Concurrent Program Security'

We have not considered a setting where attackers can observe memory during the execution in this chapter or the previous one, however, for sequential programs, modular product programs can be easily adapted to be used in this setting: One can simply add an assertion for every assignment to a (potentially) low memory location that the assigned value must be low, and the assignment itself must be a low event, as shown in Figure 4.15. In sequential programs, these added proof obligations ensure that for any low memory location, the same sequence of values will be written to it throughout the execution of a verified program, independently of high inputs.

For concurrent programs, it has been shown before that observational determinism holds if (in addition to the usual rules that prevent illicit information flows in sequential programs), there are no competing (i.e., racy) accesses to low memory locations from different threads [233]. Note that the notion of racy here is different from the notion of data races we used before (which is automatically ruled out by techniques like CSL), in that competing accesses that are protected by locks are still problematic. That is, in this setting, we say two accesses compete if they can occur in any order and at least one of them is a write, no matter if the accesses are protected by locks.

[233]: Zdancewic et al. (2003), 'Observational Determinism for Concurrent Program Security'

Figure 4.16 shown an example program consisting of two threads, one of which receives a high input secret, that satisfies probabilistic non-interference with a probabilistic scheduler, but does not satisfy observational determinism. In this example, the val field of the Cell object modified by both threads is a low memory location and observable by the attacker. The first thread acquires the lock that protects the Cell object five times and writes the values 0, 1, 2, 3, 4 to the val field, in this order.

```
def thread1(l: Lock, c: Cell, secret: int):
  ctr = 0
  for i in range(5):
    l.acquire()
    c.val = i
    l.release()
```

```
def thread2(l: Lock, c: Cell) -> int:
  # ensures low(result)
  for i in range(2):
    pass
  l.acquire()
  c.val = 1
  l.release()
  for i in range(2):
    pass
  l.acquire()
  res = c.val
  l.release()
  return res
```

Figure 4.16.: Example program that satisfies probabilistic non-interference, but does not satisfy observational determinism. Cell.val is a low location, secret is a high input, all other inputs are low.

The second thread waits for some time, then writes the value 1 to the Cell, waits some more, and then reads and returns the Cell's current value.

With a probabilistic scheduler, the second thread can potentially insert value 1 at any point before, inbetween, or after the writes of the first thread; most likely (if threads are scheduled with uniform probability), it will write it somewhere in the middle of the first thread's sequence. Crucially, the value of secret does not influence the probability of the point when the second thread's write occurs relative to the first threads' writes. Similarly, the second thread will either read some value written by thread 1, or the value it wrote itself, depending on scheduling, but the high input does not influence the probabilities of different relative timings. As a result, the example satisfies probabilistic non-interference.

However, it does not satisfy observational determinism: Without making any assumptions about the scheduler, the writes of the first and second thread can be interleaved in arbitrary ways, so the sequence of values seen in the Cell's field is not deterministic. If the scheduling depends on variable secret (which we have assumed was not the case when talking about probabilistic non-interference), then the sequence of values the observer sees can easily leak high information (e.g., the scheduler could prefer thread 1 if secret is positive and vice versa, s.t. a large secret value leads to value 1 being written after all writes by thread 1). In this case, clearly also the result returned by thread 2, which is intended to be low, will leak secret information.

Observational determinism can be verified using modular product programs and an existing IVL encoding that prevents data races (in the original sense) by simply not allowing lock invariants to be relational, i.e., by forbidding low-assertions in lock invariants, and by forbidding permissions to low locations in lock invariants. It is easy to see why: When using a verification techniques that does not allow data races, competing accesses are *only* possible if the two competing threads both acquire a lock before performing the access, and the lock gives them the permission to make the access. If, however, acquiring a lock never gives a thread additional permission to a memory location regarded as low, then it also cannot enable the thread to modify a low memory location in any way. Additionally, if a lock invariant does not contain any low-assertions, then all threads must regard data read from the locked memory location as potentially high, and (if proven to be correct) will therefore not leak any information about such data to their low outputs.

Put differently, this restriction on lock invariants ensures that

- the low results of each thread can depend *only* on the thread's own low inputs (since any data from other threads, which could potentially be influenced by thread scheduling, has to be regarded as high).
- all writes to a low location will happen by a single thread (since otherwise different threads need permission to the same location, which is only possible if the permission is passed back and forth between different threads through a lock)[4]; in this scenario, the simple check shown in Figure 4.15 suffices to ensure this thread always writes the same sequence of values to a given location.

In addition to these restrictions, it can be useful to also prevent secret-dependent thread termination; this is not required by the original definition of observational determinism [233] but was convincingly argued to be vital by Huisman et. al. [106]. To achieve this, we can use either of the two encodings we already showed in Sec. 3.5.5 in the previous chapter or in our encoding for possibilistic non-interference in this section.

We will not formalize the notion of observational determinism for our setting, nor will we provide a detailed soundness argument for this encoding here, since our main setting does not contain the notion of low memory locations.

To summarize, we have shown that it is possible to extend existing verifiers for concurrent programs to verify both possibilistic and probabilistic non-interference as well as observational determinism with very small changes in the frontend, using only a sequential product construction and not requiring any changes on the level of the IVL (except the ability to write relational specifications).

4.4. Implementation and Evaluation

In this section, we evaluate the performance of the proposed architecture, by extending the previously information flow unaware Nagini verifier according to our design, enabling it to verify non-interference for sequential Python programs as well as possibilistic and probabilistic non-interference for concurrent Python programs. We will first briefly describe the adaptations we needed to make to both Nagini and the underlying Viper verification infrastructure, then evaluate the performance overhead generated by the product transformation, and subsequently evaluate the implementation on a number of information flow examples, comparing it to SecC [72] in the process.

4.4.1. Implementation in Viper

We did not have to modify any part of the core Viper IVL, its backend verifiers, or any other parts of the core Viper code base. Instead, we added new code that extends the language and performs the product construction, building on the implementation described in Chapter 3. Crucially, this means that the added code is compiled separately and existing Viper code does not depend on it, showing that (at least in this case), the proposed architecture could be implemented without any support from the IVL developers.

[4] There is one exception: Writes can be by different threads, but only in a clearly defined order, if one thread forks another and passes the permission to a low location to the new thread. In this scenario, all writes by the new thread will always happen after all writes by the old thread, s.t. the writes are still essentially sequential.

[233]: Zdancewic et al. (2003), 'Observational Determinism for Concurrent Program Security'

[106]: Huisman et al. (2006), 'A Temporal Logic Characterisation of Observational Determinism'

[72]: Ernst et al. (2019), 'SecCSL: Security Concurrent Separation Logic'

```
predicate list(x: Ref) {
  acc(x.next) * acc(x.val) *
  low(x.val) *
  x.next != null ==> list(x.next)
}

method m(x: Ref)
  requires list(x)
{ ... }
```

```
predicate list0(x: Ref) {
  acc(x.next0) * acc(x.val0) *
  x.next0 != null ==> list0(x.next0)
}
predicate list1(x: Ref) {
  acc(x.next1) * acc(x.val1) *
  x.next1 != null ==> list1(x.next1)
}
function list_low(x0: Ref, x1: Ref)
  requires list0(x0) * list1(x1)
{
  unfolding list0(x0) in
  unfolding list1(x1) in
    x0.val0 == x1.val1 *
    (x0.next0 != null * x1.next1 != null ==>
      list_low(x0.next0, x1.next1))
}

method m(p0: Bool, p1: Bool,
         x0: Ref, x1: Ref)
  requires p0 ==> list0(x0)
  requires p1 ==> list1(x1)
  requires (p0 * p1) ==> list_low(x0, x1)
{ ... }
```

Figure 4.17.: Relational recursive predicate defining a linked list whose values are low (left) and its transformed version that can be used in a modular product program (right).

For encoding the program heap and object allocation, we used the second option described in Sec. 3.3.3, which is sound only for languages whose references are opaque. This is generally the case in Python, however, as was pointed out by Toby Murray, it is actually possible to access information about the values of the underlying pointers: the default implementation of __str__, i.e., the default string representation of an object contains (a part of) the pointer value. This problem can be mitigated, e.g., by considering this string representation to be high even for low references, but it illustrates the potential danger of unsound assumptions we alluded to in Sec. 2.4.

As explained in Chapter 3, we implemented the modular product program transformation for the existing Viper AST, enriched with new AST nodes for information flow specifications. Compared to the implementation described in the previous chapter, we further extended this transformation to cover a larger part of the Viper language; in particular, we added the ability to transform recursive predicates containing relational assertions, as shown in Figure 4.17. For convenience, we also slightly extended the Viper-based product transformation to directly transform statements that Nagini previously encoded using gotos, such as break and continue statements, that is, we *added* new kinds of AST nodes to the existing Viper AST. This was simpler than requiring the product transformation to be able to deal with arbitrary reducible control flow graphs (though that is possible in principle [50]).

[50]: Collingbourne et al. (2013), 'Interleaving and Lock-Step Semantics for Analysis and Verification of GPU Kernels'

We decided not to implement the additional checks required for possibilistic and probabilistic non-interference on the Viper level, and to add them on the Nagini level instead, i.e., to generate them as part of Nagini's encoding into Viper. Our reason for this is that adding these checks on the Viper level would have led to spurious checks: For example, the encoding for probabilistic non-interference shown in Figure 4.14 adds proof obligations for every branch (i.e., conditional or loop) in the program. However, Nagini's existing encoding sometimes generates branch statements on the IVL level that do not correspond to branches on the source level (like, for example, for field assignments, as shown in Figure 2.13 in Chapter 2).

Our implementation of the product transformation includes several optimizations to the standard modular product encoding shown in the previous chapter:

1. We assume in every method that the activation variable for the first execution is initially true. As a result, the verifier does not have to consider the case that both executions are inactive throughout (which causes overhead, but can never lead to a verification error, and may thus be skipped). Similarly, it does not have to consider *both* the case where the first execution is initially inactive and the second is active, *and* the case where the first execution is initially active and the second is inactive: These two cases are completely symmetrical for non-interference verification, and considering one of them is sufficient.
2. We merge conditionals that branch on the same activation variable into blocks where possible. As an example, according to the product transformation shown in the previous chapter, two subsequent assignments in a program would result in four conditionals, each containing one assignment. In our optimized encoding, we instead merge them into two conditionals that each contain two assignments, reducing the number of branches in the generated program.
3. We omit some checks used for ordinary safety verification for the second execution, since they are already checked in the first execution. This optimization is implemented in a generic way, i.e., a frontend can provide the product transformation with information regarding which checks are to be omitted in the second execution. For Nagini, we omit all checks about variable definedness (this optimization could likely be extended to also avoid checking other properties twice). Note that it is not possible to omit *all* checks of functional properties in the second execution in practice; for example, permission operations like exhales still have to be performed (requiring the checks that ensure permissions are actually present) in order to properly frame information, and checks e.g. for type correctness are required to ensure that all specifications are well-defined.

4.4.2. Implementation in Nagini

For Nagini, we made several changes directly in its existing code base. We added command line options to enable verification of information flow properties, and to specify which property to verify (standard non-interference, possibilistic non-interference, or probabilistic non-interference). We extended Nagini's existing specification language to include information flow specifications, i.e., we added new contract functions like `Low`, `LowEvent`, and `LowVal`. The latter can be used with values of "primitive" built-in types like integer to specify e.g. that the *integer value* of an expression is low, but the *reference* might not be, a distinction that exists in languages like Python which wrap all primitive values into objects: Figure 4.18 shows an example of a method that always returns the

```
def add_zero(i: int, secret: int) -> int:
    # requires low(i)
    # ensures lowVal(result)
    if secret == 0:
        return i + 0
    return i
```

Figure 4.18.: Method whose result does not leak the secret through the returned integer value, but may leak it via the object representing that value; since i + 0 can return an object different from i, if the secret is zero, the returned object may be different from i.

same integer value in both executions, but may return different integer objects depending on secret information.

New contract functions also give programmers the ability to specify and verify the absence of termination channels by using the encoding described in Chapter 3, adapted for use with Nagini's existing methodology for proving termination [40]. We also provide support for easily marking entire methods that do not deal with secret data at all without marking every single parameter and field as low, by using a decorator @AllLow, which signifies that a method does not deal with any sensitive data, and automatically encodes this into preconditions, postconditions and loop invariants.

[40]: Boström et al. (2015), 'Modular Verification of Finite Blocking in Non-terminating Programs'

Since Nagini's existing encoding from Python to Viper is almost entirely operational or obviously operational-equivalent, we only adapted the encoding of dynamically-bound calls as shown in Sec. 4.2.1, and re-implemented the encoding of AST nodes that previously used **goto** statements using the new AST nodes mentioned above.

Since Nagini's encoding of concurrent programs is based on implicit dynamic frames [199] and principles of concurrent separation logic [169], we could modify its existing encoding by adding the checks shown in Sec. 4.3 to prove both possibilistic and probabilistic non-interference for concurrent programs. In the encoding for probabilistic non-interference, we also added another optimization: Since the encoding ensures that the control flow of both executions always *stay* aligned, it is sound in this case to always assume that the control flow is also *initially* aligned, i.e., that the values of both activation variables are equal at the beginning of each method. This assumption (soundly) removes the need for the verifier to consider executions with differing control flow, which, as we will show in the evaluation, can have a big impact on performance in practice.

[199]: Smans et al. (2012), 'Implicit dynamic frames'

[169]: O'Hearn (2007), 'Resources, concurrency, and local reasoning'

Finally, we adapted the back-translation of errors and counterexamples in Nagini: In particular, when verifying non-interference and getting a counterexample for a verification error in a product program, Nagini translates this information into a counterexample that cleanly shows the states of a *pair* of executions that lead to a verification error, instead of showing only a single state. Figure 4.19 shows a counterexample returned by Nagini for a verification error stating that a relational property might not hold.

The Viper extension for product programs[5] and the extended version of Nagini[6] are open source and available online. Further information about the implementation can be found in [148].

[5]: `https://github.com/viperproject/silver-sif-extension`

[6]: `https://github.com/marcoeilers/nagini`

[148]: Meier (2018), 'Verification of information flow security for Python programs'

```
Verification failed
Errors:
Postcondition of abs might not hold.
Assertion Low(Result()) might not hold. (arttest.py@5.12).
First execution:
Old store:
  x -> False
Old heap: Empty.
Current store:
  x -> False,
  Result() -> False
Current heap: Empty.

Second execution:
Old store:
  x -> True
Old heap: Empty.
Current store:
  x -> True,
  Result() -> True
Current heap: Empty.
```

Figure 4.19.: Counterexample to a relational assertion, showing states of two different executions.

4.4.3. Performance Overhead of the Product Construction

Our first goal is to evaluate the performance overhead generated by the product construction. To this end, we compared the verification times of Nagini's entire functional test suite with and without the product transformation enabled. That is, we artificially forced the construction of a product program without adding relational specifications, so that no additional properties are verified, but the overhead resulting from verifying the (more complex) product program instead of the original program can be measured. The test cases range from small programs targeting specific language or specification constructs, to realistic code examples taken from programming tutorials. We ran each test five times on a warmed-up JVM with the information flow extension enabled and disabled, without adding any information flow specifications. Our test system was a 12 core AMD Ryzen 3900X with 32GB of RAM running Ubuntu 20.04.1.

We summarize our main findings here; all test cases and measured times are shown in Table B.1 in App. B. All tests report the same results with and without the product transformation (i.e., either verification succeeds in both settings or the same verification errors are reported), meaning that soundness and completeness are not impacted by the extension, and that we can indeed still reason about the entire language subset supported by Nagini (even though we only had to *implement* the product construction for the much simpler Viper IVL). Without the product transformation, each test case takes between 3 and 9 seconds, with the majority taking between 3 and 5. For most cases, enabling the product construction leads to an increase in verification time that is clearly acceptable (less than 11% for half the tests, less than 30% for three quarters, and less than 100% for 90% of the tests). For five test cases, the slowdown is a factor between 5 and 12, and a single outlier (a quicksort implementation) has a slowdown factor of 17.5 and a resulting verification time of two minutes. We believe that the main reason for the large slowdown for these particular test cases is the extensive use of quantifiers in their specifications (e.g., to specify properties of all elements in a list). Quantifier handling is difficult for automated verification in general, because unbounded chains of quantifier instantiations can occur during the proof search [58], and this problem seems to be exacerbated when using the product encoding. We found examples where this problem (i.e., bad heuristics leading to unnecessarily many quantifier instantiations) was already present in the original program, but was exacerbated when using the product encoding; fixing the underlying problem by restricting quantifier instantiations then led to slightly improved performance of the original program but greatly improved performance of the product version.

[58]: Detlefs et al. (2005), 'Simplify: a theorem prover for program checking'

We conclude that the performance impact of the product transformation is acceptable for most examples, but can be significant for programs with complex functional specifications. Nevertheless, those programs can still be verified, and modular verification enables scaling to larger systems.

4.4.4. Expressiveness and Comparison with SecC

In a second step, we evaluated the expressiveness and performance of our implementation on a number of challenging examples from the literature. In particular, we use the information flow examples from the evaluation of the previous chapter (sequential examples collected from various previous papers) translated to Python and from this chapter, both shown in Table 4.1, as well as examples taken from SecC [72], the only other automated and modular verification tool for concurrent programs we are aware of, shown in Table 4.2.

[72]: Ernst et al. (2019), 'SecCSL: Security Concurrent Separation Logic'

Our examples represent the state of the art in automated information flow verification, requiring semantic reasoning that would not be possible in a type system, and using complex information flow specifications including declassification, termination-sensitive non-interference, and value-dependent sensitivity [160]. As show in the previous chapter, these features can be easily encoded into modular product programs.

[160]: Murray et al. (2018), 'COVERN: A Logic for Compositional Verification of Information Flow Control'

Table 4.2 includes the CDDC case study [160], which models an embedded device that interacts simultaneously with multiple users and classified networks, and an example (Declassify) from a different paper [194] that uses several non-trivial declassification policies and has been encoded using SecC (and now, Nagini). For one of the SecC examples (DB) whose original version uses arrays, pointer arithmetic, and recursive predicates containing relational assertions, we created two versions in Nagini: One that also uses relational recursive predicates and, since pointer arithmetic is not possible in Python, a linked list (which results in very similar specifications), and one (DB-List) that uses Python's built-in list and expresses specifications using quantifiers; the latter is arguably much more natural in Python, but may be more difficult to verify.

[160]: Murray et al. (2018), 'COVERN: A Logic for Compositional Verification of Information Flow Control'

[194]: Schoepe et al. (2020), 'VERONICA: Expressive and Precise Concurrent Information Flow Security'

Additionally, we verified several of the SecC examples using two different settings: SecC, which is designed specifically for verifying concurrent programs, *always* requires that all branches are low and both executions thus follow the same path, similar to our approach for probabilistic non-interference described in Sec. 4.3.3. Thus, in a context with probabilistic thread scheduling, SecC also ensures probabilistic non-interference.[7] When applied to sequential programs, the requirement of low branching is unnecessary (unless one aims for a constant time implementation, as in the CT example) but sound; as a result, SecC soundly but incompletely verifies standard non-interference when applied to sequential programs

[7] While the soundness result of the underlying logic was formalized for a setting with a deterministic scheduler, in which case this approach guarantees ordinary non-interference (as we also pointed out in Sec. 4.3.3), we will continue to refer to the property proved as probabilistic non-interference to simplify the presentation.

Table 4.1.: Programs evaluated for proving information flow security. We show the total lines of code (LOC) including implementation and specification but excluding whitespace, lines of specification and proof annotation (Ann.), the property we proved (Prop., where NI = non-interference, TNI = termination sensitive non-interference, PS = possibilistic non-interference, PR = probabilistic non-interference) and the verification time in seconds (T), averaged over five runs.

	LOC	Ann.	Prop.	T
banerjee	77	21	NI	5.19
constanzo	21	12	NI	5.39
darvas	38	18	NI	4.20
example	27	12	NI	5.39
Example-decl	27	12	NI	5.76
Example-term	8	4	TNI	3.59
joana-1-tl	22	7	NI	3.87
joana-2-bl	13	5	NI	3.64
joana-2-t	12	4	NI	3.72
joana-3-bl	36	15	TNI	3.55
joana-3-br	33	14	TNI	4.60
joana-3-tl	23	9	TNI	4.50
joana-3-tr	25	10	TNI	4.19
joana-13-l	11	2	NI	4.54
kusters	28	12	NI	4.35
naumann	27	17	NI	8.46
product	39	18	NI	11.35
smith	39	21	NI	6.81
terauchi1	10	3	NI	3.59
terauchi3	19	6	NI	3.69
terauchi4	18	8	NI	3.97
Fig. 4.3	19	6	NI	3.82
Fig. 4.11	53	17	PS	4.92
Fig. 4.12	24	11	PS	4.19
Fig. 4.9	23	8	PS	4.37
Fig. 4.10	36	15	PS	4.40
Fig. 4.13	34	15	PS	4.57
Fig. 4.16	38	17	PR	9.60
Fig. 4.18	7	3	NI	2.56

Table 4.2.: Comparison with SecC. We show the total lines of code and lines of specification for Nagini (LOC_N, Ann_N) and SecC (LOC_S, Ann_S), the property we proved (Prop., where NI = non-interference, PR = probabilistic non-interference) and the verification time in seconds in both tools (T_N and T_S) and in Nagini *without* the product construction (T_{NP}), averaged over five runs.

	LOC_N	Ann_N	LOC_S	Ann_S	Prop.	T_S	T_N	T_{NP}
SecC CAV	40	13	50	11	PR	0.81	3.00	2.42
SecC CDDC	278	105	214	47	PR	9.53	33.93	7.54
SecC CT	64	35	211	159	PR	0.99	3.56	2.66
SecC DB	260	197	256	167	PR	1.37	10.53	4.06
SecC DB	260	197	-	-	NI	-	12.73	4.06
SecC DB-List	100	48	-	-	PR	-	15.50	4.47
SecC DB-List	100	48	-	-	NI	-	107.28	4.47
SecC Encrypt	29	12	49	18	PR	0.85	3.16	2.41
SecC Encrypt	29	12	-	-	NI	-	3.50	2.41
SecC Declassify	218	148	265	172	PR	1.08	5.17	3.31

(for example, it would not be able to verify the examples joana-2-t and joana-13-l from Table 4.1). For the sequential examples that are not aiming for constant time, i.e., DB (both versions) and Encrypt, we run Nagini both in ordinary non-interference mode (which is more complete for sequential programs but requires reasoning about potentially differing control flow between executions) and in probabilistic non-interference mode (which mirrors SecC in the generated proof obligations).

Nagini was able to verify all correct examples and correctly flagged incorrect ones, which demonstrates that our approach can handle concurrent implementations and express complex non-interference properties. For the examples from Table 4.1, Nagini takes between 3 and 12 seconds each, which we believe is clearly acceptable.

For the examples taken from SecC, Nagini takes 3-5 seconds for five of them and 10-15 seconds for three variations of the DB example, which we believe is also acceptable. Two examples take longer: The CDDC example, which is a complex case study, takes 34 seconds, and the list-version of the DB example, when verified in ordinary non-interference mode (which, as described, is more complete for sequential programs than the probabilistic non-interference mode), takes 107 seconds. The latter demonstrates again that an abundance of quantifiers in specifications can lead to bad performance. However, performance even for this example is acceptable when verified in probabilistic non-interference mode, meaning that the extreme negative performance impact of quantifiers is only seen if the verifier has to consider different control flow between both executions. As expected, the versions of the same example that use recursive predicates instead of quantifiers take less time, and every example takes less time when verified in probabilistic non-interference mode than in ordinary non-interference mode.

Table 4.2 shows that SecC is faster than our implementation for all five examples and configurations it is able to verify, by a factor of 3 to 8; for the biggest example (CDDC), the performance difference is around a factor of 3.5. However, this difference cannot be blamed entirely on the product construction since, as shown in the table, Nagini is slower than SecC for four of these five examples even *without* verifying non-interference, due to both architectural reasons and, more importantly, the additional properties that have to be verified in Python code due to its more dynamic nature; direct performance comparisons are therefore difficult.

We conclude that Nagini matches SecC in its expressiveness, and is in fact more complete for sequential programs as discussed before; while its performance is significantly worse for all examples, it is not worse by an order of magnitude, so if it is feasible to verify a program with SecC in terms of performance, it is likely also feasible to verify it with Nagini. The main exception we observe is the verification of programs with many quantifiers in specifications in Nagini's complete ordinary non-interference mode, but as stated before, such a mode is not available in SecC in the first place (and its general ability to deal with quantifiers is also more limited than Nagini's). Additionally, we stress that SecC was designed and implemented for information flow verification for concurrent programs from scratch (which meant it could be optimized for relational property verification and, even more specifically, for the scenario where both executions' control flow never diverges) without being able to reuse code from an existing verifier, whereas our extended Nagini version could be implemented with minimal effort.

4.5. Related Work

In this section, we discuss other existing work related to the combination of product constructions and IVL encodings, or to the verification of non-interference properties for concurrent programs. Note that we will not discuss product constructions in general or sequential information flow security verification, as those topics have been covered in the previous chapter.

To the best of our knowledge, SymDiff [118] for the Boogie IVL is the only existing tool that constructs product programs on an IVL level. SymDiff is a tool for differential program verification, which requires reasoning about pairs of executions of two different (but related) programs and is thus similar to hyperproperty verification; in fact, SymDiff has also been used to verify non-interference in the past [4]. The authors of SymDiff have proposed different techniques for modularly proving mutual function summaries, similar to relational specifications, one of which uses a kind of product construction [98, 119]. However, they do not examine potential soundness problems arising from this approach, nor do they discuss if it can be applied to concurrent source programs.

In addition to SymDiff, there are some other tools that perform analyses or program transformations on the level of the Boogie IVL (e.g., Corral [121] performs stratified inlining). We believe that it is possible that, like for

[118]: Lahiri et al. (2012), 'SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs'

[4]: Almeida et al. (2016), 'Verifying Constant-Time Implementations'

[98]: Hawblitzel et al. (2013), 'Towards Modularly Comparing Programs Using Automated Theorem Provers'
[119]: Lahiri et al. (2013), 'Differential assertion checking'

[121]: Lal et al. (2013), 'Reachability Modulo Theories'

product constructions, there are hidden soundness conditions for IVL encodings that apply to these tools.

The phenomenon that non-interference is not necessarily preserved by some transformations that do preserve trace properties (like, in our case, the encoding into an IVL) is not entirely new and also exists, for example, in refinement [91]. However, the underlying reasons are different: in refinement, resolving non-determinism can introduce information flows, whereas in our case, IVL encodings (which, if anything, tend to *introduce* non-determinism) can *hide* unwanted information flows.

There are various existing type systems (e.g. [161, 203]) and static analyses (e.g. [45, 89]) for proving information flow security of concurrent programs. Compared to verification based on product programs or specialized program logics, these are generally more automated, but less precise. As discussed before, type systems and static analyses exist for proving possibilistic non-interference (by forbidding secret-dependent non-termination) [205], probabilistic non-interference (by forbidding branches on high values, sometimes only if the branches have observably different effects) [201, 202], observational determinism (by forbidding competing variable modifications) [233], as well as other forms of information flow security [89].

Moreover, there are several dedicated program logics for information flow verification for concurrent programs: Covern [160] builds on rely-guarantee-reasoning [111] and adds some elements of CSL, though it stops short of supporting features like pointers or arrays. The latter are supported in its successor logic, SecCSL, which is a full extension of CSL for information flow verification. Veronica [194] builds on Owicki-Gries style reasoning [170], separates functional verification from information flow verification by requiring functional information as inputs, and supports complex declassification policies. All three of these logics work in the same setting as our approach, allowing inter-thread communication only through locks, and their central way of preventing information flows resulting from internal timing channels is the same as in the type systems mentioned above and in our approach for ensuring probabilistic non-interference: they either forbid branching on high data, or, in Veronica's case, allow branching on high values only if all branches are indistinguishable to the attacker, i.e., have for example the same timing behavior. The only logic of these three that has been automated is SecCSL, whose implementation in SecC we compared against in the previous section. One example in our evaluation (Declassify) is an extended example from the Veronica paper, encoded into SecC's and Nagini's syntax, showing that these tools are also able to encode non-trivial declassification policies even without dedicated support in the logic. The implementation of SecC additionally goes beyond the logic it implements by also supporting rely-guarantee reasoning.

More recently, two logics have enabled verification for more complex settings: SecRSL [230] targets concurrent programs executed in relaxed memory models and builds on relaxed separation logic (RSL) [217], and SeLoC [86], which is built on the Iris framework [112], targets fine-grained concurrency verification using protocols. The latter also overlays a type system on top of the logic, making it possible to use both (automatable) type checking and (precise) verification on different parts of a program.

[91]: Graham-Cumming et al. (1991), 'On the Refinement of Non-Interference'

[161]: Myers et al. (2001), 'Jif: Java information flow'
[203]: Smith (2007), 'Principles of Secure Information Flow Analysis'

[45]: Chen et al. (2014), 'Hybrid Information Flow Analysis for Python Bytecode'
[89]: Giffhorn et al. (2015), 'A new algorithm for low-deterministic security'

[205]: Smith et al. (1998), 'Secure Information Flow in a Multi-Threaded Imperative Language'

[201]: Smith (2001), 'A New Type System for Secure Information Flow'
[202]: Smith (2006), 'Improved typings for probabilistic noninterference in a multi-threaded language'

[233]: Zdancewic et al. (2003), 'Observational Determinism for Concurrent Program Security'

[89]: Giffhorn et al. (2015), 'A new algorithm for low-deterministic security'

[160]: Murray et al. (2018), 'COVERN: A Logic for Compositional Verification of Information Flow Control'

[111]: Jones (1981), 'Developing methods for computer programs including a notion of interference'
[194]: Schoepe et al. (2020), 'VERONICA: Expressive and Precise Concurrent Information Flow Security'

[170]: Owicki et al. (1976), 'An Axiomatic Proof Technique for Parallel Programs I'

[230]: Yan et al. (2021), 'SecRSL: security separation logic for C11 release-acquire concurrency'

[217]: Vafeiadis et al. (2013), 'Relaxed separation logic: a program logic for C11 concurrency'

[86]: Frumin et al. (2021), 'Compositional Non-Interference for Fine-Grained Concurrent Programs'

[112]: Jung et al. (2015), 'Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning'

Instead of statically proving that a program does not leak high data, some techniques aim to prevent unwanted information leakage for arbitrary programs at runtime, including leakage as a result of internal timing channels. Different techniques achieve this goal either by modifying the language runtime or the scheduler [218], by using information flow aware concurrency primitives, which can be implemented as a library [208], or by transforming the original program [185]. Such techniques have the advantage of allowing to run programs that would be rejected by static techniques, but at the cost of requiring specific runtimes, additional synchronization, or affecting execution performance in other ways.

As already discussed, most product program constructions [23, 119], including modular product programs, are defined explicitly for sequential languages. Self-composition [25] can be used with any programs, including concurrent ones, in principle, but (in addition to not allowing for method-modular verification) it offers no support for relating intermediate states of different executions, which makes it difficult (and in practice often infeasible) to relate the final states of two executions which may have had completely different thread schedules, since, in principle, any combination of schedules has to be considered. The only product program construction we are aware of that explicitly supports concurrency [78] partly addresses this problem by creating a reduced product that only represents *relevant* combinations of thread schedules; however, for large programs, we believe that this approach will still suffer from scalability problems. In contrast, our approach of constructing a sequential product program of a sequential IVL-encoding of a concurrent source program is thread-modular and therefore does not have to explicitly reason about different interleavings at all.

[23]: Barthe et al. (2011), 'Relational Verification Using Product Programs'
[119]: Lahiri et al. (2013), 'Differential assertion checking'

[25]: Barthe et al. (2011), 'Secure information flow by self-composition'

[78]: Farzan et al. (2019), 'Automated Hypersafety Verification'

4.6. Conclusion

We presented an approach for retrofitting existing IVL-based program verifiers to check non-interference properties using product programs. This approach allows reusing existing frontends to reduce the required implementation effort. We have shown when this technique is sound, that it can incorporate concurrency, and that it can be implemented in an existing verifier with acceptable performance.

5. Rich Specifications for Modular Smart Contract Verification

Smart contracts are programs that execute inside blockchains such as Ethereum [229], and allow the execution of resource transactions between different parties without the need for a trusted third party. Smart contracts tend to be comparatively short programs, which makes the use of formal methods viable and economical in practice. In addition, since they handle large amounts of currency and are non-trivial to implement correctly, there are huge incentives for attackers to find and exploit any vulnerabilities for their own, sometimes significant, financial gain (as demonstrated, e.g., by the attack on TheDAO [94]).

Proving smart contracts secure poses special challenges that do not apply to most standard kinds of applications, mainly because they execute in and are forced to interact with a completely open and adversarial environment: Their public functions can be called both by external attackers and by other (malicious) contracts, which anyone can freely deploy; conversely, most contracts are forced to themselves perform calls to other, potentially malicious, contracts as part of their intended workflow. Additionally, the contracts themselves (i.e., their bytecode) and their current state as well as all communication are public and therefore visible to attackers.

This has a number of consequences for proving standard security properties:

- Since all state is public, many standard notions of confidentiality (including non-interference) do not apply; specific privacy properties can instead be enforced using Non-Interactive Zero Knowledge proofs [209].
- Integrity properties are of utmost importance: It must not be possible for malicious calls to the contract, or calls from the contract itself to malicious outside contracts, to corrupt a contract's state or lead to unintended functionality. Most bugs that lead to immediate benefits for attackers are the result of key contract invariants being violated, often through the use of *re-entrant* calls, i.e., callbacks from functions called by the contract itself.
- Availability of the contract's functions themselves is guaranteed by the underlying blockchain (which we will not focus on) and contracts cannot get stuck due to non-termination, since each execution terminates once its (finite) budget of *gas*, which is used to execute any contract statement, is depleted. However, contracts can potentially reach states in which some vital functionality (e.g., typically, the ability for an outside contract to withdraw currency they are owed) can no longer be executed, because any attempt to do aborts for one reason or another.

In this chapter, which is based on the OOPSLA 2021 paper "Rich Specifications for Ethereum Smart Contract Verification" [43], we aim to *modularly* prove integrity and, to some degree, availability properties of Ethereum smart contracts, and address the following three challenges: (1) Since smart contracts execute in an adversarial environment, standard modular reasoning techniques such as separation logic [180], which reason about

[229]: Wood et al. (2014), 'Ethereum: A secure decentralised generalised transaction ledger'

[94]: Güçlütürk (2018), *The DAO Hack Explained: Unfortunate Take-off of Smart Contracts*

[209]: Steffen et al. (2019), 'zkay: Specifying and Enforcing Data Privacy in Smart Contracts'

[43]: Bräm et al. (2021), 'Rich specifications for Ethereum smart contract verification'

[180]: Reynolds (2002), 'Separation Logic: A Logic for Shared Mutable Data Structures'

calls under the assumption that *all* code is verified and obeys a set of basic rules, do not apply in this setting. This problem is exacerbated by the presence of re-entrant calls. A sound reasoning technique must account for *all* behaviors a call to an unverified contract could possibly exhibit, which requires novel ways of specifying the calling contract. (2) Many contracts collaborate with some small set of trusted outside contracts, forming distributed applications. The integrity of such applications often depends on invariants spanning multiple contracts. To decouple collaborating contracts, their implementations are often hidden behind interfaces. To enable *contract-modular* verification, these interfaces need to be equipped with specifications that provide enough information to allow verification. (3) Typical smart contracts are primarily concerned with modeling and executing resource transactions of different kinds. Examples range from simple token contracts to escrow implementations, ICOs, and complex decentralized finance (DeFi) applications. However, even though notions such as resource ownership and agreed exchanges are central to programmer intentions, resources themselves are often implicit in smart contract implementations. As a result, even when formal methods are used to prove smart contract correctness according to some specification, this discrepancy between high-level intentions and low-level implementations makes it difficult to correctly specify intended behavior on the implementation level, potentially leading to subtle and costly mistakes.

Existing (automated) verifiers for smart contracts do not fully address these challenges. Some verifiers [79, 120, 212, 213] prove specific properties, e.g., the absence of overflows or re-entrancy bugs, but cannot be used to prove full functional correctness of a contract. Other verifiers [96, 103, 113, 174, 226] aim to prove arbitrary, user-defined properties using variations of established specification and verification techniques. However, because these techniques are not sufficiently adapted to the setting of smart contracts, they are either not generally applicable or very imprecise in the presence of arbitrary re-entrancy. In general, no existing technique allows reasoning modularly about compositions of multiple smart contracts while preserving interface abstractions. Furthermore, existing techniques offer limited support for specification and reasoning in terms of high-level notions of custom resources such as tokens.

In this chapter, we propose a novel specification and verification methodology for the sound, unbounded verification of security properties of Ethereum smart contracts. Our goal is to specify and verify general safety properties, which will enable us to ensure contract integrity and, to some degree, availability. To that end, we offer specification constructs tailored to the domain of smart contracts, enabling users to prove strong functional correctness properties of arbitrary smart contracts, with specifications that capture their intended resource manipulations explicitly.

We make four main contributions:

(1) Reasoning in the presence of unverified code: To the best of our knowledge, we present the first smart-contract verification technique that is sound and precise in the presence of calls from and to unverified and potentially malicious code with arbitrary re-entrancy. Our technique: (a) identifies and proves properties which cannot be invalidated by calls to malicious code, including vital properties such as access control (which lacks

[79]: Feist et al. (2019), 'Slither: a static analysis framework for smart contracts'
[120]: Lai et al. (2020), 'Static Analysis of Integer Overflow of Smart Contracts in Ethereum'
[212]: Tikhomirov et al. (2018), 'SmartCheck: Static Analysis of Ethereum Smart Contracts'
[213]: Tsankov et al. (2018), 'Securify: Practical Security Analysis of Smart Contracts'
[96]: Hajdu et al. (2019), 'solc-verify: A Modular Verifier for Solidity Smart Contracts'
[103]: Hildenbrandt et al. (2018), 'KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine'
[113]: Kalra et al. (2018), 'ZEUS: Analyzing Safety of Smart Contracts'
[174]: Permenev et al. (2020), 'VerX: Safety Verification of Smart Contracts'
[226]: Wesley et al. (2022), 'Verifying Solidity Smart Contracts via Communication Abstraction in SmartACE'

direct language support and must be implemented manually in smart contracts), and (b) provides specification constructs that abstract over *all* modifications a call can *potentially* make, exploiting language-level encapsulation guarantees.

(2) Modular reasoning about collaborating smart contracts: We demonstrate the challenges of verifying collaborating contract structures, in particular, proving invariants spanning multiple contracts in an adversarial environment. We show that our specification methodology can express all required information at the interface level, forming the basis of the first *modular* verification technique for collaborating smart contracts.

(3) Intuitive specifications of resource manipulation: We introduce a specification mechanism for contracts which manage resources and resource transactions. Specifications are expressed directly at the abstraction level of transferring, exchanging, and loaning resources, which results in concise and intuitive specifications and, effectively, makes smart contract specifications more similar to actual business contracts. Ubiquitous integrity properties of resources such as ownership, access control, and non-duplicability are baked into our system, avoiding potentially repetitive and error-prone boilerplate specifications; violations of these properties are found by default. While other blockchain systems build (different kinds of) resources directly into the language [35], those do not always guarantee the integrity properties built into our approach; additionally, our approach is the first to enable reasoning about custom resources in smart contracts systems without dedicated language support.

[35]: Blackshear et al. (2019), *Move: A language with programmable resources*

(4) Implementation and evaluation: We implemented our approach in 2Vyper, an automated, SMT-based verification tool for the Vyper language [75] for Ethereum smart contracts. To the best of our knowledge, 2Vyper is the first verification tool specifically aimed at Vyper contracts. It supports the entire Vyper language as of version 0.2.0 and allows specifying contracts and interfaces in the form of readable, source-level code annotations. Our evaluation shows that 2Vyper enables automated verification of strong correctness properties of (collaborating) real-world contracts with reasonable performance and annotation overhead. In particular, we demonstrate that 2Vyper can verify contracts that use re-entrancy patterns not supported by other verification tools, and that it enables modular verification of collaborating smart contracts used in practice.

[75]: Ethereum (2021), *Vyper documentation*

The chapter is structured as follows: We introduce Ethereum smart contracts in Sec. 5.1. In Sec. 5.2, we informally introduce the specification constructs we use to reason about contracts containing re-entrant calls; subsequently, we show how they can be used to reason about collaborating contracts in Sec. 5.3. We introduce our resource-based specification approach in Sec. 5.4, discuss availability properties in Sec. 5.5, and present our verification technique in the form of a Hoare logic in Sec. 5.6. We describe our implementation in 2Vyper and evaluate it in Sec. 5.7. We discuss related work in Sec. 5.8 and conclude in Sec. 5.9.

```
beneficiary: address
highestBid: int256
highestBidder: address
ended: bool
pendingReturns: map(address, int256)
lock: bool

def bid():
  assert not self.lock and msg.value > self.highestBid and not self.ended
  self.pendingReturns[self.highestBidder] += self.highestBid
  self.highestBidder = msg.sender
  self.highestBid = msg.value

def withdraw():
  assert not self.lock
  toSend = self.pendingReturns[msg.sender]
  self.pendingReturns[msg.sender] = 0
  self.lock = True
  send(msg.sender, value=toSend)
  self.lock = False

def end():
  assert not self.lock and not self.ended and msg.sender == self.beneficiary
  self.ended = True
  self.lock = True
  send(self.beneficiary, value=self.highestBid)
  self.lock = False
  self.highestBid = 0
```

Figure 5.1.: Simplified auction contract written in Vyper. assert statements revert the current transaction, whereas send statements send Ether to another contract. Since the contract does not have an explicit constructor function, all fields are initialized with default values.

5.1. Ethereum Smart Contracts

Ethereum smart contracts are programs usually written in a high-level language, most commonly Solidity [74] or the newer Vyper [75] language, and then compiled to bytecode for execution in the Ethereum Virtual Machine (EVM) [229]. Fig. 5.1 shows an example of a Vyper smart contract implementing an auction. Note that in this example and throughout the chapter, we use a simplified presentation of Vyper and Ethereum contracts and omit details that are irrelevant to our approach (e.g., that Ether can be transferred only by calling functions marked as payable, or that Vyper functions revert when encountering under- or overflows[1]). We also ignore the fact that contract execution consumes *gas*, i.e., a fixed cost associated with every executed instruction, which is not relevant for proving safety properties, the focus of this chapter.

[74]: Ethereum (2021), *Solidity documentation*

[75]: Ethereum (2021), *Vyper documentation*

[229]: Wood et al. (2014), 'Ethereum: A secure decentralised generalised transaction ledger'

A contract can declare *fields* that form its persistent state. In our example, the contract stores the beneficiary of the auction, the current highest bid and bidder, and the amounts of *wei*, a sub-unit of Ethereum's built-in currency *Ether*, it owes to bidders who have been outbid. In addition to explicitly declared fields, every contract has a built-in balance field that tracks the amount of Ether currently held by the contract. Unlike ordinary fields, which can be written to directly by the contract (but, crucially, *not* by other contracts), the balance cannot be written to directly. Ether is the only resource with native language support; programmers can, however, implement smart contracts that provide custom resources (often called *tokens*), illustrated later in this section.

Contracts define a set of functions and a special constructor function called __init__ that is executed when the contract is set up. Smart contracts are executed as *transactions*: a caller outside the blockchain can request to invoke a contract's function, and miners can then decide to execute

[1] Our tool nonetheless allows one to verify that a function does not revert due to under- or overflows.

```
minter: address
balances: map(address, int256)

def transfer(from: address, to: address, amount: uint256):
  assert self.balances[from] >= amount and msg.sender == from
  newAmount: int256 = self.balances[from] - amount
  self.balances[to] += amount
  self.balances[from] = newAmount
  to.notify(from, self, amount)

def mint(to: address, amount: uint256):
  assert msg.sender == self.minter
  self.balances[to] += amount
```

Figure 5.2.: Simplified token contract implemented in Vyper. The minter can create new tokens by calling mint; other users call transfer to give their own tokens to another user. assert statements ensure that the transaction reverts if a user tries to spend tokens they do not own.

this function as part of the next block. If this happens, the function is executed, and may in turn call functions of the same or other contracts as part of the same transaction[2]. (Note that throughout this chapter, we inline internal calls to private functions for simplicity.) External calls typically occur via *interfaces* that list (a subset of) the available functions of a contract. Importantly, there is no observable concurrency while all transitively-called functions are executed.

The intended workflow of the auction contract is that clients call the bid function and transfer along a larger amount of Ether than the current highest bid. If another client bids a higher value later, the contract updates pendingReturns to remember that no-longer-highest bidders can get their Ether back. Such a bidder can call withdraw to have the Ether transferred back to them. Contracts can transfer Ether to other contracts via send statements (e.g., in function end), where the parameter value specifies the transferred amount, or by calling a function (like the bid function) on another contract and implicitly passing along some amount of Ether. Internally, these are the same: executing a send statement is implemented by calling a default function on the recipient.

Ethereum transactions can *revert*, meaning they abort and all state changes they made are reset, for several reasons. Smart contracts commonly use assert statements to revert a transaction if the asserted condition is false. This is intended behavior used to enforce that e.g., arguments supplied to a call are valid and that the call is allowed given the current state of the called contract. For example, a call to the end function will revert if the auction is already over, and bid reverts if the new bid is not higher than the current highest bid. This contract also reverts if called while the lock field is set, a pattern commonly used to explicitly prevent a contract from being called in unexpected situations (often to prevent re-entrancy vulnerabilities, discussed below).

In addition to the contract's fields and explicitly declared arguments, a contract can always access the implicit arguments msg and block, which contain information about the current call and the block the current transaction is a part of. For example, msg has the particularly important field msg.sender, which contains the address of the caller of the current function. Function end uses this variable to ensure that only the beneficiary of the auction can end the auction, whereas the bid function uses msg.value to obtain the amount of Ether sent with the call.

Custom resources. While the auction contract works directly with the built-in Ether currency, many real contracts implement or work with

[2] Throughout this chapter, when we say that a contract interacts with other contracts, we mean other contract *instances*, i.e., contracts deployed at other addresses that may contain the same or (usually) different code from our contract.

[220]: Vogelsteller et al. (2015), 'EIP-20: ERC-20 Token Standard'

tokens [220], i.e., custom currencies tracked via ad-hoc implementations in smart contract fields. Fig. 5.2 shows a very simple version of a token contract. Its state consists of a map that represents the balances that each other contract holds for this token. Contracts can call transfer to transfer tokens from one contract to another, which simply corresponds to updating the map. This contract enforces important properties common to resources in general: Each client holding a balance should *be able to transfer only tokens that it owns*. This implicit notion of resource *ownership* (tracked via numeric values in a map here) is a native notion in our specification methodology, explained in Sec. 5.4. This contract's implementation enforces this intention by reverting if it is asked to transfer tokens away from anyone except the caller. Similarly, the right to mint *new* tokens is restricted to a special privileged contract (represented by its *address*), **self**.minter.

The token contract can also be used to illustrate the infamous concept of *re-entrancy vulnerabilities*: the subtle potential for a called contract to perform malicious callbacks and achieve undesirable outcomes. Say, for example, that lines 8 and 9 in the token contract were swapped, i.e., the contract first called the receiver contract to notify it that it has received tokens, *before* reducing their balance. If the notified contract called the sender of the transaction, it could in turn call back into the token contract and transfer the tokens it just transferred away *a second time*; in particular, the **assert** on line 5 would not prevent the transfer because the balance was not yet updated at the time the callback happens. This would allow clients of the token contract to create tokens out of thin air. Variations of this pattern, which was first described in 2008 [223], are behind most re-entrancy vulnerabilities, e.g., the infamous DAO exploit [94]; as we will show in Sec. 5.4, our explicit resource reasoning will uncover such coding errors by default.

[223]: Wagner (2008), *An attack on a mint*

[94]: Güçlütürk (2018), *The DAO Hack Explained: Unfortunate Take-off of Smart Contracts*

5.2. Verification in the Presence of Untrusted Code and Re-Entrancy

Smart contracts frequently interact with other contracts; in particular, contracts that offer services to arbitrary clients often call functions on arbitrary other contracts. For example, the auction contract above sends Ether to (i.e., calls) an arbitrary **msg**.sender in its withdraw function, which is necessary to ensure that any contract that has previously placed a bid and has been outbid can get back the Ether they sent. While calls to functions of the same contract (*internal* calls) can simply be verified by inlining the callee function, calls to other contracts (*external* calls) are challenging for two reasons. First, as we explained earlier, the implementations of other contracts are in general untrusted and not verified, and therefore do not have trustworthy specifications (e.g., standard pre- and postconditions) we could use to reason about them. Second, the implementations of other contracts are in general not known: we cannot make any assumptions about the callbacks they perform directly or via other contracts[3]. That is, we do not know if an external call simply modifies the state of the called contract and then returns, or if it triggers more complicated interactions such as those shown in Fig. 5.3: Contract A calls contract B, which performs a re-entrant call to A; subsequently, A performs an external call

[3] It is common to limit the amount of gas sent with a call so that it is not sufficient to perform callbacks, but relying on this is considered bad practice, since the gas cost of Ethereum Virtual Machine instructions can change.

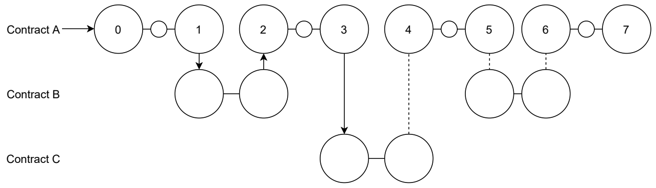


Figure 5.3.: Example of a smart contract transaction involving re-entrancy. Calls and returns are marked by arrows and dashed lines, respectively.

to a third contract C before all calls return. In this scenario, when A's call to B returns, its own state may have changed as a result of the re-entrant call from B. Consequently, if the implementation of B is unknown, one does not know which functions of A have been re-entrantly called (if any), and how A's state has changed as a result. In this section, we present a specification and verification technique that is sound in the presence of unverified code and arbitrary re-entrancy. In the following, we assume that all calls are external (internal calls are inlined) and that the implementation of the callee is neither known nor verified (trusted, known and verified callees could provide stronger assumptions about the effects of the calls, but we focus on the common, most difficult case here).

5.2.1. The Challenge

The problem of re-entrancy by itself is not specific to smart contracts; it can also occur, for example, between objects in object-oriented programming languages. When reasoning about programs in such languages, this problem is usually solved by constraining what a called function may assume about the state and which parts of the heap it may manipulate [19, 115, 157, 180]. However, these verification techniques require that *all* executed code adheres to the rules of the verification technique. In particular, called functions are typically verified to *only* cause side effects allowed by the verification technique. In a smart contract setting, however, external code is often unverified, potentially malicious, and cannot be soundly assumed to follow *any* particular rules beyond those of the execution environment. For the same reason, classical *preconditions* on public functions are of limited use in this setting, since one cannot rely on external callers actually respecting them; any contract that critically relies on preconditions it does not enforce itself must be considered insecure. In order to reason soundly, we have to conservatively assume that any external call may lead to arbitrary callbacks into the original contract and, in particular, mutate the original contract's state in any possible way.

[19]: Barnett et al. (2004), 'Verification of Object-Oriented Programs with Invariants'

[115]: Kassios (2006), 'Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions'

[157]: Müller (2002), *Modular Specification and Verification of Object-Oriented Programs*

[180]: Reynolds (2002), 'Separation Logic: A Logic for Shared Mutable Data Structures'

Some existing reasoning techniques for smart contracts either assume [174] or aim to prove [3, 92] that re-entrancy cannot lead to behaviors that cannot also occur without re-entrancy (i.e., that contracts are *effectively callback-free* (ECF) [92]). However, these techniques do not apply to the increasing number of contracts that use re-entrant calls as an essential part of their intended workflow (e.g., [151], which we explain Sec. 5.7). In contrast, our methodology applies to *all* contracts, even if they are not ECF: Its central feature is that it allows users to express and prove critical properties of a contract despite the potential for arbitrary re-entrancy, as we explain next.

[174]: Permenev et al. (2020), 'VerX: Safety Verification of Smart Contracts'

[3]: Albert et al. (2020), 'Taming callbacks for smart contract modularity'

[92]: Grossman et al. (2018), 'Online detection of effectively callback free objects with applications to smart contracts'

[92]: Grossman et al. (2018), 'Online detection of effectively callback free objects with applications to smart contracts'

[151]: Minacori (2020), 'EIP-1363: ERC-1363 Payable Token'

5.2.2. Specification and Verification Technique

We propose to use a two-pronged approach to smart contract specification and verification: (1) We introduce a novel specification construct that lets us specify constraints on how a contract directly manipulates its own state. These constraints can be verified without considering external calls. (2) We introduce two additional specification constructs that allow us to reason about the effects of external calls, even when the callee functions are unverified and potentially malicious and may potentially trigger re-entrant calls. Both ingredients exploit a key feature of smart contracts: *All* contract state is private, i.e., cannot be directly modified by functions in other smart contracts. In particular, all updates to a contract's local state are performed by *some* function from the contract's own code.

This feature provides a key distinction between smart contracts and most standard object-oriented languages; however, there are some non-smart contract languages (e.g. actors in Swift [147]) that do offer this feature and can therefore also be verified using our approach. Generally, our technique enables code verification in the presence of re-entrancy and arbitrary unknown and unverified code in any language that offers (1) objects (or contracts) with only object-private state, that is, the state of an object can be modified only by methods of that object, and (2) non-aliasing guarantees for object-private state, that is, there are never any mutable outside references to private object state.

[147]: McCall et al. (2021), *SE-0306: Actors*

In this section, we will informally describe our proposed specification and verification technique using concrete code examples; a formal definition of the resulting proof obligations will be given in Sec. 5.6.

Reasoning about call-free code.

Since the state of a contract can be modified only by its own functions, one can express many important properties as constraints on local code, that is, the call-free code segments between (external) calls. For instance, if *all* such code segments of a contract only ever increase the value of a counter, its value will never get smaller, even when external, potentially re-entrant functions are called. We refer to call-free sequences of statements as *local segments*. In a sense, they represent the atomic operations a contract can perform: Outsiders can observe a contract's state *between* local segments, but never in the middle of a segment. In Fig. 5.3, the local segments of contract A are the ones between the state pairs (0, 1), (2, 3), (4, 5) and (6, 7).

One class of properties that can be enforced by imposing constraints only on local segments is access control, i.e., restricting the right to perform certain operations or modify certain data (indirectly, by calling a function on the contract) to specific callers. Access control is particularly important for smart contracts, since, unlike in standard object-oriented programs, where each object typically manages its own data, smart contracts have to store *all* the data of their clients in their own storage (e.g., the balances in the token contract), making it vital to enforce that clients can only modify parts of that storage that conceptually belong to them. Access control restrictions are therefore a necessary part of the public specification of a contract. For example, for the auction contract from Fig. 5.1 we may

```
def end():
  { msg.sender ≠ self.beneficiary ∧ self.ended = $x_1$ }
  assert not self.lock and not self.ended and msg.sender == self.beneficiary
  self.ended = True
  self.lock = True
  { self.ended = $x_1$ }
  send(self.beneficiary, value=self.highestBid)
  { msg.sender ≠ self.beneficiary ∧ self.ended = $x_2$ }
  self.lock = False
  self.highestBid = 0
  { self.ended = $x_2$ }
```

Figure 5.4.: Proof obligations generated by segment constraint msg.sender $\neq$ *old*(self.beneficiary) $\Longrightarrow$ self.ended = *old*(self.ended) for function end; note that, since msg.sender does not change throughout the execution of the function, this is equivalent to *old*(msg.sender) $\neq$ *old*(self.beneficiary) $\Longrightarrow$ self.ended = *old*(self.ended). The function is divided into two local segments by the call in the middle; each segment is call-free and can therefore be verified using standard techniques. We use the logical variables x_1 and x_2 to represent the old values of the ended field. The same property has to be proved for every local segment in all other functions in the contract.

want to prove that only the beneficiary can end the auction (by setting the ended flag).

To express constraints on local segments, we introduce *segment constraints*—two-state assertions (i.e., assertions that can refer both to the current state and an old state) on the local state of a contract that must hold between the start and end states of *each* local segment in the contract. Segment constraints are specified per contract, not per individual segment. In our auction example, we can express the access restriction for the ended field using the segment constraint msg.sender $\neq$ old(self.beneficiary) $\Longrightarrow$ self.ended = old(self.ended), i.e., if the caller of the current function is not the beneficiary, then the value of the ended field will not be modified by any local segment. Since, by definition, there are no external calls between the start and end states of a local segment, segment constraints are verified without considering external (unverified) code. Figure 5.4 illustrates the proof obligations generated by this example constraint for the end function from our auction contract.

Reasoning about (external) calls

An external call may modify the state of the calling contract A only via (one or several) re-entrant calls. These re-entrant calls perform the modifications of A's state by executing some number of A's functions, which in turn will execute some number of A's local segments (in Fig. 5.3, the segments (2, 3) and (4, 5)). Consequently, the reflexive and transitive closure of constraints describing the effects of A's functions and segments can be used to soundly approximate the effects of an external call. In the following, we introduce two complementary forms of such transitive constraints, which are useful for expressing different kinds of common properties. Both are *auxiliary* specifications in the sense that they typically do not directly express vital correctness properties (unlike e.g., segment constraints prescribing access control properties, which are important in and of themselves), but instead allow us to preserve properties across external calls.

Transitive segment constraints. A vital property the end function of the auction contract must fulfill is that, when it returns, the contract's ended flag is set. Proving this requires showing that any re-entrant calls resulting from the send-statement do not set the flag to false. We can show

Figure 5.5.: Proof outline showing the use of the transitive segment constraint *old*(self.ended) ⇒ self.ended to prove the postcondition self.ended. We use logical variables x_1 and x_2 to represent the values of the ended field at the beginning of each local segment, and x_3 to represent its value before the send statement. The transitive segment constraint must be proved for every local segment of all functions (blue). It can then be assumed to hold between the pre- and post-states of the external call (green), i.e., if self.ended is true before the call, we may assume it is true after the call. This is sufficient to prove the desired postcondition (red).

```
def end():
  { self.ended = x_1 }
  assert not self.lock and not self.ended and msg.sender == self.beneficiary
  self.ended = True
  self.lock = True
  { (x_1 ⇒ self.ended) ∧ x_3 = self.ended }
  send(self.beneficiary, value=self.highestBid)
  { self.ended = x_2 ∧ (x_3 ⇒ self.ended)}
  self.lock = False
  self.highestBid = 0
  { x_2 ⇒ self.ended ∧ self.ended }}
```

that this is the case because no local segment of the contract ever unsets the flag once it has been set, a property which can be expressed as the segment constraint *old*(self.ended) ⇒ self.ended.

The reflexive and transitive closure of all segment constraints of a contract describes the effect of an arbitrary number of local segments. Thus, it is known to hold between the pre-state and the post-state of any external call (i.e., between states 1 and 6 in Fig. 5.3) and can be used to reason about such calls. While, in our example, the single segment constraint we have is already reflexive and transitive, it is generally not possible to compute the reflexive, transitive closure of a set of segment constraints automatically. Therefore, we allow programmers to explicitly specify the *transitive segment constraints* of a contract. These are checked to be reflexive and transitive, and are verified to hold across each local segment of the contract. Like segment constraints, transitive segment constraints are two-state assertions on the local state of a contract, similar to history constraints [137]. Since we enforce that *any* sequence of local segments satisfies the transitive segment constraints of a contract, they may soundly be assumed to hold between the pre- and post-state of each external call. Note that transitive segment constraints do *not* subsume ordinary segment constraints, since typical segment constraints used for access control (e.g., the restriction on who can end an auction, shown before) are not transitive.

[137]: Liskov et al. (1993), 'Specifications and Their Use in Defining Subtypes'

Transitive segment constraints are useful to express constancy properties, such as the fact that the auction's beneficiary never changes, or monotonicity properties like that of the ended field discussed before. The latter can be expressed using the transitive segment constraint old(self.ended) ⇒ self.ended; Figure 5.5 illustrates how this constraint can be used to prove the desired postcondition for function end (even without the lock, which we will discuss below).

Transitive segment constraints subsume single-state *contract invariants*, which are often useful to specify consistency conditions on contract states, which must hold whenever the contract relinquishes control to other contracts (and, thus, its state becomes observable to the environment). The verification of transitive segment constraints implies that single-state contract invariants hold at the end of each local segment, which includes the state before any call as well as the post-state of each function. Consequently, each function may soundly assume such contract invariants to hold in its pre-state, as well as after the return of each external call, analogously to class or object invariants in object-oriented programs [63, 122]. As an example, for the auction, an important invariant is that its funds suffice to pay all its obligations, which can be written as the transitive segment constraint $\texttt{self.balance} \geq \mathit{sum}(\texttt{self.pendingReturns}) + \texttt{self.highestBid}$.

[63]: Drossopoulou et al. (2008), 'A Unified Framework for Verification Techniques for Object Invariants'
[122]: Leavens et al. (2008), *JML reference manual*

Function constraints. It is common that each individual function of a contract *as a whole* satisfies a two-state property, even if some of its local segments do not. Such situations occur for instance if *some* sequences of local segments violate the property, but no function in the contract ever executes such a sequence. The re-entrancy lock in the auction contract is an example: The field self.lock is set to true by withdraw and end before their calls to send, and reset to false afterwards. Since each function of the contract reverts if the locks is set, this pattern ensures that each function of the auction contract leaves the contract state completely unchanged if the lock is set in its pre-state.

However, this property cannot be verified as a transitive segment constraint: Some local segments reset the lock, such that any subsequent state change violates the property. That is, the property does not hold for arbitrary sequences of local segments, but it does hold between the pre-state and the post-state of each contract function. Note that any external call can modify the contract state only by executing these contract functions (via re-entrant calls) from start to finish.

To exploit this fact, we introduce *function constraints*: two-state assertions on the local state of a contract that must hold between the pre- and post-state of every function in the contract. In Fig. 5.3, this means they have to hold between states 2 and 5 as well as states 0 and 7. Like transitive segment constraints, function constraints are specified per contract; they must be satisfied by *all* of its functions (reflecting that we do not know statically which re-entrant calls are triggered by an external call). Since external calls may trigger the execution of an arbitrary number of contract functions, we require function constraints to be reflexive and transitive. For the lock example, we can express the desired property as the function constraint $\texttt{old}(\texttt{self.lock}) \Rightarrow \texttt{self} = \texttt{old}(\texttt{self})$, meaning that the entire state of the self object (including its lock field) stays will be unchanged at the end of each function, assuming that the lock field was set initially.

Note that function constraints do *not* subsume transitive segment constraints. For instance, in the special case of single-state assertions, transitive segment constraints (that is, the contract invariants discussed above) are known to hold before each call and may, thus, be assumed in the pre-state of each function, whereas function constraints may not, since they do not have to be established before calls. Neither do transitive segment constraints subsume function constraints, as we illustrated with the lock example above.

With these two specification constructs, we can modularly verify properties in the presence of calls to unverified and potentially malicious contracts with arbitrary re-entrancy. In Sec. 5.4, we will complement these constraints with effect specifications on a contract's resources to obtain even stronger guarantees.

5.3. Inter-Contract Invariants

```
interface Token:
  balances: map(address, uint256)

  def transfer(from: address, to: address, amount: uint256):
    pass

contract Auction:
  token: Token

  def withdraw():
    assert not self.lock
    toSend = self.pendingReturns[msg.sender]
    self.pendingReturns[msg.sender] = 0
    self.lock = True
    self.token.transfer(self, msg.sender, toSend)
    self.lock = False

  def distributeExcess():
    excess: int128 = self.token.balances[self] − sum(self.pendingReturns)
    excess −= self.highestBid
    assert excess != 0
    perBidder: int128 = excess / size(self.pendingReturns)
    for bidder in keys(self.pendingReturns):
      self.pendingReturns[bidder] += perBidder
```

Figure 5.6.: Minimal interface of the token contract in Fig. 5.2, and part of an adapted auction contract that deals in tokens and additionally has a function that distributes excess tokens among previous bidders.

Smart contract applications are frequently implemented via multiple contracts which call one another. As in many programming languages, the *interfaces* of the Vyper and Solidity languages are designed to facilitate such collaborations. Interfaces declare that a contract offers *at least* some set of functions and fields[4], but do not give any information about their implementation, or preclude the existence of additional functions in the contract. Therefore, they decouple (in the software engineering sense) client contracts from the concrete implementations of the contracts they build on.

For example, an auction contract similar to our example from Figure 5.1 could, instead of Ether, deal in tokens conforming to e.g., the ERC20 standard interface [220]. Fig. 5.6 shows a minimal interface of the token contract from Fig. 5.2 as well as (part of) a modified version of the auction contract, where calls to send are replaced by calls to the token contract's transfer function (we will discuss the added function distributeExcess later).

However, our techniques for equipping contracts with invariants and proof obligations to maintain them no-longer suffice for collaborating contracts, since such collaborations naturally give rise to invariants that depend on the state of *other* contracts. For example, the modified auction still needs an invariant that it has sufficient funds (now tokens) to pay its obligations to all participants, which can be expressed in terms of the states of *both* the auction and token contract by: self.highestBid + *sum*(self.pendingReturns) ≥ self.token.balances[self]. In this section, we extend the technique presented in Sec. 5.2 to such *inter-contract invariants*[5].

An inter-contract invariant has a single *primary* contract (the contract depending directly on the property); any other contracts whose state is mentioned are its *secondary* contracts. In the example, the modified auction contract is the primary contract, since the auction's funds must be sufficient for the auction to function correctly, whereas the token contract can have many (non-auction) clients with different functionality and is

[220]: Vogelsteller et al. (2015), 'EIP-20: ERC-20 Token Standard'

[4] Interfaces actually contain constant functions that guarantee not to modify any state instead of fields, but we model them as fields here to simplify the presentation.

[5] We focus on single-state invariants here for simplicity only: our technical solution also supports two-state assertions.

```
contract BadToken implements Token:

  def steal(from: address, amount: uint256):
    assert self.balances[from] >= amount
    self.balances[msg.sender] += amount
    self.balances[from] -= amount

  def transfer(from: address, to: address, amount: uint256):
    assert self.balances[from] >= amount and msg.sender == from
    newAmount: uint256 = self.balances[from] - amount
    self.balances[to] += amount
    self.balances[from] = 0
    thirdparty.notify()
    assert self.balances[from] == 0
    self.balances[from] = newAmount

contract ThirdParty:
  auction: Auction

  def notify():
    auction.distributeExcess()
```

Figure 5.7.: Possible implementation of the Token interface. The implementing contract may offer additional functions, e.g., in this case, one that allows anyone to steal tokens from any existing account.

not responsible for their correctness. This asymmetry is reflected in the above invariant, where self is the auction contract.

Ensuring non-trivial inter-contract invariants requires that both the primary and all secondary contracts are verified (and that the primary contract trusts the secondary contracts to adhere to their specifications), since the state of unverified, unknown and adversarial contracts may change arbitrarily, which precludes the verification of invariants that depend on it. However, all contracts other than the primary and secondary ones are still regarded as untrusted, do not have to be verified, and can in fact be malicious, and as before, verified contracts may still call functions of unverified, untrusted ones. Additionally, we do *not* depend on having access to the implementations of the secondary contracts. These may in particular be hidden behind interfaces.

5.3.1. Challenges

Modular verification of inter-contract invariants in our setting poses two main challenges:

Challenge 1: Missing encapsulation. The first challenge is that the state that an inter-contract invariant depends on is not fully-encapsulated in the way we have exploited so far. While, of course, the rules of the programming language have not changed and each contract's state can only be modified by its own code, now, the invariants we aim to prove involve the states of multiple contracts, but the primary contract directly controls only its own state. In other words, it is now no longer the case that the invariant can only be broken by code of the primary contract; instead, it can be also be broken by the code of a secondary contract, which may not be known.

To illustrate this challenge, consider a scenario in which the token contract has a function steal that lets an arbitrary contract steal another contract's funds, as shown in Figure 5.7: If this function existed, a third party could call it to steal the tokens of the auction contract; if the auction contract also has any pending returns, this would break our inter-contract invariant. Note that there is nothing the primary contract can do to prevent this; in fact, steal may be called in a transaction that does not involve the primary contract at all. Additionally, one cannot conclude from the token contract's interface alone whether or not it has such a function (or any other function that allows one contract to decrease another contract's funds).

Challenge 2: Temporarily-broken invariants. Second, secondary contracts may *temporarily* break inter-contract invariants when called by the primary contract in a way that makes the inconsistent state visible to other contracts. Note that it is normal and unavoidable that invariants are temporarily broken while a contract is executing its own code; however, states in which this is the case must *never* be visible to outside contracts.

The challenge is illustrated in function transfer in Figure 5.7, which, when used by the auction contract can lead to the invariant being broken. This function performs a token transfer from one contract to another, as it should, but (perhaps as an attempt to avoid a TheDAO-like re-entrancy vulnerability) it temporarily sets the balance of the token sender to zero and performs a call to the outside, before restoring the balances to the desired end state. This can lead to problematic behavior: Assume that the auction contract has non-zero pending returns for two contracts, A and B, and that contract B is the ThirdParty contract shown in Figure 5.7. If contract A calls withdraw, the auction contract will call transfer, which will set its token balance to zero. Now the token contract calls function notify of contract B. Note that, in this state, the inter-contract invariant is broken: The auction contract still has pending returns for B, but its current balance is zero. When B in turn calls the function distributeExcess in the auction contract this state, this function does not work as designed: The purpose of the function is to distribute any excess tokens the auction contract may own among previous bidders (which may be part of some intended functionality where third parties are rewarded for taking part in the auction, or simply a failsafe in case someone accidentally transfers tokens to the auction contract). It calculates the excess by subtracting the sum of the pending returns and the current highest bid from the auction contract's token balance, assuming that the result will be non-negative, which *should* be guaranteed by the invariant. As a result, since we assume the invariants (transitive segment constraints) at the beginning of each function, we could prove a postcondition here that states that pending returns can only be increased by this function. Now, however, the result can actually be negative, and as a result, the pending returns of all previous bidders will be decreased, breaking the postcondition.

Note that, again, it is not possible to see from the interface that this problem exists: If function transfer has a postcondition that describes its behavior, it will state that (by the time the function returns) the transfer has been executed as desired; nor is it possible to see from an interface whether the function performs any calls to the outside. Thus, while one may regard the behavior of transfer in our example as artificial or clumsy, our point is that such behavior *cannot be ruled out* knowing only the token contract's interface. Therefore, a sound verification technique whose goal is to ensure that states in which the inter-contract invariant is never broken must account for cases like this and prevent them. Also note that, unlike the previous problem, this problem is unrelated to encapsulation: Now, it is not third parties that can modify state in unintended ways, but it is the primary contract itself (which *must* be able to modify the state in the token contract that conceptually belongs to it) whose call has unintended consequences.

We must therefore adjust our verification technique to ensure that this invariant *cannot* be proved for this contract, since it does not hold in practice. Subsequently, we will show that we can prove an *alternative* invariant that still serves our purpose: we exploit that whenever both contracts' state may be out of sync, the lock is set and the auction contract cannot be called. So, we require that the desired consistency criterion (that the auction has sufficient funds to pay its obligations) is true whenever the lock field is not set[6], resulting in the invariant $\neg\textbf{self}.\text{lock} \Rightarrow \textbf{self}.\text{highestBid} + sum(\textbf{self}.\text{pendingReturns}) \geq \textbf{self}.\text{token.balances}[\textbf{self}]$. As we will show below, this invariant actually holds, and makes explicit that distributeExcess can rely on its funds being greater or equal the contract's obligations *only* when the lock is not set. As a result, when using the alternative invariant, it will be impossible to prove a postcondition for this function stating that it only increases pending returns, unless the function is fixed by adding **assert not self**.lock at the beginning.

[6] This pattern is often used in OO-verification when proving invariants between multiple objects [129].

5.3.2. Solution

In order to enable modular verification of inter-contract invariants, we build on our existing approach for proving transitive segment constraints, introduce one additional specification construct, and add proof obligations that address both of the challenges mentioned above (as before, we will informally discuss these proof obligations in this section, and subsequently formalize them in Sec. 5.6). More concretely, transitive segment constraints are now (unlike all other specification constructs we introduced) allowed to express inter-contract invariants, i.e., they may now refer to the state of other contracts that are reachable from the primary contract. All existing proof obligations for transitive segment constraints remain, that is, we check that they are reflexive and transitive, prove that they are established by the primary contract's constructor, and verify that each local segment of the primary contract satisfies them. Additionally, interfaces may declare both function postconditions and transitive segment constraints, which all contracts implementing the interface must adhere to.

We now address Challenge 1 by extending interfaces with specifications that provide the missing guarantees: we allow annotating interfaces with novel *privacy constraints*, which express which part of the contract state conceptually belongs to specific clients and, thus, cannot be freely manipulated by other clients. Essentially, a privacy constraint extends the encapsulation guarantees that already exist for the state of the primary contract to (parts of) the state of a secondary contract. Privacy constraints are segment constraints of the form $\forall a.\textbf{msg}.\text{sender} \neq a \Rightarrow P$, where P is reflexive and transitive. That is, they have exactly the form of typical segment constraints that express access control properties, as shown in the previous section (recall that the fact that only the auction's beneficiary can end the auction was expressed by the segment constraint $\textbf{msg}.\text{sender} \neq \text{old}(\textbf{self}.\text{beneficiary}) \Rightarrow \textbf{self}.\text{ended} = \text{old}(\textbf{self}.\text{ended})$), with one crucial difference: Instead of stating what privileges are restricted to specific callers, privacy constraints universally quantify over the caller's address, and thus express guarantees that hold for *every* arbitrary caller. This is by design, as their purpose is to express which parts of a secondary's contract's state is controlled *only* by the primary contract, which, from the secondary contract's perspective, may be an arbitrary client it has no special relationship with. The privacy constraints specified in an interface must be satisfied by *all* functions of a contract implementing the interface, even those not mentioned in the interface.

On the token interface from Fig. 5.6, the privacy constraint $\forall a.\,\textbf{msg}.\text{sender} \neq a \Rightarrow \textbf{self}.\text{balances[a]} \geq \text{old}(\textbf{self}.\text{balances[a]})$ expresses that for any client, if it does not directly call the token contract, its balance cannot decrease; in other words, a caller may increase the balance of any contract, but decrease only its own. Since the steal function from Figure 5.7 violates this property, BadToken is now no longer a valid implementation of the Token interface. In other words, the privacy constraint constitutes a promise that the secondary contract does not contain *any* function that will allow third parties to decrease the auction contract's balance. This is exactly the information needed to prove that calls on the token contract by third parties cannot violate the inter-contract invariant stated at the beginning of this section.

In general, assuming that all secondary contracts are annotated with privacy constraints, we prove that those privacy constraints re-encapsulate the state our invariant depends on as follows: We enforce that each inter-contract invariant is *stable* under state changes allowed by the privacy constraints of all secondary contracts, i.e., that the privacy constraints forbid all changes that could break the invariant. We formally define the notion of assertion stability in Sec. 5.6 and illustrate it here with our example: Assume that a function of the secondary token contract is called by a party other than the primary contract. For any local segment of the token contract, the token's privacy constraint shown above guarantees that $\textbf{self}.\text{token.balances}[\textbf{self}] \geq \text{old}(\textbf{self}.\text{token.balances}[\textbf{self}])$. If, in the old state, the inter-contract invariant $\textbf{self}.\text{highestBid} + sum(\textbf{self}.\text{pendingReturns}) \geq \textbf{self}.\text{token.balances}[\textbf{self}]$ held, then the privacy constraint (along with the knowledge that the local segments of the secondary contract cannot directly change the state of any other contracts) implies that the inter-contract invariant also holds in the new state.

Challenge 2 is not addressed by the introduction of privacy constraints; since the caller in this scenario is the primary contract itself, privacy constraints offer no guarantees about the potential behavior of the secondary contract. To address the second challenge, we require that, in every state where the primary contract directly calls a secondary contract, *any* changes that any local segment of the secondary contract could possibly make cannot break the invariant. That is, we require that we can prove a *stronger version* of the invariant at the call site, such that this stronger invariant is stable under the conjunction of all function constraints of the primary contract, all transitive segment constraints of all secondary contracts, plus the privacy constraints of all secondary contracts *except* the called one (see Sec. 5.6 for the precise stability condition). This stability criterion is the same as before, except that we are not using the privacy constraint of the called secondary contract (since privacy constraints only give guarantees to contracts that are currently *not* calling them, but the guarantees we get from all other secondary contracts are unaffected by the primary contract's call).

For our original invariant, this stability criterion does not hold; however, we can show that it does hold for our adapted invariant from above (stating that the auction has sufficient funds to pay its debts *if* the lock field is *not* set) as follows: We show that whenever the primary contract calls the secondary contract, the assertion self.lock holds, which is a stronger version of our adapted invariant. This assertion is independent of the token contract and so cannot be broken by any changes to its state. In other words, when the lock flag is set, the adapted invariant trivially holds independently of the secondary token contract's state, and therefore cannot possibly be broken by any changes the secondary contract might make to its own state. We can also show that this stronger assertion is preserved by calls to the primary contract by using a suitable function constraint old(self.lock) ⇒ self.lock. This last point is also sufficient to fulfill the stability criterion we just described.

The proof obligations we have outlined ensure that secondary contracts cannot break inter-contract invariants when called by the primary contract (Challenge 2) or anyone else (Challenge 1). Along with the proof obligations that ensure that the primary contract establishes and maintains the invariant, which we described in the previous section, this is sufficient to guarantee that the inter-contract invariant will hold at the end of every local segment of any contract.

In summary, the added proof obligations generalize our previously-introduced specification constructs to verify invariants of collaborating contracts. This verification is fully modular, based on the implementation of the primary contract and specified interfaces for all secondary contracts. It is sound even when these contracts interact with unverified code, and in the presence of arbitrary re-entrancy.

5.4. Resource-Based Specifications

The vast majority of smart contracts in some way model resources and resource transfers, such as the token and auction contracts we have seen before. Resources have a number of basic properties that are important for the correctness of every contract that works with them: they *cannot be duplicated*, they *have an owner*, and they *cannot be taken away from that owner without their consent*. Successful smart contract exploits are often the result of one or more of these key integrity properties being violated. Explicitly specifying these properties for every smart contract that uses some sort of resource is possible, but laborious and error-prone. Instead, we propose a dedicated specification and verification technique that has basic resource properties built-in and that offers high-level specification constructs to declare resources and to describe resource transactions. The potential of resource-based reasoning for smart contracts has been recognized before; for instance, Move [35] has native support for resources in the blockchain and language (but does not have built-in guarantees of all the basic properties mentioned above). Compared to specifications that express resource properties via changes of the contract state, our resource specification system has three main advantages (note that Move, due to it

[35]: Blackshear et al. (2019), *Move: A language with programmable resources*

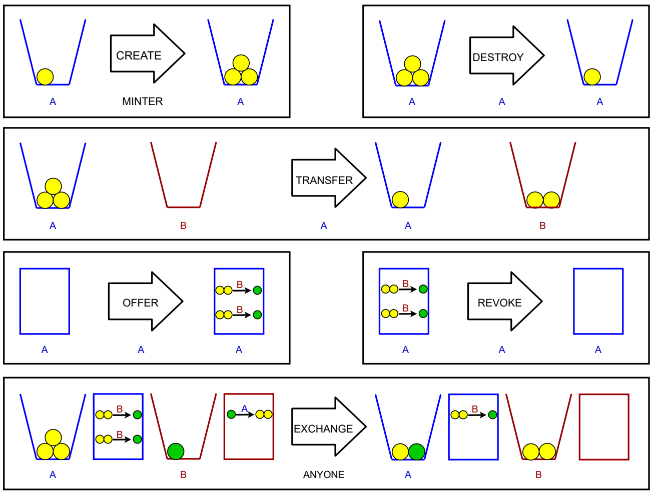


Figure 5.8.: Operations on resources and offers. Buckets represent resources owned by addresses A and B; the rectangles contain the offers they have made. Big arrows show the name of the resource operation; the names under the arrows show who can perform the operation (create-operations may be performed by anyone who has a special resource representing the right to mint new resources). For example, only A can *transfer* two of its three yellow resources to B, or it can *offer* B to exchange two of its yellow resources against a green one.

having a different resource model, does not have these three advantages built-in, see Sec. 5.8):

1. Guaranteed integrity: Basic integrity properties of resources, such as the fact that they cannot be duplicated and cannot be taken away from their current owner without their consent, are baked into the system. Our verification approach ensures that these properties hold by default, without developers having to specify them, so that there is no danger that important properties are missing in the specifications, and there is no need to write them down for every contract.
2. Higher-level reasoning: Developers think about resources as an abstract concept; for instance, they think of a token as a kind of currency, not some contract whose state contains a map. Resource-based specifications let developers describe their contracts' states and interactions on this abstraction level, leading to simpler and more intuitive specifications.
3. Client documentation: Writing postcondition-based specifications for smart contract functions is often difficult because of potentially re-entrant calls with unbounded effects. Our resource system enables users to prove novel effect-based function specifications that give a caller an upper bound on the negative consequences it may suffer from calling a function (e.g., losing some Ether) and a lower bound on the positive consequences (e.g., receiving some tokens).

In this section, we describe the basic attributes of our resources, the operations that can be performed on them, and how we connect the contract's actual state to the resource state. We show how effect-based function specifications based on resources give callers extra information. Finally, we describe advanced concepts such as *derived* resources, representing titles to other resources. As in the last two sections, our explanations here will be informal, and a formal description of all proof obligations will follow in Sec. 5.6.

5.4.1. Resource Model

In our system, a resource can represent anything that cannot be duplicated, has an *owner*, and has some non-negative value. Resources are owned by addresses on the blockchain. Ownership implies control of the resource, i.e., only the owner of a resource can transfer or destroy it. Receiving additional resources does not require consent. In this regard, our resources are similar to Ethereum's built-in Ether resource (which is treated as a built-in resource in our system).

Fig. 5.8 shows the operations that can be performed on resources, and who is allowed to perform them. Resources can be *created* by privileged parties who have the right to do so (usually called *minters*). They can be *transferred* to other addresses or *destroyed* by their owners (and by noone else). In addition, addresses can make *offers* to exchange some resources against others; when an offer from one address exists and a second party makes a matching counter-offer, the *exchange* can be performed at an arbitrary point in time, without further agreement by the involved parties (consuming the offers). For maximum flexibility, an address need not own the resources it offers at the point when it makes the offer, but an exchange requires that both addresses actually hold the offered resources. Addresses can *revoke* previous offers they have made. The resulting set of operations is simple but sufficient to model the behavior of a wide range of real smart contracts.

5.4.2. Resource State

Every smart contract may declare one or more resources that they implement (e.g., token contracts would declare a token resource). For each resource, all addresses implicitly have a balance (as for the built-in Ether). Similarly, each address has a set of existing offers on the resources it declares. These balances and offer sets are *ghost state*: state that exists only for verification purposes, but is not present at execution time. Our specifications can refer to this state, e.g., a postcondition can refer to the caller's balance for resource R as $\mathsf{balances}_R[\mathbf{msg}.\mathsf{sender}]$. Note that specifications about resource state can be arbitrarily combined with the other specification constructs we have introduced; for example, one could write a segment constraint stating that a contract may perform some operation only if it owns some minimum amount of a resource.

The resource ghost state can be changed only by executing *ghost statements*, written in the verified contract, that each perform one of the resource operations mentioned above. As an example, the ghost statement $\mathsf{transfer}_R(f, t, a)$ transfers a amount of resource R from f to t. This ghost statement requires that f has sufficient amounts of R, and that f is the contract invoking the ghost statement, i.e., that f has called the function that contains it (modulo delegation, which we discuss later). These conditions are checked by the verifier, and they enforce the basic properties of the resource system; the latter check in particular enforces ownership constraints. Similar ghost statements exist for the other basic resource operations:

- Resource can be created by executing the ghost statement $\mathsf{create}_R(e_c, e_t, e_a)$, which means that e_c creates e_a amount of resource R and allocates it to e_t, and which has the effect $\mathsf{create}_R(e_t, e_a)$. Creating a resource requires that e_c is the current **msg**.sender, and additionally, that e_c owns the *right* to do so, which we represent by a special resource $\mathsf{creator}(R)$.

- When executing the statement $\mathsf{destroy}_R(e_f, e_a)$, e_f destroys e_a amount of its reserves of resource R, which requires that e_f actually has that amount of resource, and, as before, that e_f is the **msg**.sender.
- The statement $\mathsf{offer}_{R_1 \leftrightarrow R_2}(e_f, e_t, e_{a_1}, e_{a_2}, e_n)$ creates e_n offers from e_f to contract e_t to exchange e_{a_1} amount of R_1 against e_{a_2} amount of R_2 if e_f is the **msg**.sender.
- The statement $\mathsf{revoke}_{R_1 \leftrightarrow R_2}(e_f, e_t, e_{a_1}, e_{a_2}, e_n)$ revokes e_n offers from e_f to contract e_t to exchange e_{a_1} amount of R_1 against e_{a_2} amount of R_2 if e_f is the **msg**.sender, and requires that those offers exist before its execution.
- The statement $\mathsf{exchange}_{R_1 \leftrightarrow R_2}(e_f, e_t, e_{a_1}, e_{a_2})$ performs an exchange between e_{a_1} amount of R_1 from e_f and e_{a_2} amount of R_2 from e_t. Executing it requires that both e_f and e_t have made an offer to perform such an exchange, and consumes the offer. There is one exception to this: if either e_{a_1} or e_{a_2} are zero, i.e., the exchange simply gives a resource to one party without requiring anything in return, no offer is required from the party receiving the resource. Unlike any other ghost statement, there are no requirements as to who can execute this statement (i.e., who the **msg**.sender is), since its effect is one that all affected parties have already agreed to previously.

The resource ghost state has the same encapsulation as ordinary contract state, that is, the ghost statements in a contract can modify only the state of the resources declared in that contract.

5.4.3. Connecting Resource State and Contract State

In order to be useful for verification, the resource ghost state must be connected to the contract's actual state. Our system achieves this by letting developers write invariants (i.e., transitive segment constraints) that relate the resource ghost state (i.e., balances and existing offers) to the contract state. When verifying a contract, we then enforce that these coupling invariants, like all transitive segment constraints, hold at the end of every local segment. For the token contract, the invariant would be $\mathsf{balances}_{token} =$ **self**.balances, meaning that the balances of the resource named "token" are recorded in the contract's balances field.

This check essentially forces changes on the resource state and on the contract state to happen in lockstep: If a change happens on the resource state with no equivalent change on the contract state (or vice versa), the invariant cannot be verified. As a result, the properties our system guarantees for resources (like ownership) carry over to the actual contract state. For instance, our system prevents the verification of a function steal that allows arbitrary callers to steal some client's funds: the modification to the contract state must be mirrored in a corresponding change of the resource state (by the coupling invariant). The ghost statement that makes this change checks that the transferred funds belong to the function's caller, which would fail here.

```
def transfer(from: address, to: address, amount: uint256):
 { balances_token = self.balances }
 assert self.balances[from] >= amount and msg.sender == from
 self.balances[to] += amount
 self.balances[from] -= amount
 transfer_token(from, to, amount)
 { balances_token = self.balances }
 to.notify(from, self, amount)
```

Figure 5.9.: Example of contract state and resource state moving in lockstep. In order to re-establish the coupling invariant (blue) at the end of the first local segment of the token contract's transfer function after modifying the contract state, one must execute a transfer statement (red) that modifies the resource state accordingly. Verification ensures that all conditions imposed by the resource model are fulfilled when transfer is executed.

As an example, in the transfer function of the token contract, we would have to insert the ghost statement transfer$_{token}$(**from**, to, amount) before the call to notify in order to re-establish the invariant, as shown in Figure 5.9. Verifying the function requires proving that **from** is the **msg**.sender and that **from** owns at least amount tokens, both of which we can prove because of the assertion at the beginning of the function.

Similarly, in function mint of the token contract, which increases the number of tokens in the contract state, we would have to insert the statement create$_R$(**self**.minter, to, amount) to preserve the relation between balances$_{token}$ and the contract state. Verifying mint then requires showing that the caller is **self**.minter, which we know because of the assertion in the first line, and that **self**.minter owns a creator resource for token. To show the latter, we must create such a resource in the contract's constructor, and record the fact that the minter owns it in an additional invariant.

5.4.4. Client Specifications

The system described so far guarantees that others cannot take away the resources owned by a contract. However, the contract itself may perform operations that lead to a loss of resources, e.g., by transferring or destroying them. Our rules for resource statements ensure that any such operation is initiated by the owner calling a function on the contract that declares the resource (e.g., a function on the token contract that contains a transfer ghost statement). Therefore, it is vital that functions provide callers with specifications describing how they affect the caller's resources.

We address this problem by introducing *effects-clauses* on contract functions, which specify ghost statements that will be executed when the function is called (assuming it does not revert). Each function has a multiset of effects, and each effect corresponds directly to one of the ghost statements introduced above, meaning that there are effects for creating, transferring, and exchanging resources, etc. Effects-clauses are unordered and do not give any information regarding *when* during the call the effects occur. As an example, if a function's effects-clause contains transfer$_R$(**msg**.sender, t, 4) and transfer$_R$(**msg**.sender, t, 6), this means that after successful execution, it will have transferred 10 R (in two separate steps of 4 and 6) from the caller to address t at some point during the call.

In contrast to traditional effects-systems, our effects-clauses are *not* required to be transitive: the ghost operations performed directly by the called contract's function must be included, but those caused by further external calls made by this contract need not be tracked. This non-standard design is motivated by the presence of calls to unknown code; ultimately, there will be cases in which we could not know the effects of arbitrary external code. The rules for checking effects clauses are simple: (a) the effects-clause of a function *must* contain the effects of all ghost statements directly in its body, and (b) it *may* declare any additional known effects resulting from its own calls to other functions.

This might sound problematic: a caller of a function is only able to see the effects clauses (and postcondition) to judge whether they *consent* to these effects happening, but if the effects do not track all transitive calls, one might expect that the caller could be tricked into allowing, e.g., more of their resource to be transferred than they realize. Perhaps surprisingly, due to the resource ownership principles built into our methodology, our non-transitive effects actually remain powerful and useful: they ultimately describe worst-case information about what could happen to a caller's resources by calling the function in question *except possibly for additional calls that go via the same original caller.* In other words, the caller is explicitly consenting to at most these effects happening, unless they subsequently consent to additional effects by making a further call.

To see why this is the case, consider a function that may have a negative effect on a calling contract's resources. Since all ghost operations that have a negative effect impose a proof obligation that the resources are owned by `msg.sender`, the negative effect *must* occur either in the initially-called function, or, after a sequence of additional calls, in some function that was again called *by the original caller*. In the first case, according to check (a), this effect will be included in the initially-called function's effect clause: the caller was aware of the effect and allowed it to happen by making the call. In the second case, for the same reason, this effect must be declared on the subsequently-called function called by the same caller, who consents to the additional effects by making this subsequent call. Note that it *is* possible that a call will cause effects that are positive or neutral for the caller (e.g., an unknown contract giving them tokens) which the called function did not declare; however, since those are not negative for the caller, not knowing about them does not impact the caller's ability to consent to the call.

As a result, our effects-clauses enable each contract to know which negative effects a call may have on its resources, such that it can refrain from making calls with undesired effects. This solution gives strong guarantees in the presence of arbitrary re-entrancy, when it is impossible to give the called function a precise postcondition.

Figure 5.10 illustrates the use of resource specifications and effects- clauses in a realistic setting, namely on the token contract from before, more precisely, an extended version of the token contract that is close to a real implementation of ERC20 in Vyper. Effects-clauses are introduced with the keyword `performs`.

5.4.5. Derived Resources

As a final ingredient, our system contains one additional concept to model the difference between physically having a resource and conceptually being its rightful owner. As an example, consider the auction contract again: Whenever a bidder sends some wei to it, that wei now physically belongs to the auction contract, which could in principle do with it whatever it wants. However, conceptually, as long as the auction is running, the bidder is still the owner of the wei it sent, and it rightfully expects to either be able to get it back later (if someone else makes a higher bid) or to exchange it for the auctioned good when the auction ends. That is, after a bid and before the end of the auction, the physical owner of that wei (the auction contract) is different from the conceptual owner (the bidder). This is a relatively common notion that occurs whenever some contract (temporarily) manages another contract's resources, and it obviously comes with certain expectations (e.g., the auction contract should not be able to give the wei it has to anyone but its rightful owners).

Our system has support for this scenario in the form of *derived resources*, representing conceptual ownership of a resource physically owned by someone else; essentially, a kind of *title*. In our example, the auction contract could declare a resource wei_in_auction derived from the built-in wei resource, as shown in Figure 5.11. When a bidder sends wei to the auction contract by calling function bid, it transfers its wei to it, but gets the same amount of wei_in_auction in return, signifying that it is owed that amount of wei from the auction contract. If another higher bid comes in and the bidder gets its wei back by calling withdraw, its titles are destroyed again. In contrast, the winner of the auction exchanges its titles against the auctioned good, so that their bid is now owed to the beneficiary of the auction. At any given point, the amount of titles address c has in the auction contract is $\textbf{self}.\text{pendingReturns}[c] + (\textbf{self}.\text{highestBidder} = c ? \textbf{self}.\text{highestBid} : 0)$, meaning that this is also the amount of wei_in_auction contract c owns.

Derived resource creation and destruction. The existence of an instance of a derived resource is always bound to an instance of the resource it is derived from. That is, if a contract declares resource D derived from another resource R, then an instance of D comes into existence for every instance of R it receives (via a transfer operation or an exchange), and is automatically allocated to the sender of the R; there is no way to create an instance of D without receiving an instance of R. Similarly, whenever the contract sends some amount of R to another contract, this destroys the same number of D instances said other contract currently owns. This mechanism ensures that the contract always owns enough of the original resource to "pay back" its title loans. The reader may recall that this fact was an invariant of the auction contract that we explicitly mentioned in Sec. 5.2; now, with derived resources, this invariant is checked automatically and does not have to be specified explicitly.

Derived resource transfers. In order to ensure that contracts do not give away resources that (according to an existing title) belong to someone else, our system enforces that the contract may transfer R to another contract *only* if that other contract already owns a sufficient amount of D, i.e., the original contract already owes the second contract at least the amount to be transferred. As an example, when the auction contract sends some amount of wei to a previous bidder of the auction in line 18, this is allowed only if the bidder currently owns an equal amount of wei_in_auction, and if the beneficiary has offered to exchange its wei_in_auction back to ordinary wei. If this is the case, then, the moment the send executes, that amount of the beneficiary's wei_in_auction is automatically destroyed, and the offer to exchange it is consumed.

Apart from the aforementioned restrictions, derived resources behave just like other resources. In particular, they can be traded like other resources (e.g., someone could pay for some good in wei_in_auction, meaning that they give the right to get wei back from the auction contract to someone else). This is relevant for some DeFi contracts that give out tokens that represent ownership of some other goods (i.e., derived resources), but are traded as tokens on their own.

Proof technique. Our proof technique enforces the properties listed above by automatically creating and/or destroying instances of the derived resource whenever a contract calls an external function that declares (in its effects-clause) that it performs a transfer or exchange of the underlying resource to or from the calling contract. Sending or receiving wei is a special case but is treated analogously, i.e., when sending wei, this is handled as if the called function declared that it transfers wei away from the calling contract. To avoid that a contract loses resources that conceptually belong to others without its knowledge (which would mean that it cannot perform the aforementioned checks), our system enforces that the contract declaring D cannot make offers to give away R, since such offers could result in the contract losing R-instances (when the exchange happens) at an arbitrary point in the future.

Figure 5.12 shows the entire (slightly simplified) auction contract with derived resource specifications. Note that in our tool, a derived resource for wei is declared automatically (meaning that by default, the assumption is that wei sent to a contract should still conceptually belong to the sender, i.e., they get a title for it); here, we have explicitly declared it with the name wei_in_auction for illustration purposes.

5.4.6. Further Extensions

Our system contains a few more features that we have not described so far. The most important is the notion of *delegation*: It is sometimes useful or necessary for collaborating contracts to be able to act in each others' names when interacting with other contracts (in fact, this functionality is built into the core of many existing token standards, like ERC 721 [71]). To enable this, we allow a contract A to decide to *trust* another contract B w.r.t. outside contract C (by calling a function on c that executes the ghost statement trust(B, *true*), which sets the current caller's trust to the contract at address B to true), meaning that when B interacts with C (and only then), it can perform actions that normally only A would be able to perform (e.g., transfer A's resources to someone else). As a result, all restrictions on who may execute certain ghost statement that we have discussed so far are implemented modulo trust. Since trusting someone weakens the guarantees one has for one's own resources, users must use this feature with caution; however, as with all other potentially negative effects, functions that establish new trust relations must always state that they do so in their effects-clause.

[71]: Entriken et al. (2018), 'EIP-721: ERC-721 Non-Fungible Token Standard'

Our core methodology is easily extended in several further ways; our implementation e.g., has support for resources with identifiers (resources whose instances can be distinguished from one another) useful for modeling non-fungible tokens (NFTs) [71]. Other generalizations are possible, e.g., for some contracts it may be useful to have derived resources that represent ownership not of a single resource of a different type, but of different amounts of other resources. This feature (like many others) does not have to be built into the system; it can be emulated by using the existing resource model in combination with additional invariants, segment constraints etc. that represent the additional rights and constraints that would result from such resources. As we show in Sec. 5.7, the set of features we have described gives users a sufficiently expressive model to be able to verify real contracts, while being simple enough for users to reason about.

[71]: Entriken et al. (2018), 'EIP-721: ERC-721 Non-Fungible Token Standard'

```
 1  minter: address
 2  balances: map(address, int256)
 3  allowances: map(address, map(address, int256))
 4
 5  resource: token()
 6
 7  transitive segment constraint: self.minter == old(self.minter)
 8  transitive segment constraint: self.total_supply == sum(self.balances)
 9
10  transitive segment constraint: balances[token] == self.balances
11  transitive segment constraint: forall(o: address, s: address,
12  self.allowances[o][s] == offered[token <-> token](1, 0, o, s))
13
14
15  performs: create[token](_value, to=_to)
16  def mint(_to: address, _value: uint256):
17      assert msg.sender == self.minter
18      assert _to != ZERO_ADDRESS
19      self.total_supply += _value
20      create[token](_value, to=_to)
21      self.balanceOf[_to] += _value
22      log.Transfer(ZERO_ADDRESS, _to, _value)
23
24  performs: transfer[token](_value, to=_to)
25  def transfer(_to: address, _value: uint256) -> bool:
26      self.balanceOf[msg.sender] -= _value
27      transfer[token](_value, to=_to)
28      self.balanceOf[_to] += _value
29      log.Transfer(msg.sender, _to, _value)
30      return True
31
32  performs: exchange[token <-> token](1, 0, _from, msg.sender, times=_value)
33  performs: transfer[token](_value, to=_to)
34  def transferFrom(_from: address, _to: address, _value: uint256) -> bool:
35      self.balanceOf[_from] -= _value
36      self.balanceOf[_to] += _value
37      self.allowances[_from][msg.sender] -= _value
38      exchange[token <-> token](1, 0, _from, msg.sender, times=_value)
39      transfer[token](_value, to=_to)
40      log.Transfer(_from, _to, _value)
41      return True
42
43  performs: revoke[token <-> token](1, 0, to=_spender)
44  performs: offer[token <-> token](1, 0, to=_spender, times=_value)
45  def approve(_spender: address, _value: uint256) -> bool:
46      revoke[token <-> token](1, 0, msg.sender, _spender,
47      offered[token <-> token](1, 0, msg.sender, _spender))
48      self.allowances[msg.sender][_spender] = _value
49      offer[token <-> token](1, 0, to=_spender, times=_value)
50      log.Approval(msg.sender, _spender, _value)
51      return True
```

Figure 5.10.: Complete token contract, annotated with resource specifications. Transitive segment constraints in lines 10-12 link contract state and resource state. Resource ghost statements (red) manipulate the resource state. Performs-clauses declare the effects of each contract function.

```
resource: good()
resource: wei_in_auction() derived from wei

performs: create[wei_in_auction](msg.value)
performs: offer[wei_in_auction <-> good](msg.value, 1, to=self.beneficiary, times=1)
def bid():
    assert block.timestamp < self.auctionEnd and not self.ended
    assert msg.value > self.highestBid and msg.sender != self.beneficiary
    offer[wei_in_auction <-> good](msg.value, 1, to=self.beneficiary, times=1)
    self.pendingReturns[self.highestBidder] += self.highestBid
    self.highestBidder = msg.sender
    self.highestBid = msg.value

performs: destroy[wei_in_auction](self.pendingReturns[msg.sender])
def withdraw():
    pending_amount: wei_value = self.pendingReturns[msg.sender]
    self.pendingReturns[msg.sender] = 0
    send(msg.sender, pending_amount)
```

Figure 5.11.: Example usage of derived resources in a part of the auction contract. Ghost statements are red and specifications like effects-clauses (using the performs keyword) and resource declarations are green. Since the contract declares a resource wei_in_auction derived from wei, sending some wei to it when calling function bid will implicitly create the same amount of wei_in_auction, which then belongs to the bidder. Every bidder offers to exchange their wei_in_auction for the auctioned good if they win the auction. When calling withdraw, previous bidders get back the wei they sent, implicitly destroying their wei_in_auction.

```
beneficiary: address
highestBid: int256
highestBidder: address
ended: bool
pendingReturns: map(address, int256)

resource: good()
wei_in_auction() derived from wei

transitive segment constraint: ... # relate contract state and resource state

segment constraint: msg.sender != self.beneficiary ==> self.ended == old(self.ended)
transitive segment constraint: self.beneficiary == old(self.beneficiary)
transitive segment constraint: old(self.ended) ==> self.ended

performs: create[wei_in_auction](msg.value)
performs: offer[wei_in_auction <-> good](msg.value, 1, to=self.beneficiary, times=1)
def bid():
    assert block.timestamp < self.auctionEnd and not self.ended
    assert msg.value > self.highestBid and msg.sender != self.beneficiary

    offer[wei_in_auction <-> good](msg.value, 1, to=self.beneficiary, times=1)

    self.pendingReturns[self.highestBidder] += self.highestBid
    self.highestBidder = msg.sender
    self.highestBid = msg.value


performs: destroy[wei_in_auction](self.pendingReturns[msg.sender])
def withdraw():
    pending_amount: wei_value = self.pendingReturns[msg.sender]
    self.pendingReturns[msg.sender] = 0
    send(msg.sender, pending_amount)


performs: exchange[wei_in_auction <-> good](self.highestBid, 1, self.highestBidder,
self.beneficiary, times=1)
performs: destroy[wei_in_auction](self.highestBid, actor=self.beneficiary)
def endAuction():
    assert block.timestamp >= self.auctionEnd and not self.ended
    self.ended = True

    exchange[wei_in_auction <-> good](self.highestBid, 1, self.highestBidder,
    self.beneficiary, times=1)

    send(self.beneficiary, self.highestBid)
```

Figure 5.12.: Complete auction contract annotated with resource specifications.

5.5. Availability

So far, we have focused on proving integrity properties; in this section, we will briefly describe how our technique can also prove availability properties of smart contracts.

As mentioned in the introduction, for smart contracts, the blockchain itself is responsible for guaranteeing that clients can always invoke transactions on contracts. However, for each individual contract, it is still possible for functionality to become essentially unavailable, or even for third parties to perform a kind of denial-of-service attack (we will show an example below). In particular, since any transaction can abort, it is possible that some intended contract functionality can no longer be executed because any attempt to do so results in an aborted transaction.

Thus, to reason about contract availability, it is vital to be able to reason about successful contract execution as well as transaction abortion. In particular, it can be important to reason about the possible *causes* for transaction abortion.

Smart contracts can abort for four distinct reasons: (1) because an assertion statement **assert** *e* fails, if *e* is not true, (2) in Vyper's case, because some check performed implicitly by the language fails, e.g., because of an integer overflow, (3), because a call to an outside contract fails, terminating the entire transaction, or (4) because a transaction runs out of gas.

At least the latter two can potentially be influenced by third parties: A called contract can intentionally abort, can intentionally use up gas when called, or can influence the original contract's state in a way that leads to higher gas consumption for certain functions. These factors can be used by third parties to intentionally block another contract's ability to execute, i.e., to perform a denial-of-service attack.

In Fig. 5.13, we show an alternative way of paying out outstanding debts for the auction contract that is vulnerable to denial-of-service attacks. Imagine that our auction contract does not have the withdraw function shown before, and instead, the shown function payAllPending, which can be called by anyone, and is used to pay out all outstanding debts at once. This function iterates through all previous bidders with pending returns, and sends the owed Ether to each of them. However, if even one of those send operations aborts, the entire transaction will revert. As a result, any malicious contract can essentially prevent all existing bidders from getting their bid back by simply placing a bid itself, and subsequently ensuring that any attempt by the auction to send Ether back to the malicious contract aborts. This problem is the reason why most contracts that implement similar functionality offer withdraw functions

```
def payAllPending():
 assert not self.lock
 for recipient in self.pendingReturns:
  toSend = self.pendingReturns[recipient]
  self.pendingReturns[recipient] = 0
  self.lock = True
  send(recipient, value=toSend)
  self.lock = False
```

Figure 5.13.: Alternative function for paying outstanding debts in the auction contract, which is vulnerable to denial-of-service attacks. Note that the Vyper language does not actually allow iterating over maps in the pictured way, but we write the code as shown to simplify the presentation.

like the one shown in the original auction contract, which pay out only a single client at a time.

The central problem here is that functionality that is important for multiple clients can be blocked by others. Thus, we propose that the central property that should be proved for important functions of smart contracts is the following: If some (satisfiable) function precondition is fulfilled by a caller, then the function's execution either succeeds, or fails because of a reason that is controlled exclusively by the caller.

For example, for the withdraw function in the auction code, it would be too strict to demand that it must always succeed: The caller might provide too little gas, or it might fail when receiving Ether. However, both of those potential sources of transaction abortion are under the control of the caller itself[7], and cannot be influenced by third parties. For payAllPending, however, this is not the case, since any contract in the map of pending returns can force the transaction to revert.

In our implementation (though not in our formalization in the next section, where we only model successful executions), we offer a specification construct success(), which can be used in function postconditions and specifies whether a function has executed successfully. That is, one can prove postconditions of the form $P \Rightarrow$ success(), stating that some condition guarantees that the function will execute successfully. Additionally, we offer variations of this specification construct to model success modulo other factors. For example, the assertion success(unless=out_of_gas) is true if the function either executed successfully or ran out of gas (but did not fail for other reasons), and the crucial property mentioned before, that a function will succeed unless it fails because of lack of gas or a failed call to its own caller, can be written as success(unless=sender_failed). Proving postconditions including such a construct is possible using standard techniques, since it simply requires accurately modeling that (and why) different statements in a contract can abort.

As in the example of the auction contract shown above, a vital property of many contracts is that the functionality that enables other contracts to get resources (e.g., tokens or Ether) they are owed back from a contract, must not be vulnerable to denial-of-service attacks. Since this case is so common, our implementation also adds an additional specification construct for this case: the assertion accessible[R](amount, recipient), to be used in invariants, is true for a contract if there is some way for the recipient to receive the specified amount of resource R from the contract (in a way that cannot be prevented by third parties). This specification construct is essentially useful for any contract that declares derived resources and may therefore owe resources to some of its clients. Expressing this property as an invariant makes it possible to state more directly that, in any public state of the contract (or potentially under some conditions, e.g., after a certain point in time), clients can access owed resources in a way that cannot be blocked by others. To prove such an invariant, we simply translate it into a postcondition of the function that performs this transfer (which can often be inferred using heuristics, and will otherwise require additional user input to identify). More information on the handling of the success and accessible predicates can be found in the thesis of Robin Sierra [198].

[7]: Since there is an upper limit to the amount of gas a transaction may use, it may actually be impossible to successfully execute a function that consumes too much gas. While our tool does not explicitly model gas consumption, in principle, one can prove that this is not the case by showing that the maximum gas construction of a function is always below this limit.

[198]: Sierra (2019), 'Verification of Ethereum Smart Contracts Written in Vyper'

5.6. Proof Technique

In this section, we formalize our specification and verification technique for a simple smart contract language. Since our focus in this chapter is on novel specification constructs, we will not provide a proof of soundness here; the purpose of our formalization is to describe the proof obligations we have so far described informally in an unambiguous and formal way.

For a potential soundness proof, we envision the following steps: The first step is to prove the soundness of our general verification technique in the presence of untrusted code (Sections 5.2 and 5.3), by showing that the imposed proof obligations are sufficient to ensure that (transitive) segment constraints and function constraints hold, by induction of the size of the call stack of a transaction. From this, the soundness of our basic resource-based specification techniques follows immediately, since a sound proof of a coupling invariant linking resource state to contract state is sufficient to ensure that the contract state follows the rules of the resource system. Subsequently, one could (separately) prove the guarantees of the effects-system (in particular, the fact that negative effects for the caller are overapproximated by the declared effects of a function) and the consistency of the derived resource system (ensuring that owed resources are never illegally spent). We leave such a proof as future work.

Our language has the following statements:

$$s \;::=\; \texttt{skip} \mid x := e \mid \mathsf{self}.f := e \mid x := e.\mathsf{fun}(e, \mathsf{value} = e) \mid s;s \mid \mathsf{assert}\ e$$

To reflect the design of Ethereum smart contracts, only fields of self (the current contract) can be assigned; function calls take a second argument representing the amount of wei to send along with a call. We assume a standard expression language with a reserved result identifier (to refer to function results in postconditions); field lookups include those on the implicit msg and block arguments. To express two-state assertions such as our segment constraints, our formalization includes *labels* l denoting earlier points in execution, and expressions $\mathsf{old}_l(e)$ denoting the value e had at label l. We use three labels: *pre*, representing the pre-state of the current function, *last*, representing the pre-state of the current local segment, and *call*, representing the pre-state of the last call to another contract.

A state Σ has the form $\langle \mathcal{H}, \mathcal{R}, \mathcal{E}, \mathcal{O}, \sigma \rangle$, where $\mathcal{H}$ is the heap (a partial map from contract addresses and fields to their values), and σ is the current variable store. $\mathcal{E}$ is a multiset of *effects* produced so far by the current function and $\mathcal{R}$ is a record containing the state of all resources declared in the current contract. In particular, for every such resource R, $\mathcal{R}.\mathsf{balances}_R$ maps addresses to their balances, and $\mathcal{R}.\mathsf{offered}_{R \leftrightarrow R'}$ tracks the offers to exchange R against another resource R' declared in the contract. $\mathcal{R}.\mathsf{trusted}$ is a partial map from pairs of addresses to boolean values that represent whether the first address currently trusts the second; expressions can refer to these maps. Finally, $\mathcal{O}$ maps label names to pairs $(\mathcal{H}, \mathcal{R})$ that represent the heap and resource state at label l.

Expression evaluation in a state, denoted by $[\![e]\!]_{(\sigma,\mathcal{H},\mathcal{R},\mathcal{O})}$, is largely standard; most interestingly, the evaluation of $[\![\mathsf{old}_l(e)]\!]_{(\sigma,\mathcal{H},\mathcal{R},\mathcal{O})}$ is $[\![e]\!]_{(\sigma,\mathcal{H}',\mathcal{R}',\mathcal{O})}$, where $(\mathcal{H}', \mathcal{R}') = \mathcal{O}[l]$.

We now define our assertion language as follows:

$$P, Q \quad ::= \quad \mathsf{emp} \mid e \mid P * P \mid P \wedge P \mid P \mathrel{-\!\!*} P \mid P \vee P \mid$$
$$\mathsf{perf}(E) \mid \mathsf{owns}_R(e, e) \mid \mathsf{offers}_{R \leftrightarrow R}(e, e, e, e, e) \mid \mathsf{trusts}(e, e, e)$$

Assertion truth in a state is defined by a judgement $\langle \mathcal{H}, \mathcal{R}, \mathcal{E}, \mathcal{O}, \sigma \rangle \models P$ whose cases are given in Fig. 5.14. In contrast to traditional separation logics [180], we do not use the *linear/separating* aspects of the $*$ and $\mathrel{-\!\!*}$ connectives to govern access to the (already encapsulated) *heap*, but rather for the resource state and effects concepts added by our methodology[8]. The separating conjunction $P * Q$ splits the resource state and the effects into two parts; the first described by P and the second by Q. Descriptions of constituent parts of the resource state come via assertions such as $\mathsf{owns}_R(e_o, e_a)$, which prescribes that $\mathcal{R}$ is empty (no offers, no trust, and all balances are zero) *except* for the balance of e_o, which owns exactly e_a of resource R, and that $\mathcal{E}$ is empty (no effects). The (multiplicative) separating conjunction builds up larger descriptions of these states; e.g., $\mathsf{owns}_R(e_o, e_{a_1}) * \mathsf{owns}_R(e_o, e_{a_2})$ is equivalent to $\mathsf{owns}_R(e_o, e_{a_1} + e_{a_2})$. The assertions $\mathsf{offers}_{R_1 \leftrightarrow R_2}(e_f, e_t, e_{a_1}, e_{a_2}, e_n)$ and $\mathsf{trusts}(e_{c_1}, e_{c_2}, e_v)$ form the analogous base cases prescribing offers and trust. Similarly, $\mathsf{perf}(E)$ states that *exactly* the effects in multiset E have been performed and no others (and the resource state is empty). Note that $\mathsf{owns}_R(e_o, e_a)$ can be used alongside assertions containing references to the $\mathsf{balances}_R$ map; both are useful in different contexts. For example, an invariant that states that the allocation of *token* in the resource state is stored in the contract in the field $\mathbf{self}.\mathsf{tokens}$ can be easily expressed as $\mathsf{balances}_{token} = \mathbf{self}.\mathsf{tokens}$, whereas the proof rules for framing and resource statements, which we will show later, are much easier to express using the exact assertion $\mathsf{owns}_R(e_o, e_a)$. The interpretation of other assertions is standard for a classical separation logic; in particular, an assertion e is true only if there are *no* effects in the current state. As a result, assertions always have to describe the effects-state precisely: If a state containing $\mathcal{E}$ fulfills $P * \mathsf{perf}(E)$, and P does not syntactically contain any $\mathsf{perf}()$-assertions, then we must have that $\mathcal{E} = E$. This is important to ensure that functions report all effects they directly cause.

We write CS_a to refer to the entire state, including the resource state, of the contract at address a. We denote (the conjunction of) the primary contract's transitive segment constraints (which may refer to other contracts' states) by TSC, its segment constraints by SC, and its function constraints by FC. The latter two may only refer to the primary contract's state. By ITSC, we denote the conjunction of the transitive segment constraints of all known interfaces. By $\mathsf{PC}(e_1, e_2)$, we denote the conjunction of the reflexive and transitive assertions P from the privacy constraints of all known interfaces *except* those in set e_2, instantiated for e_1. In all those specification constructs, the old state is referred to without a label (simply as $\mathsf{old}(e)$), since the kind of specification construct determines which old state it refers to.

To handle the various kinds of two-state specifications our methodology employs (in each of which $\mathsf{old}(e)$ is used to denote evaluation in the appropriate "old" state), we define a judgement $\Sigma_1, \Sigma_2 \models P$ in which Σ_1 represents the appropriate state to use as the old one (e.g., for local segment constraints we use the state at the start of the local segment):

[180]: Reynolds (2002), 'Separation Logic: A Logic for Shared Mutable Data Structures'

[8] Note that it would be entirely possible to describe the heap using standard separation logic points-to predicates and separating conjunctions as well, but unlike in other programs, there is not benefit in doing so.

$$
\begin{aligned}
\langle \mathcal{H},\mathcal{R},\mathcal{E},\mathcal{O},\sigma\rangle \models \mathsf{emp} &\iff \mathcal{R} = \mathcal{R}_{empty} \land \mathcal{E} = \emptyset^{\#}\\
\langle \mathcal{H},\mathcal{R},\mathcal{E},\mathcal{O},\sigma\rangle \models e &\iff [\![e]\!]_{(\sigma,\mathcal{H},\mathcal{R},\mathcal{O})} \land \mathcal{E} = \emptyset^{\#}\\
\langle \mathcal{H},\mathcal{R},\mathcal{E},\mathcal{O},\sigma\rangle \models P * Q &\iff \exists \mathcal{R}_1,\mathcal{R}_2,\mathcal{E}_1,\mathcal{E}_2.\, \mathcal{R} = \mathcal{R}_1 \uplus \mathcal{R}_2 \land \mathcal{E} = \mathcal{E}_1 \cup^{\#} \mathcal{E}_2 \land\\
&\qquad \langle \mathcal{H},\mathcal{R}_1,\mathcal{E}_1,\mathcal{O},\sigma\rangle \models P \land \langle \mathcal{H},\mathcal{R}_2,\mathcal{E}_2,\mathcal{O},\sigma\rangle \models Q\\
\langle \mathcal{H},\mathcal{R},\mathcal{E},\mathcal{O},\sigma\rangle \models P \land Q &\iff \langle \mathcal{H},\mathcal{R},\mathcal{E},\mathcal{O},\sigma\rangle \models P \land \langle \mathcal{H},\mathcal{R},\mathcal{E},\mathcal{O},\sigma\rangle \models Q\\
\langle \mathcal{H},\mathcal{R},\mathcal{E},\mathcal{O},\sigma\rangle \models P \mathrel{-\!\!*} Q &\iff \forall \mathcal{R}',\mathcal{E}'.\, \langle \mathcal{H},\mathcal{R}',\mathcal{E}',\mathcal{O},\sigma\rangle \models P \Rightarrow\\
&\qquad \langle \mathcal{H},\mathcal{R} \uplus \mathcal{R}',\mathcal{E} \cup^{\#} \mathcal{E}',\mathcal{O},\sigma\rangle \models Q\\
\langle \mathcal{H},\mathcal{R},\mathcal{E},\mathcal{O},\sigma\rangle \models P \lor Q &\iff \langle \mathcal{H},\mathcal{R},\mathcal{E},\mathcal{O},\sigma\rangle \models P \lor \langle \mathcal{H},\mathcal{R},\mathcal{E},\mathcal{O},\sigma\rangle \models Q\\
\langle \mathcal{H},\mathcal{R},\mathcal{E},\mathcal{O},\sigma\rangle \models \mathsf{perf}(E) &\iff \mathcal{R} = \mathcal{R}_{empty} \land \mathcal{E} = E\\
\langle \mathcal{H},\mathcal{R},\mathcal{E},\mathcal{O},\sigma\rangle \models \mathsf{owns}_R(e_o, e_a) &\iff \mathcal{R} = \mathcal{R}_{empty}[\mathsf{balances}_R := b] \land \mathcal{E} = \emptyset^{\#}\\
\text{where} &\qquad b = [o \mapsto a, \_ \mapsto 0],\\
&\qquad o = [\![e_o]\!]_{(\sigma,\mathcal{H},\mathcal{R},\mathcal{O})}, a = [\![e_a]\!]_{(\sigma,\mathcal{H},\mathcal{R},\mathcal{O})}\\
\langle \mathcal{H},\mathcal{R},\mathcal{E},\mathcal{O},\sigma\rangle \models \mathsf{offers}_{R_1 \leftrightarrow R_2}(e_f, e_t, e_{a_1}, e_{a_2}, e_n) &\iff \mathcal{R} = \mathcal{R}_{empty}[\mathsf{offered}_{R_1 \leftrightarrow R_2} := o] \land \mathcal{E} = \emptyset^{\#}\\
\text{where} &\qquad o = [(f, t, a_1, a_2) \mapsto n, \_ \mapsto 0],\\
&\qquad f = [\![e_f]\!]_{(\sigma,\mathcal{H},\mathcal{R},\mathcal{O})}, t = [\![e_t]\!]_{(\sigma,\mathcal{H},\mathcal{R},\mathcal{O})}\\
&\qquad a_1 = [\![e_{a_1}]\!]_{(\sigma,\mathcal{H},\mathcal{R},\mathcal{O})}, a_2 = [\![e_{a_2}]\!]_{(\sigma,\mathcal{H},\mathcal{R},\mathcal{O})},\\
&\qquad n = [\![e_n]\!]_{(\sigma,\mathcal{H},\mathcal{R},\mathcal{O})}\\
\langle \mathcal{H},\mathcal{R},\mathcal{E},\mathcal{O},\sigma\rangle \models \mathsf{trusts}(e_{c_1}, e_{c_2}, e_v) &\iff \mathcal{R} = \mathcal{R}_{empty}[\mathsf{trusted} := t] \land \mathcal{E} = \emptyset^{\#}\\
\text{where} &\qquad t = [(c_1, c_2) \mapsto v], v = [\![e_v]\!]_{(\sigma,\mathcal{H},\mathcal{R},\mathcal{O})},\\
&\qquad c_1 = [\![e_{c_1}]\!]_{(\sigma,\mathcal{H},\mathcal{R},\mathcal{O})}, c_2 = [\![e_{c_2}]\!]_{(\sigma,\mathcal{H},\mathcal{R},\mathcal{O})}
\end{aligned}
$$

Figure 5.14.: Definition of assertion truth in a state. If r is a record, $r[f := v]$ updates field f of the record to value v. Operator $\uplus$ for resource states is defined s.t. it performs pointwise addition for balance and offer maps, and combination of partial functions with disjoint domains for the trusted map. $\mathcal{R}_{empty}$ denotes an empty resource state (i.e., all balances are zero, there are no offers, and the domain of the partial trust map is empty). We write $\emptyset^{\#}$ for the empty multiset and $\cup^{\#}$ for multiset union.

Definition 5.6.1 *For any two states Σ_1 and Σ_2 s.t. $\Sigma_1 = \langle \mathcal{H}_1, \mathcal{R}_1, \mathcal{E}_1, \mathcal{O}_1, \sigma_1\rangle$ and $\Sigma_2 = \langle \mathcal{H}_2, \mathcal{R}_2, \mathcal{E}_2, \mathcal{O}_2, \sigma_2\rangle$, we define $\Sigma_1, \Sigma_2 \models P$ to hold if and only if $\langle \mathcal{H}_2, \mathcal{R}_2, \mathcal{E}_2, \mathcal{O}_2[last \mapsto (\mathcal{H}_1, \mathcal{R}_1)], \sigma_2\rangle \models P[\mathsf{old}_{last}(\_)/\mathsf{old}(\_)]$*

Using this notion, we can define two-state assertion reflexivity and transitivity as follows:

Definition 5.6.2 *An assertion P is* reflexive *if, for all Σ_0, Σ_1, if $\Sigma_0, \Sigma_1 \models P$, then $\Sigma_1, \Sigma_1 \models P$. An assertion P is* transitive *if, for all $\Sigma_0, \Sigma_1, \Sigma_2$, if $\Sigma_0, \Sigma_1 \models P$ and $\Sigma_1, \Sigma_2 \models P$, then $\Sigma_0, \Sigma_2 \models P$.*

We can now also define assertion *stability*, i.e., the fact that an assertion is preserved by another:

Definition 5.6.3 *An assertion P is* stable under *Q, written $\mathsf{stable}(P, Q)$, if, for all $\Sigma_0, \Sigma_1, \ldots, \Sigma_n$, if $\Sigma_0, \Sigma_1 \models P$ and $\Sigma_i, \Sigma_{i+1} \models Q$ for all $i \in \{1, \ldots, n-1\}$, then $\Sigma_0, \Sigma_n \models P$.*

Finally, we need to define the notion of a stateless assertion:

Definition 5.6.4 *An assertion P is* stateless *if it does not refer to the current state (including resource state) or an old state* except *for the pre-state (i.e., only old-expressions with label pre are allowed).*

We define our proof technique via a Hoare Logic formulation, whose details are given in Fig. 5.16 and Fig. 5.17. Since the rule for calls is quite complex, we also show a simplified version (*SCall*) of it in Figure 5.15, which we will use for illustration purposes. $FV(P)$ are the free variables in

$$\frac{e_r : T \qquad T.\mathsf{fun}(x)\ \textbf{ensures}\ Q\ \textbf{performs}\ S \qquad S' \subseteq S}{\vdash \left\{\begin{array}{c}\mathsf{TSC}[\mathsf{old}_{last}/\mathsf{old}]\\ \wedge\mathsf{SC}[\mathsf{old}_{last}/\mathsf{old}]\\ \wedge e_O\end{array}\right\} \begin{array}{c} x := e_r.\mathsf{fun}\\ (e_a, \mathsf{value} = e_v)\end{array} \left\{\begin{array}{c}\mathsf{perf}(S')[e_r/\textbf{self}][x/\mathsf{result}]\\ [\textbf{self}/\textbf{msg}.sender][e_a/x]*\\ \mathsf{old}_{call}(e_O)\wedge\\ \mathsf{TSC}[\mathsf{old}_{call}/\mathsf{old}]\wedge\\ \mathsf{FC}[\mathsf{old}_{call}/\mathsf{old}]\wedge\\ Q[e_r/\textbf{self}][x/\mathsf{result}]\\ [\textbf{self}/\textbf{msg}.sender][e_a/x]\wedge\\ e_N \Rightarrow \mathsf{old}_{last}(e_N)\end{array}\right\}}\ (\textsc{SCall})$$

Figure 5.15.: Simplified proof rule for calls.

P, $mods(s)$ are the variables modified by s. We write $e[e_1/e_2]$ to substitute all occurrences of e_2 in e by e_1.

The rules for ordinary statements are standard; the bulk of the work happens in the rule for calls, as well as in the rules for resource statements. We will explain the checks for every element of our methodology step by step.

We will first explain the entire proof system with the simplified rule (*SCall*) for calls, and will subsequently explain the additions made by the actual rule (*Call*). To ensure that ordinary (SC) and transitive segment constraints (TSC) hold at the end of every local segment, the function rule requires them to be true at the end of each function, and the simplified call rule requires them to be true in the precondition of the Hoare triple, using the state labeled "last" as old state. This notion of "last" state is reset in the postcondition to the new current state (we begin a new local segment), as indicated by $e_N \Rightarrow \mathsf{old}_{last}(e_N)$ which can be instantiated for any e_N. A similar connection can be made to any facts known before the call via e_O.

The frame rule (*Frame*) is non-standard in that it ensures that *no* information about the last state or the current state can be framed around calls; this represents the fact that the entire contract state can change with every call, since calls to unverified outside contracts can result in arbitrary, unknown callbacks. After a call, one may however assume the transitive segment constraints and function constraints w.r.t. the call's pre-state. To remember information about said pre-state, we use the same trick as before, and allow assuming any expression e_O after a call about its pre-state that was known to be true before the call.

In constructors, no calls are allowed, and we check at the end that transitive segment constraints hold in the current state w.r.t. itself, which ensures that all single-state invariants contained in the transitive segment constraints are established. The rule for constructors also performs the necessary checks of transitivity and reflexivity of transitive segment and function constraints, and ensures that transitive segment constraints fulfill stability criteria described previously.

The proof obligations for function constraints work in a similar way: They must be shown to hold at the end of each function w.r.t. to its pre-state (which may again be remembered by assuming that everything that holds in the beginning of the function holds in its pre-state), and may be assumed after a call w.r.t. to the call's pre-state.

Each ghost statement of Sec. 5.4 gets a corresponding Hoare Logic rule. These rules are mostly intuitive: Each of them (a) checks that the participant for whom the statement has a negative effect (e.g., giving away resource) trusts the current msg.sender (typically, this is simply who they are), (b) checks that required resources for the statement are available, consuming them, (c) adds appropriate new resources in the postcondition assertion, and (d) records the effect that was performed. A transfer, for example, requires that the sender initially has the resources that are to be transferred, and ends in a state where the recipient has them. It also requires that the caller of the function is trusted by the address whose resources are being transferred (everybody always trusts themselves). Finally, the conclusion states that a transfer-effect has occurred. Rules for other resource statements are analogous; for example, the rule for exchanges requires two compatible offers (unless the offered amount of a party is zero) and consumes them, switches ownership of the involved resources, and records that an exchange-effect has occurred.

The rule for resource creation requires that e_c (the party who is creating the new resource of type R) owns a creator-resource for R, which represents the right to create new resources. These creator resources can be created and given to arbitrary addresses in the contract's constructor: In the constructor rule, the caller of the constructor is given the right to create such creator-resources for any resource the contract declares. This is denoted by CREATORS, which, for a contract with resources $R_0, \ldots, R_n$, is defined as $\mathsf{owns}_{\mathsf{creator}(\mathsf{creator}(R_0))}(\mathsf{msg.sender}, 1) * \cdots * \mathsf{owns}_{\mathsf{creator}(\mathsf{creator}(R_n))}(\mathsf{msg.sender}, 1)$.

The function and constructor rules ensure that, at the end of the function or constructor, the multiset of recorded effects is exactly that which has been declared.

This concludes the description of our system with the simplified call rule; the actual call rule considers three additional points:

- Since calls can send wei and therefore decrease the balance of the current contract, the actual call rule checks that segment constraints and transitive segment constraints hold in the *updated* state after removing e_a wei from the contract's balance.
- The actual call rule considers the fact that calls can also interact with the resource state. First, any subset S' of the effects declared by the called function may be recorded by the caller. Second, if a called function declares resource effects w.r.t. a resource R s.t. the current contract has a resource D derived from R, then these effects lead to implicit effects on the derived resources. For example, transferring R away to someone implicitly destroys the same number of D they must currently own, and requires that they have offered to exchange that amount of D for the same amount of R. This is captured by the functions derCreated() and derDestroyed(), the former of which defines which derived resources are implicitly created by an effect, and the latter defines which derived resources are implicitly destroyed. Their definitions can be found in Fig. 5.18. The premise of the call rule ensures that transitive segment constraints etc. hold in a state where destroyed derived resources have already been removed and created derived resources have already been added. The derDestroyed() clause also ensures that no offers are made

$$\dfrac{}{\vdash \{\mathsf{true}\}\ \mathbf{assert}\ e\ \{e\}}\ (\textsc{Assert}) \qquad \dfrac{}{\vdash \{Q[e/x]\}\ x := e\ \{Q\}}\ (\textsc{Assign})$$

$$\dfrac{}{\vdash \left\{Q[(e' = e_1 ? e_2 : e'.f)/e'.f]\right\}\ e_1.f := e_2\ \{Q\}}\ (\textsc{Write})$$

$$\dfrac{\begin{array}{c}\mathsf{TSC}' = \mathsf{TSC}[\mathsf{old}_{last}(\_)/\mathsf{old}(\_)] \\ [(e' = \mathbf{self}?(\mathbf{self}.\mathsf{balance} - e_a) : e'.\mathsf{balance})/e'.\mathsf{balance}] \\ \mathsf{SC}' = \mathsf{SC}[\mathsf{old}_{last}(\_)/\mathsf{old}(\_)][(e' = \mathbf{self}?(\mathbf{self}.\mathsf{balance} - e_a) : e'.\mathsf{balance})/e'.\mathsf{balance}] \\ e_r : T \quad T.\mathsf{fun}(x)\ \mathbf{ensures}\ Q\ \mathbf{performs}\ S \quad S' \subseteq S'' \\ S'' = S[e_r/\mathbf{self}][x/\mathsf{result}][\mathbf{self}/\mathbf{msg}.sender][e_a/x] \\ secondary(e_r) \Rightarrow \mathsf{stable}(\mathsf{TSC} \wedge e_S, \mathsf{ITSC} \wedge \mathsf{PC}(\mathbf{self}, \{e_r\}) \wedge \mathsf{CS}_{\mathbf{self}} = \mathsf{old}(\mathsf{CS}_{\mathbf{self}})) \\ secondary(e_r) \Rightarrow \mathsf{stable}(\mathsf{TSC} \wedge e_S, \mathsf{ITSC} \wedge \mathsf{FC})\end{array}}{\vdash \left\{\begin{array}{c}\mathsf{derDestroyed}(S'')* \\ (\mathsf{derCreated}(S'') \mathbin{-\!\!*} \\ (\mathsf{TSC}' \wedge \mathsf{SC}' \\ \wedge e_O \wedge e_S))\end{array}\right\} \begin{array}{c} x := e_r.\mathsf{fun} \\ (e_a, \mathsf{value} = e_v)\end{array} \left\{\begin{array}{c}\mathsf{perf}(\mathsf{derPerformed}(S''))* \\ \mathsf{perf}(S') * (\mathsf{old}_{call}(e_O) \wedge \\ \mathsf{TSC}[\mathsf{old}_{call}(\_)/\mathsf{old}(\_)] \wedge \\ \mathsf{FC}[\mathsf{old}_{call}(\_)/\mathsf{old}(\_)] \wedge \\ Q[e_r/\mathbf{self}][x/\mathsf{result}] \\ [\mathbf{self}/\mathbf{msg}.sender][e_a/x] \\ [\mathsf{old}_{call}(\_)/\mathsf{old}(\_)] \wedge \\ e_N \Rightarrow \mathsf{old}_{last}(e_N))\end{array}\right\}}\ (\textsc{Call})$$

$$\dfrac{\vdash \{P\}\ s_1\ \{R\} \qquad \vdash \{R\}\ s_2\ \{Q\}}{\vdash \{P\}\ s_1; s_2\ \{Q\}}\ (\textsc{Seq}) \qquad \dfrac{\vdash \left\{P'\right\}\ s\ \left\{Q'\right\} \quad P \models P' \quad Q' \models Q}{\vdash \{P\}\ s\ \{Q\}}\ (\textsc{Cons})$$

$$\dfrac{\begin{array}{c} FV(R) \cap mods(s) = \emptyset \qquad \vdash \{P\}\ s\ \{Q\} \\ R \text{ is stateless if } s \text{ contains a call}\end{array}}{\vdash \{P * R\}\ s\ \{Q * R\}}\ (\textsc{Frame})$$

$$\dfrac{\vdash \left\{\begin{array}{c}\exists l.\, \mathsf{TSC}[\mathsf{old}_l(\_)/\mathsf{old}(\_)] \wedge \\ e_{N_1} \Rightarrow \mathsf{old}_{last}(e_{N_1}) \wedge \\ e_{N_2} \Rightarrow \mathsf{old}_{pre}(e_{N_2})\end{array}\right\} s \left\{\begin{array}{c} Q[\mathsf{old}_{pre}(\_)/\mathsf{old}(\_)] \wedge \\ \mathsf{FC}[\mathsf{old}_{pre}(\_)/\mathsf{old}(\_)] \\ \wedge \mathsf{TSC}[\mathsf{old}_{last}(\_)/\mathsf{old}(\_)] \\ \wedge \mathsf{SC}[\mathsf{old}_{last}(\_)/\mathsf{old}(\_)] \\ * \mathsf{perf}(S)\end{array}\right\}}{\mathbf{def}\ f(x) : T\ \{s\}\ \mathbf{ensures}\ Q\ \mathbf{performs}\ S}\ (\textsc{Func})$$

$$\dfrac{\begin{array}{c}\vdash \left\{\mathsf{CREATORS} * \mathsf{default}(\mathsf{CS}_{\mathbf{self}})\right\}\ s \left\{\begin{array}{c}\mathsf{TSC}[\_/\mathsf{old}(\_)] \wedge Q \\ * \mathsf{perf}(S)\end{array}\right\} \\ \mathsf{TSC} \text{ precisely determines resource state.} \\ \mathsf{TSC}, \mathsf{FC} \text{ are reflexive and transitive.} \\ s \text{ does not contain any calls.} \\ \mathsf{stable}(\mathsf{TSC}, \mathsf{ITSC} \wedge \mathsf{PC}(\mathbf{self}, \emptyset) \wedge \mathsf{CS}_{\mathbf{self}} = \mathsf{old}(\mathsf{CS}_{\mathbf{self}}))\end{array}}{\mathbf{def}\ \mathsf{init}(x)\ \{s\}\ \mathbf{ensures}\ Q\ \mathbf{performs}\ S}\ (\textsc{Init})$$

Figure 5.16.: Statement, function and constructor proof rules.

to give away resource R by a contract that defines D, and that the contract cannot trust someone w.r.t. to the contract that declares R; either of these could lead to some amount of R being removed from the contract without the appropriate checks that the receiver has sufficient amounts of the derived resource D. Finally, the call-rule ensures that all such implicit effects on derived resources (defined by derPerformed()) are also recorded.

- The actual call also performs additional checks in the case where the call target is a secondary contract, i.e., a trusted outside contract which may be constrained by an inter-contract invariant. For calls to secondary contracts, the call rule requires that the transitive

$$\frac{R \neq \text{wei} \qquad R \text{ is not derived}}{\vdash \left\{\begin{array}{c}(e_a \neq 0 \Rightarrow\\ \text{trusts}(e_c, \textbf{msg}.\text{sender}, \text{true}))*\\ \text{owns}_{\text{creator}(R)}(e_c, 1)\\ \wedge e_a \geq 0\end{array}\right\} \text{create}_R(e_c, e_t, e_a) \left\{\begin{array}{c}(e_a \neq 0 \Rightarrow\\ \text{trusts}(e_c, \textbf{msg}.\text{sender}, \text{true}))*\\ \text{owns}_R(e_t, e_a)*\\ \text{perf}(\text{create}_R(e_t, e_a))*\\ \text{owns}_{\text{creator}(R)}(e_c, 1)\end{array}\right\}} \ (\textsc{Create})$$

$$\frac{R \neq \text{wei} \qquad R \text{ is not derived}}{\vdash \left\{\begin{array}{c}(e_a \neq 0 \Rightarrow\\ \text{trusts}(e_f, \textbf{msg}.\text{sender}, \text{true}))*\\ \text{owns}_R(e_f, e_a) \wedge e_a \geq 0\end{array}\right\} \text{destroy}_R(e_f, e_a) \left\{\begin{array}{c}(e_a \neq 0 \Rightarrow\\ \text{trusts}(e_f, \textbf{msg}.\text{sender}, \text{true}))*\\ \text{perf}(\text{destroy}_R(e_f, e_a))\end{array}\right\}} \ (\textsc{Destroy})$$

$$\frac{}{\vdash \left\{\begin{array}{c}e_a \geq 0*\\ (a \neq 0 \Rightarrow\\ \text{trusts}(e_f, \textbf{msg}.\text{sender}, \text{true}))*\\ \text{owns}_R(e_f, e_a) \wedge e_a \geq 0\end{array}\right\} \text{transfer}_R(e_f, e_t, e_a) \left\{\begin{array}{c}\text{owns}_R(e_t, e_a)*\\ (e_a \neq 0 \Rightarrow\\ \text{trusts}(e_f, \textbf{msg}.\text{sender}, \text{true}))\\ *\text{perf}(\text{transfer}_R(f, t, a))\end{array}\right\}} \ (\textsc{Transfer})$$

$$\frac{}{\vdash \left\{\ \text{trusts}(\textbf{msg}.\text{sender}, e_c, \_)\ \right\} \text{trust}(e_c, e_v) \left\{\begin{array}{c}\text{trusts}(\textbf{msg}.\text{sender}, e_c, e_v)\\ *\text{perf}(\text{trust}_{\textbf{self}}(\textbf{msg}.\text{sender}, e_c, e_v))\end{array}\right\}} \ (\textsc{Trust})$$

$$\frac{R_1 \neq \text{wei} \qquad \text{No resource derived from } R_1.}{\vdash \left\{\begin{array}{c}e_{a_1} \geq 0 \wedge e_{a_2} \geq 0\\ \wedge e_n \geq 0 \wedge \text{emp}*\\ (e_n \neq 0 \Rightarrow\\ \text{trusts}(e_f, \textbf{msg}.\text{sender}, \text{true}))\end{array}\right\} \begin{array}{c}\text{offer}_{R_1 \leftrightarrow R_2}\\ (e_f, e_t, e_{a_1}, e_{a_2}, e_n)\end{array} \left\{\begin{array}{c}\text{offers}_{R_1 \leftrightarrow R_2}(e_f, e_t, e_{a_1}, e_{a_2}, e_n)\\ *\text{perf}(\text{offer}_{R_1 \leftrightarrow R_2}\\ (e_f, e_t, e_{a_1}, e_{a_2}, e_n))\\ *(e_a \neq 0 \Rightarrow\\ \text{trusts}(e_f, \textbf{msg}.\text{sender}, \text{true}))\end{array}\right\}} \ (\textsc{Offer})$$

$$\frac{}{\vdash \left\{\begin{array}{c}e_{a_1} \geq 0 \wedge e_{a_2} \geq 0\\ \wedge e_n \geq 0 \wedge \text{emp}*\\ (e_n \neq 0 \Rightarrow\\ \text{trusts}(e_f, \textbf{msg}.\text{sender}, \text{true}))\\ *\text{offers}_{R_1 \leftrightarrow R_2}(e_f, e_t, e_{a_1}, e_{a_2}, e_n)\end{array}\right\} \begin{array}{c}\text{revoke}_{R_1 \leftrightarrow R_2}\\ (e_f, e_t, e_{a_1}, e_{a_2}, e_n)\end{array} \left\{\begin{array}{c}(e_a \neq 0 \Rightarrow\\ \text{trusts}(e_f, \textbf{msg}.\text{sender}, \text{true}))*\\ \text{perf}(\text{revoke}_{R_1 \leftrightarrow R_2}\\ (e_f, e_t, e_{a_1}, e_{a_2}, e_n))\end{array}\right\}} \ (\textsc{Revoke})$$

$$\frac{}{\vdash \left\{\begin{array}{c}e_{a_1} \geq 0 * e_{a_2} \geq 0\\ *(e_{a_1} > 0 \Rightarrow\\ \text{offers}_{R_1 \leftrightarrow R_2}(e_f, e_t, e_{a_1}, e_{a_2}, 1))\\ *(e_{a_2} > 0 \Rightarrow\\ \text{offers}_{R_2 \leftrightarrow R_1}(e_t, e_f, e_{a_2}, e_{a_1}, 1))\\ *\text{owns}_{R_1}(e_f, e_{a_1}) * \text{owns}_{R_2}(e_t, e_{a_2})\end{array}\right\} \begin{array}{c}\text{exchange}_{R_1 \leftrightarrow R_2}\\ (e_f, e_t, e_{a_1}, e_{a_2})\end{array} \left\{\begin{array}{c}\text{owns}_{R_1}(e_t, e_{a_1})*\\ \text{owns}_{R_2}(e_f, e_{a_2})*\\ \text{perf}(\text{exchange}_{R_1 \leftrightarrow R_2}\\ (e_f, e_t, e_{a_1}, e_{a_2}, 1))\end{array}\right\}} \ (\textsc{Exchange})$$

Figure 5.17.: Rules for resource ghost statements.

segment constraints, potentially *strengthened* by conjoining them with some arbitrary information e_S that is known about the current state (like the fact that **self**.lock is set in the example in Sec. 5.3), are stable under each of the following two assertions:

1. The conjunction of the transitive segment constraints of all interfaces, the knowledge that the primary contract's state does not change, and the privacy constraints of all secondary contracts *except* the called one (which is now trivial).
2. The conjunction of the transitive segment constraints of all interfaces and the function constraints of the primary contract.

Fig. 5.19 illustrates the scenario where the primary contract calls a

$$
\begin{aligned}
\text{derCreated}(\text{transfer}_R(f, \textbf{self}, a)) &= \text{owns}_D(f, a)\\
\text{derCreated}(\text{exchange}_{R\leftrightarrow R'}(f, \textbf{self}, a_1, a_2, n)) &= \text{owns}_D(f, n * a_1)\\
\text{derCreated}(\text{create}_R(\textbf{self}, a)) &= \text{owns}_D(\textbf{msg}.\text{sender}, a)\\
\text{derCreated}(\_) &= \text{emp}\\
\text{derDestroyed}(\text{transfer}_R(\textbf{self}, t, a)) &= \text{owns}_D(t, a) * \text{offers}_{D\leftrightarrow R}(t, \textbf{self}, 1, 1, a)\\
\text{derDestroyed}(\text{destroy}_R(\textbf{self}, a)) &= \text{owns}_D(\textbf{msg}.\text{sender}, a) *\\
&\ \text{offers}_{D\leftrightarrow R}(\textbf{msg}.\text{sender}, \textbf{self}, 1, 1, a)\\
\text{derDestroyed}(\text{offer}_{R\leftrightarrow R'}(\textbf{self}, t, a_1, a_2, n)) &= \text{false}\\
\text{derDestroyed}(\text{trust}_c(\textbf{self}, t, v)) &= \text{false if } c \text{ declares } R\\
\text{derDestroyed}(\_) &= \text{emp}\\
\text{derPerformed}(\text{transfer}_R(f, \textbf{self}, a)) &= \{\text{create}_D(f, a)\}^{\#}\\
\text{derPerformed}(\text{exchange}_{R\leftrightarrow R'}(f, \textbf{self}, a_1, a_2, n)) &= \{\text{create}_D(f, n * a_1)\}^{\#}\\
\text{derPerformed}(\text{create}_R(\textbf{self}, a)) &= \{\text{create}_D(\textbf{msg}.\text{sender}, a)\}^{\#}\\
\text{derPerformed}(\text{transfer}_R(\textbf{self}, t, a)) &= \{\text{destroy}_D(t, a)\}^{\#}\\
\text{derPerformed}(\text{destroy}_R(\textbf{self}, a)) &= \{\text{destroy}_D(\textbf{msg}.\text{sender}, a)\}^{\#}\\
\text{derPerformed}(\_) &= \emptyset^{\#}
\end{aligned}
$$

Figure 5.18.: Functions describing the implicit consequences of the effects of called functions on derived resources. D is assumed a resource derived from R. We write $\{...\}^{\#}$ for multiset literals.

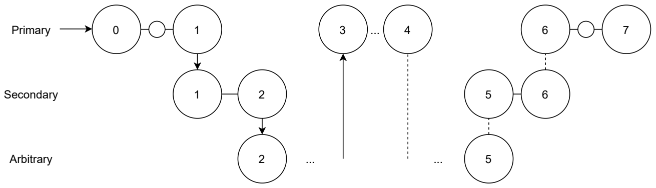


Figure 5.19.: Example showing a call from a primary to a secondary contract that leads to a re-entrant call to the primary contract.

secondary contract. The strengthened invariant is known to hold in state 1. Whatever local changes the secondary contract performs cannot break the strengthened invariant, since it is stable under the first assertion (e.g., in the example, changes in the secondary contract cannot break the strengthened invariant that remains true as long as **self**.lock is set). Any contracts executing between states 2 and 5 cannot break the original non-strengthened invariant by the reasoning laid out in Sec. 5.3. Additionally, because the strengthened invariant is stable under the second assertion, it will also be re-established by the time any re-entrant calls the secondary contract may (transitively) make on the primary contract return (states 3 and 4); in the example, a function constraint guarantees that **self**.lock will again be set when such a re-entrant call returns. As a result, again because of stability under the first assertion, the secondary contract again cannot break the invariant after the return (between states 5 and 6) either.

5.7. Implementation and Evaluation

We have implemented our work in 2Vyper, an automated verification tool for the Vyper language that is available open source[9]. Since the Vyper language is syntactically similar to Python, 2Vyper is able to reuse parts of the implementation of Nagini; additionally, it uses the standard Vyper compiler to type-check input programs. It encodes Vyper programs and specifications into the Viper intermediate verification language [159], and uses Viper's infrastructure and ultimately the SMT-solver Z3 [156] to verify the program or otherwise return errors and counterexamples.

[9] `https://github.com/viperproject/2vyper`

[159]: Müller et al. (2016), 'Viper: A Verification Infrastructure for Permission-Based Reasoning'

[156]: Moura et al. (2008), 'Z3: An Efficient SMT Solver'

```
contract Client:
  def client():
    self.token.approveAndCall(self.service, amount, data

contract Token:
  def transferFrom(from : address, amount: uint256):
    # transfer 'amount' from 'from' to msg.sender
    # if msg.sender has a sufficient allowance

  #@ performs: revoke[token <-> token](1, 0, to=spender)
  #@ performs: offer[token <-> token](1, 0, to=spender, times=amount)
  def approveAndCall(spender: address, amount: uint256, data: bytes[1024]):
    #@ revoke[token <-> token](1, 0, to=spender)
    self.allowances[msg.sender][spender] = amount
    #@ offer[token <-> token](1, 0, to=spender, times=amount)
    ERC1363Spender(spender).onApprovalReceived(msg.sender, amount, data)

contract Service implements ERC1363Spender:

  def onApprovalReceived(sender: address, amount: uint256, data: bytes[1024]):
    self.token.transferFrom(sender, amount)
    self.performService(sender, amount, data)
```

Figure 5.20.: Simplified code example showing the functionality of ERC1363 payments. Function approveAndCall also shows the specification syntax used by 2Vyper. The client calls approveAndCall on the token contract and supplies as arguments both the service provider and the input for the requested service. The token contract stores that the service provider may take tokens from the client (in field allowances), and then invokes onApprovalReceived on the service provider, which re-entrantly calls transferFrom to take its tokens and then performs the service. This architecture intentionally uses re-entrancy to allow clients to do in one transaction what would usually require two (one for setting the allowance, one for invoking the service).

While less commonly used than Solidity, Vyper puts a stronger focus on correctness and simplicity, by preventing some errors on the language level (such as over- or underflows, which automatically revert the transaction, unlike in Solidity) and omitting some language features that make code more difficult to reason about (such as inheritance). 2Vyper supports the entire Vyper language as of version 0.2.0 as well as several previous versions, and is intended for full-fledged verification of real-world contracts.

2Vyper specifications are written as ♯@ comments in the source code, and use Vyper syntax wherever possible. Fig. 5.20 shows a simplified excerpt of a function annotated with specifications and containing ghost statements for resource manipulation. In addition to the core correctness properties we have focused on in this chapter, 2Vyper also supports reasoning about additional language features (e.g., events), additional abstraction functions for commonly-used cases (e.g., to refer to the sum of all values in a map), special support for reasoning about non-linear arithmetic (which is difficult for SMT solvers but commonly used in real-world contracts), and support for trading off automation versus verification performance in several ways, e.g. by choosing whether to unroll loops or verify them using user-specified invariants. More information about 2Vyper's additional features can be found in the theses of Robin Sierra [198] and Christian Bräm [42].

[198]: Sierra (2019), 'Verification of Ethereum Smart Contracts Written in Vyper'

[42]: Bräm (2020), 'Verification of Advanced Properties for Real World Vyper Contracts'

Like Nagini, 2Vyper is able to provide counterexamples when a contract's verification fails. Here, we show an error for an incorrect version of the auction contract:

```
Verification failed
Errors:
Invariant not preserved by bid. Assertion (not self.ended ==>
sum(self.pendingReturns) + self.highestBid == sum(received()) -
sum(sent())) might not hold. (auction_fail.vy@79.1,
via invariant at auction_fail.vy@46.15)
Counterexample:
  block.coinbase = 0x000000000000000000000000000000000000029b
  block.difficulty = 160
  block.number = 0
  block.prevhash = [ _ -> 0 ]: 32
```

```
    block.timestamp = 0
    chain.id = 48
    msg.gas = 244
    msg.sender = 0x000000000000000000000000000000000000005e
    msg.value = 4683
    old(received()) = { 0x0000000000000000000000000000000000000009 –> 913,
      0x000000000000000000000000000000000000005e –> 462, _ –> 0 }
    old(self.auctionEnd) = 161
    old(self.auctionStart) = 191
    old(self.balance) = 912
    old(self.beneficiary) = 0x000000000000000000000000000000000000005d
    old(self.codesize) = –542
    old(self.ended) = False
    old(self.highestBid) = 1
    old(self.highestBidder) = 0x0000000000000000000000000000000000000009
    old(self.is_contract) = False
    old(self.pendingReturns) = { 0x000000000000000000000000000000000000005e –> 462,
      0x0000000000000000000000000000000000000009 –> 911, _ –> 0 }
    old(selfdestruct()) = False
    old(sent()) = { 0x0000000000000000000000000000000000000009 –> 1, _ –> 0 }
    out_of_gas() = False
    overflow() = False
    received() = { _ –> 0 }
    self.auctionEnd = 161
    self.auctionStart = 191
    self.balance = 13095
    self.beneficiary = 0x000000000000000000000000000000000000005d
    self.codesize = –542
    self.ended = False
    self.highestBid = 4683
    self.highestBidder = 0x000000000000000000000000000000000000005e
    self.is_contract = False
    self.pendingReturns = { 0x000000000000000000000000000000000000005e –> 462,
      0x0000000000000000000000000000000000000009 –> 911, _ –> 0 }
    selfdestruct() = False
    sent() = { 0x0000000000000000000000000000000000000009 –> 1, _ –> 0 }
    success() = True
    tx.origin = 0x0000000000000000000000000000000000000032
  Verification took 12.72 seconds.

```

Users can use counterexample information to infer, for example, *why* a condition does not hold (e.g., if success() is False, this could be due to an overflow, out-of-gas error, or a failed call; the counterexample lists separate flags for each of these cases). Additionally, they can often be used to locate the program path on which an error occurs, by manually inspecting branches in the function that fails to verify and evaluating branch conditions using the values in the counterexample. Furthermore, as in the example, Vyper states often contain a number of maps and lists, which the verifier cannot always reason about precisely without programmer guidance (e.g., in the form of additional assertions). If a counterexample shows a state that is impossible, this is a sign that verification failed due to incompleteness, and additional programmer intervention is needed.

5.7.1. Evaluation Examples

We have evaluated our approach on a number of real-world smart contracts focusing on existing contracts written in Vyper as well as those involving pertinent features such as inter-contract collaboration or re-entrancy bugs [10, 37, 73, 76, 152, 173, 214]. We manually translated some

[10]: Arumugam (2019), *Serenuscoin contract*
[37]: Blockchains LLC (2016), *Decentralized Autonomous Organization (DAO) Framework*
[73]: Ethereum (2021), *Solidity by example*
[76]: Ethereum (2021), *Vyper example contracts*
[152]: Minacori (2021), *ERC-1363 Payable Token*
[173]: Permenev et al. (2019), *VerX smart contract verification benchmarks*
[214]: Uniswap (2019), *Uniswap version 1*

Table 5.1.: Evaluated examples. LOC are total lines of code, *including* specification, excluding comments and whitespace. Ann. are lines of specification. IF LOC and IF Ann. have the same meaning for the interfaces that were required to verify the contract, and T is verification time in seconds. Contracts marked with a star are simplified versions of the original; applications marked with one or two daggers collaborate with an external ERC20 or ERC1363 contract, respectively (accessed through an interface).

Application	Contract	LOC	Ann.	IF LOC	IF Ann.	T
auction	auction	63	30	-	-	12.72
auction_token	auction_token††	96	37	88	33	23.67
TheDAO	TheDAO*	17	2	-	-	5.13
ERC20	ERC20	98	31	67	25	11.51
ERC721	ERC721	178	32	-	-	15.95
ERC1363	ERC1363	142	31	88	33	22.59
ICO	gv_option_token	98	26	86	36	10.03
	gv_token	121	24	107	49	15.21
	gv_option_program*	86	12	154	67	29.19
	ico_alloc*	159	30	261	116	101.86
Mana	token*	18	3	50	25	2.78
	crowdsale*	42	14	50	25	5.77
	continuous_sale*	36	8	50	25	3.88
VerX_overview	escrow	60	11	65	33	6.36
	crowdsale	41	9	65	33	6.35
safe_remote_purchase	safe_remote_purchase	71	29	-	-	16.84
serenuscoin	serenuscoin	103	4	-	-	6.40
Uniswap V1	Uniswap†	398	115	105	45	112.81

examples without Vyper implementations from Solidity to Vyper.

Table 5.1 shows the examples; while many consist of a single contract, several either consist of multiple collaborating contracts or of a single contract interacting with external contracts via interfaces. We include several examples of complex code used in practice, e.g., ERC tokens, the first version of the Uniswap contract (the largest decentralized exchange and fourth-largest cryptocurrency exchange overall), and an application used to implement the Genesis Vision ICO, which raised 2.8 million USD in 2017. Most contracts were verified in their entirety; for the ICO, we made some small simplifications (in particular, we cut out two option tokens that behaved exactly like a third one and so added nothing of interest); for the Mana and TheDAO contracts, we focused on specific parts demonstrating inter-contract invariants and a re-entrancy bug, respectively.

5.7.2. Verified Properties

We now describe the functionality, verified properties, and used specification constructs for our examples; if no specific properties are stated, we verified a full functional specification.

ERC20, auction and auction_token: We have verified an extended version of the auction contract from Fig. 5.1 and proved all properties mentioned throughout this chapter. We have also verified the widely-used standard Vyper ERC20 implementation, which is a more complex version of the token contract in Fig. 5.2, by declaring a token resource and annotating all functions with the resource operations they perform. We also used segment constraints to specify when the contract triggers *events*, which are a means for the contract to log which transactions have happened in a way that is visible outside the blockchain, and which can easily be specified using segment constraints.

Finally, we have verified a variant of the auction that deals in custom tokens instead of wei against an ERC1363 interface [151] (see below) annotated with resource-based specifications.

[151]: Minacori (2020), 'EIP-1363: ERC-1363 Payable Token'

TheDAO: We extracted the buggy part of the DAO contract that led to the loss of ca. 50 million USD [94]. Our implementation declares a derived resource for Ether by default (i.e., it assumes that Ether sent to a contract should still conceptually belong to the sender unless otherwise specified). As a result, when the contract tries to send Ether to an address, an error is reported by default, since our resource model requires the user to justify this action by showing that Ether is only sent to its rightful owner. Since this is not always the case, the contract will be rejected.

[94]: Güçlütürk (2018), *The DAO Hack Explained: Unfortunate Take-off of Smart Contracts*

ICO: We verified four contracts that implement the Initial Coin Offering (ICO) for Genesis Vision. The ICO progresses in stages, first selling options, then starting the ICO for option holders, and subsequently for the public. Verification required all specification constructs we have presented, e.g., function constraints to describe guarantees made by locks, transitive segment constraints to preserve information across calls (e.g., that the main token, once unfrozen, will never be frozen again), and resource specifications modeling the option token and main token. We used trust to allow that an administrator can freely access other's tokens, which our technique normally rules out. Importantly, we required proving multiple inter-contract invariants to coordinate the four contracts that implement the ICO, e.g., to prove that the main token will be frozen in its first stages.

Some (inter-contract) properties of this example have also been verified in VerX [174]. Notably however, VerX requires the code of all involved contracts at once and does not allow using interface abstractions. In contrast, we use interfaces annotated with specifications to verify each contract modularly. Additionally, while we prove every property proved by VerX, we also proved additional properties (e.g., all standard resource properties such as non-duplicability and ownership, and that the resource operations the contract performs are the expected ones).

[174]: Permenev et al. (2020), 'VerX: Safety Verification of Smart Contracts'

Uniswap V1: Uniswap is a popular application that consists of many different exchanges, which together allow clients to exchange different tokens against each other, using Ether as an intermediary. A single exchange is responsible for a single token and, if it wants to buy other tokens, contacts the respective exchange contracts for those other tokens. We declared the desired resource-effects for each function and proved the exchange contract correct w.r.t. them. Again, we did so modularly, using a standard ERC20 interface for its token contract and another interface for other exchanges.

VerX overview: We verified the crowdsale application (consisting of two contracts) from the VerX paper, which again included an inter-contract invariant that we verified *modularly* using interfaces and privacy constraints. Additionally, since one of the involved contracts implements a state machine, we used transitive segment constraints to define valid transitions between states (e.g., once the contract is in the "refund" state, it remains in this state).

ERC1363: ERC1363 is a new token standard [151] that combines into one transaction what ERC20 does in several. Fig. 5.20 illustrates how this contract intentionally uses re-entrancy in a way that is not ECF and thus cannot be verified using approaches such as VerX.

[151]: Minacori (2020), 'EIP-1363: ERC-1363 Payable Token'

ERC721: ERC721 is a more complex token standard than ERC20. We declared that it implements a token resource whose tokens have identifiers (non-fungible tokens, NFTs), and specified its functions in terms of token transfers and exchanges. Like ERC20, this required using offers and exchanges, but in addition, it also required using trust, since ERC721 allows users to name other users as "operators" who can act on their behalf.

Serenuscoin: Here we use segment constraints prove both access control properties (only the owner may change the factory) and that the correct events are triggered under the right circumstances.

Mana: We verified a simplified version of the Mana application from the VerX paper, where we focused on the parts necessary to show inter-contract invariants between the three collaborating contracts that were also verified (non-modularly) by VerX. Some of these are not single-state invariants but two-state inter-contract transitive segment constraints; one example is that once the token contract's owner has been set to be the continuous_sale contract, its owner will never change again.

Safe remote purchase: This smart contract sells a good to an arbitrary buyer and holds the buyer's funds in escrow until they acknowledge that they have received the good. The contract gives both parties an incentive not to block the other party from receiving funds by holding a deposit from each of them. We use a derived resource for wei (which, as we stated above, our tool declares by default) to model the fact both buyer and seller conceptually own their deposits until the transaction is finalized, at which point the buyer's wei-titles are exchanged for the good, and the deposits can be paid back.

Conclusion: Our evaluation shows that our specification constructs allow specifying and verifying a wide variety of different properties for real-world contracts. In particular, we can modularly prove inter-contract properties, we can model the resources and resource transactions of different, complex contracts using our resource system (and find typical errors by default), and we can give guarantees for functional correctness and access control even in the presence of unbounded re-entrancy, which allows us to support contracts that employ re-entrancy by design.

5.7.3. Performance and Annotation Overhead

Table 5.1 shows the total lines of code of each contract (excluding comments and whitespace, including specification) as well as the lines of annotations we require, and the lines of code and specifications of all interfaces required to verify each contract, as well as the verification time required by 2Vyper. Times were measured by averaging over ten runs, running on a warmed-up JVM.

On our test system (a 12-core Ryzen 3900X with 32GB RAM running Ubuntu 20.04), most contracts can be automatically verified in 5-25 seconds; the two contracts with the longest verification time, both of which are from complex real-world applications, take between 1.5 and 2 minutes. Considering the strong guarantees afforded by our methodology and tool, we believe even the longest of these times is quite acceptable in practice.

The number of lines required for specifications is less than the number of lines of actual code for every contract. This comparatively modest specification overhead is partly due to our domain-specific resource specifications that allow users to express complex properties in a concise way, and partly due to the design of Vyper, which simplifies verification. Overall, considering the potential financial losses resulting from incorrect smart contracts, writing this amount of specification in exchange for strong functional correctness guarantees is clearly worthwhile.

In conclusion, our technique enables concisely specifying complex correctness properties of (collaborating) contracts, while allowing for modular verification that can be automated efficiently.

5.8. Related work

Much recent work has focused on finding problems in smart contracts and proving their absence. Atzei et al. [13] and Luu et al. [141] list different kinds of attacks and problems specific to smart contracts. A number of tools have been built to automatically find such problems (e.g., resulting from re-entrancy) using either syntactic patterns [120, 212, 213], bounded symbolic execution [5, 141, 155, 167] or data flow analyses [79]. However, most of these tools are unsound by design and can miss errors in real contracts [79, 212, 213]. Additionally, none of these tools allow proving custom functional properties.

Recent work has studied the difference between harmless re-entrant executions and re-entrancy vulnerabilities [44]. Grossman et al. [92] have introduced the notion of effectively callback free objects, for which re-entrancy does not introduce any behaviors that are not present in executions without re-entrancy. They provide an algorithm for dynamically checking for ECF-violations and study the decidability of statically proving that a contract is ECF. More recently, Albert et al. [3] show a static analysis for deciding ECF based on commutativity and projection.

[13]: Atzei et al. (2017), 'A Survey of Attacks on Ethereum Smart Contracts (SoK)'

[141]: Luu et al. (2016), 'Making Smart Contracts Smarter'

[120]: Lai et al. (2020), 'Static Analysis of Integer Overflow of Smart Contracts in Ethereum'

[212]: Tikhomirov et al. (2018), 'SmartCheck: Static Analysis of Ethereum Smart Contracts'

[213]: Tsankov et al. (2018), 'Securify: Practical Security Analysis of Smart Contracts'

[5]: Alt et al. (2018), 'SMT-Based Verification of Solidity Smart Contracts'

[141]: Luu et al. (2016), 'Making Smart Contracts Smarter'

[155]: Mossberg et al. (2019), 'Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts'

[167]: Nikolic et al. (2018), 'Finding The Greedy, Prodigal, and Suicidal Contracts at Scale'

[79]: Feist et al. (2019), 'Slither: a static analysis framework for smart contracts'

[79]: Feist et al. (2019), 'Slither: a static analysis framework for smart contracts'

[212]: Tikhomirov et al. (2018), 'SmartCheck: Static Analysis of Ethereum Smart Contracts'

[213]: Tsankov et al. (2018), 'Securify: Practical Security Analysis of Smart Contracts'

[44]: Cao et al. (2020), 'Reentrancy? Yes. Reentrancy Bug? No'

[92]: Grossman et al. (2018), 'Online detection of effectively callback free objects with applications to smart contracts'

[3]: Albert et al. (2020), 'Taming callbacks for smart contract modularity'

As already mentioned in the introduction of this chapter, existing verification tools either forbid or limit re-entrancy completely, or are simply very imprecise in the presence of re-entrancy. One exception, SmartPulse [210], which was published concurrently with the work shown in this chapter, uses model checking to automatically check liveness properties of smart contracts, and can do so using different attacker models, including one that permits a single re-entrant call, and one that permits arbitrarily many; that last setting is equivalent to our setting.

A number of tools aim to achieve verification of custom functional properties for Ethereum contracts, either on the level of the Solidity language [96, 113, 225] or on the level of EVM bytecode [103] - to the best of our knowledge, 2Vyper is the first verifier specifically aimed at Vyper. EVM-based verification is not specific to any source language and does not rely on the correctness of the compiler; however, specifications tend to be much more complex on the EVM-level, where high-level abstractions of the source language are lost. Verification tools are either based on SMT solvers [96, 174], model checking [146], matching logic [103], CHC solving [113], interactive theorem provers [104], which offer different levels of automation and expressiveness, or a mix of different techniques [200, 225]. Existing verification tools that offer dedicated, higher level specification languages (e.g., [96]) typically support single-state contract invariants, but offer no special support for reasoning in the presence of arbitrary re-entrancy beyond that, resulting in imprecision. VerX [174] and VeriSolid [146] can express temporal properties, which subsume ordinary history constraints; however, VerX explicitly only targets contracts that are ECF, and VeriSolid prevents all re-entrancy by generating code that uses locks throughout. No existing Ethereum verifiers support resource-based specifications.

To our knowledge, the only tools able to prove user-defined inter-contract properties are VerX and VeriSolid. VerX requires the source code of all involved contracts and is therefore not contract-modular, unlike our approach. VeriSolid uses model checking to prove temporal properties on a higher-level model of the contracts and their interactions, and generates correct-by-design Solidity code from this model [166]. Unlike our approach, it does not allow reasoning directly about the code of an existing contract (though contracts can be imported into the model).

Researchers have proposed new smart contract languages that aim to simplify verification and/or make it easier to write correct code [7, 35, 47, 196]. In particular, the Move language [35] offers resources on the programming language level. Unlike our resources, these resources are stateful (in fact, *all* state in Move is stored in resources) and do not have a one-to-one correspondence to physical goods or currency: For example, receiving n coins in Move means adding n to the value stored in one's existing single coin resource, since every address can have at most one resource of every kind. A linear type system ensures that resources are not duplicated in third party code, but the module that defines a resource can modify resource state in arbitrary ways. As a result, incorrect module implementations in Move can violate the properties guaranteed by our resource system (e.g., that resources cannot be taken away from their owners); however, Move's system allows users to manually implement resource models more complex than ours. An SMT-based verifier for custom properties of Move programs exists [234] but currently offers no special support for specifying resource transfers. While it does offer

[210]: Stephens et al. (2021), 'SmartPulse: Automated Checking of Temporal Properties in Smart Contracts'

[96]: Hajdu et al. (2019), 'solc-verify: A Modular Verifier for Solidity Smart Contracts'
[113]: Kalra et al. (2018), 'ZEUS: Analyzing Safety of Smart Contracts'
[225]: Wesley et al. (2021), 'Compositional Verification of Smart Contracts Through Communication Abstraction'
[103]: Hildenbrandt et al. (2018), 'KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine'
[96]: Hajdu et al. (2019), 'solc-verify: A Modular Verifier for Solidity Smart Contracts'
[174]: Permenev et al. (2020), 'VerX: Safety Verification of Smart Contracts'
[146]: Mavridou et al. (2019), 'VeriSolid: Correct-by-Design Smart Contracts for Ethereum'
[103]: Hildenbrandt et al. (2018), 'KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine'
[113]: Kalra et al. (2018), 'ZEUS: Analyzing Safety of Smart Contracts'
[104]: Hirai (2017), 'Defining the Ethereum Virtual Machine for Interactive Theorem Provers'
[200]: Smaragdakis et al. (2021), 'Symbolic value-flow static analysis: deep, precise, complete modeling of Ethereum smart contracts'
[225]: Wesley et al. (2021), 'Compositional Verification of Smart Contracts Through Communication Abstraction'
[96]: Hajdu et al. (2019), 'solc-verify: A Modular Verifier for Solidity Smart Contracts'
[174]: Permenev et al. (2020), 'VerX: Safety Verification of Smart Contracts'
[146]: Mavridou et al. (2019), 'VeriSolid: Correct-by-Design Smart Contracts for Ethereum'
[166]: Nelaturu et al. (2020), 'Verified Development and Deployment of Multiple Interacting Smart Contracts with VeriSolid'

[7]: Andersen et al. (2006), 'Compositional specification of commercial contracts'
[35]: Blackshear et al. (2019), *Move: A language with programmable resources*
[47]: Coblenz (2017), 'Obsidian: a safer blockchain programming language'
[196]: Sergey et al. (2019), 'Safer smart contract programming with Scilla'
[35]: Blackshear et al. (2019), *Move: A language with programmable resources*
[234]: Zhong et al. (2020), 'The Move Prover'

so-called update invariants, which are similar to our transitive segment constraints in that they are also two-state assertions that allow users to specify properties of valid state updates, these cannot be used to reason about possible changes of outside calls, since they are not required to be reflexive and transitive [61], and the Move languages does not allow for re-entrant calls in the first place.

Contract Specification Language (CSL) [7] is a declarative language targeting the speficition of contracts in typical smart contract domain like finance, and sufficiently simple to be amenable to static analysis [101]. It allows specification of contracts in terms of resource transfers, and is, in that sense, similar to our resource-based specification. That is, while CSL uses transfers to define contracts themselves, our resource-based specifications can be seen as a way of specifying contract implementations written in a more complex language like Solidity or Vyper in such terms.

Recent work by different researchers also proposes analyzing smart contracts in terms of *fairness* [101, 139], which, instead of directly specifying and verifying functional properties, focuses on defining limits on the utility any client can gain from one or several contracts; exploits then lead to negative utility for honest clients and vastly positive utility for attackers.

To the best of our knowledge, there are three existing approaches for reasoning about (object-oriented) programs in the presence of unverified code. First, Drossopoulou t al. [64] have introduced *holistic* specifications, which (unlike traditional ones) express *necessary* conditions for an effect to happen, in a setting with arbitrary re-entrancy. They can express e.g., that if a user's token balance decreases, then they either asked to transfer tokens themselves, or another user with a sufficient allowance must have done so. While this kind of property is similar to ones ensured by our resource system, it is not built-in and must be specified manually. Additionally, holistic specifications do not provide support for reasoning about the post-state of calls with arbitrary re-entrancy, and the required (non-standard) reasoning has not been automated, whereas the proof obligations generated by our approach can be checked and automated using standard techniques.

Second, software architectures based on object capabilities [149] and object capability patterns [150] can be used to encapsulate object state so that properties can be maintained even in an unverified environment. The central idea of object capabilities is to withhold the reference to an encapsulated object from unauthorized third parties, and thereby control who may invoke operations on the object. It is therefore crucial that third parties cannot forge capabilities and thereby obtain unintended access to the encapsulated object. However, since this is not the case in typical smart contract languages (contract addresses are not opaque and can be obtained in various ways, not only by receiving them as an intended capability from another contract), the conditions required for capability-based reasoning are not satisfied in this setting.

Third, Agten et al. [1] apply separation logic in a context with unverified code by using runtime checks at the boundary between verified and unverified code to ensure that the unverified code has not modified memory it was not permitted to modify. In contrast, our work relies on language encapsulation to ensure this property and therefore does

[61]: Dill et al. (2022), 'Fast and Reliable Formal Verification of Smart Contracts with the Move Prover'

[7]: Andersen et al. (2006), 'Compositional specification of commercial contracts'

[101]: Henglein et al. (2020), 'A Formally Verified Static Analysis Framework for Compositional Contracts'

[101]: Henglein et al. (2020), 'A Formally Verified Static Analysis Framework for Compositional Contracts'

[139]: Liu et al. (2020), 'Towards automated verification of smart contract fairness'

[64]: Drossopoulou et al. (2020), 'Holistic Specifications for Robust Programs'

[149]: Miller (2006), 'Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control'

[150]: Miller et al. (2000), 'Capability-Based Financial Instruments'

[1]: Agten et al. (2015), 'Sound Modular Verification of C Code Executing in an Unverified Context'

not require runtime checks, which are especially undesirable in a smart contract setting due to the associated gas cost.

5.9. Conclusion

We have presented a novel approach for specification and verification of Ethereum smart contracts. Our methodology exploits the features of Ethereum, such as strong encapsulation, to provide guarantees even in the presence of arbitrary re-entrancy, and provides domain-specific specification constructs for resources that make specification both more intuitive and less error-prone. Our evaluation shows that out methology can be implemented efficiently and is capable of expressing and proving complex functional specifications for real-world contracts.

Conclusions and Future Work 6.

6.1. Conclusion

In this thesis, we have proposed several novel ways of proving security-related program properties, or of extending existing techniques to previously-unsupported languages and environment.

We have introduced modular product programs, a product construction that allows proving arbitrary hypersafety properties for imperative languages. Modular product programs are the first general product construction that can be constructed fully automatically *and* allows for modular specification and verification of such properties, whereas previously existing products either required user input, did not enable modular verification, or restricted program control flow. Furthermore, we have shown how modular product programs can encode information flow specifications, which can express non-interference properties (i.e., confidentiality and integrity properties) in a modular way, enable precise specifications including value-dependent sensitivity, declassification, and termination-sensitive non-interference, and allow for proofs that do not require knowledge about the underlying product encoding.

Furthermore, we have shown how the presented product construction (and other similar ones) can be applied to more complex source languages when used in the common framework of IVL-based verifiers, by applying the product construction on the level of the simple IVL program. This approach makes it possible to extend existing IVL-based verifiers for hyperproperty verification with comparatively low effort, and thus makes product constructions substantially more useful for the verification of programs written in mainstream languages for which standard verifiers already exist. Crucially, we are the first to examine when such an approach is sound; we show that this is not always the case, but provide a simple criterion on the IVL encoding that can be used to identify parts of the encoding that may have to be adapted. Additionally, we show that this system can also be used to prove different kinds of non-interference properties for concurrent source programs, without requiring a product construction for concurrent programs, which is important since modular verification of information flow security for concurrent programs is still very rarely supported by existing automated verification tools.

Additionally, we considered two mainstream programming languages that fall outside of the spectrum of C- and Java-like languages commonly supported by existing verification tools, and provided verification techniques that enable security property verification for these and similar ones.

First, we provided a verification technique for the dynamic Python language. Here, we enriched existing verification techniques for object-oriented programs with additional checks and information (e.g., performing type checking partly on the verifier level), as well as additional specification constructs, while slightly simplifying the used language model; as a result, we do not support all Python language features, but can specify and verify typical client code in a modular way. Crucially, our approach can be combined with existing techniques for the modular verification of security-related properties like progress properties and input-output behavior, as well as our own technique for non-interference verification; thus, we enable the modular verification of these properties for typical Python code for the first time.

Second, we considered the domain of Ethereum smart contracts, which are prime targets for attackers and therefore must be shown to be free of potential exploits. Here, we introduce novel specification constructs that allow for the modular specification and verification of smart contracts in an adversarial environment, where many standard verification techniques do not apply. The resulting technique is the first one that is modular and does not impose limits on re-entrancy, and the first one that allows for the modular verification of sets of collaborating contracts in this setting. In particular, we also allow specifying contract behavior on the level of resources and resource transfers, which are the main purpose of most real smart contracts. As a result, standard properties can be checked by default, and custom contract behavior can be specified on the higher level of abstraction that most programmers and users of smart contracts tend to use when thinking about contracts (instead of the low-level of the contract implementation itself), thus reducing the likelihood of errors in specifications.

While we focused on Python programs and Ethereum contracts specifically, (parts of) our techniques transfer to other languages: For example, the idea of using optimistic static typing by default but letting users opt out and delegate parts of the type checking to the verifier applies to other dynamic languages, and our technique for modular verification in an adversarial environment applies to other languages that provide strong encapsulation guarantees, like for example some actor-based languages.

6.2. Future Work

The presented work can be extended in a number of ways not discussed before. Obvious extensions are soundness proofs for our verification techniques for Python programs and smart contracts, as well as various improvements in the developed tools. Here, we make four suggestions to go beyond these.

6.2.1. General Relational Property Verification

While modular product programs compose a program with copies of itself, the core idea behind the product construction can be extended to allow the composition of two (or more) different programs, which would enable the verification of general relational properties like program equivalence.

As a basic idea, clearly two different programs can be combined into a product program by simply executing each on its respective renamed version of the program variables, and under the condition that the respective activation variable is true. Crucially, however, control structures in the different programs can be combined: For example, if each program contains a loop, the loops can be combined into one in the product program, which uses different loop conditions and different loop bodies for the two executions; similarly, two calls to different (but potentially related) methods can be combined as well, enabling the use of relational specifications.

Such a combination requires information on which parts of programs to combine in the product, which can be expressed for example in the form of biprograms [17], though this information will either have to be provided by a user or inferred using a separate technique. However, there are many applications for verifying relational properties (in particular, equivalence modulo some parts of the state space) of pairs of programs that have a similar structure by construction, so that alignment information is trivial to extract. One example is proving a slice of a program equivalent to the original program for relevant parts of the state; here, one program is typically a copy of the other program with some parts being removed. The same applies when removing ghost code from a program (which should not affect the the non-ghost part of the program state), or when adding an instrumentation to a program.

[17]: Banerjee et al. (2016), 'Relational Logic with Framing and Hypotheses'

When manually proving two programs with different structures equivalent, one could, as a first step, use proven equivalence-preserving program transformations (e.g. partially unrolling a loop, or transforming a loop into recursive calls) to partially align program structure, and subsequently use a modular product construction to prove equivalence using relational specifications.

6.2.2. Extended Python Object Model

As explained in Sec. 2.2.4 in Chapter 2, our verification technique for Python uses a simplified model of attribute lookups and modifications in order to enable modular verification with a reasonable annotation overhead and acceptable verification performance; we then forbid overriding and directly calling so-called magic methods and attributes that could expose Python's actual, more complex object model.

Obviously, this comes at the cost of not being able to verify code that makes use of the actual internal model of Python attributes. Currently, we only support manually modeling the resulting behavior of such code on the level of the simplified object model by building said behavior into the verifier, meaning that some high-level behavior of the code is simply assumed by the verifier, but not actually verified to correspond to the actual implementation.

One way of approaching this problem is to allow *selectively* modeling Python's actual behavior when verifying specific classes or methods, and subsequently proving that the actual behavior of the code that uses low-level attribute behavior can be soundly approximated by some model of its behavior on the level of the simplified object model. Such a proof would be a kind of refinement proof: All attribute accesses that are valid on the level of the simplified object model must also be valid (i.e., execute without errors) in the actual low-level code, and must return the same results.

Since this is essentially a relational property (the program with the simplified object model behaves like a program with the actual, more complex object model) of two closely-related programs, one could use modular product programs to prove this equivalence. In combination with a proof that shows that for ordinary attribute lookups, our model is accurate if all relevant magic methods are not overridden, such a proof would remove any uncertainty about the soundness of working with a simplified object model when verifying other code.

6.2.3. Quantitative Information Flow Verification

As we showed in Sec. 3.5.4 in Chapter 3, modular product programs can easily be used to implement a simple but powerful model of declassification. In particular, the declassified information can be specified in arbitrarily precise terms: For example, instead of declassifying an entire high integer value h, one can also declassify only its third least significant bit by declassifying $h/4 \mod 2$, or e.g. whether the information if its digit sum is equal to three, by declassifying the expression $dsv(h) = 3$. Information flow specifications subsequently allow passing this detailed sensitivity information throughout the rest of the program.

This precise modeling of sensitivity, combined with a permission logic, can easily be used to soundly (but very imprecisely) verify *quantitative* information flow properties like “statement s leaks no more than n bits of its high input”, as follows:

1. First, we prove non-interference of the program as usual, but allow declassifying secret values. Once a non-interference proof succeeds, we know that the program leaks high information *only* through its declassification operations.
2. Second, we restrict declassification operations to apply only to boolean values and to not happen in branches under high conditions. This has the result that a single declassification action can leak at most a single bit of information (but possibly less). As an example, if h is a high 32-bit integer with a uniform value distribution, then declassifying $h \mod 2$ leaks exactly one bit of information, but declassifying $h = 25612$ leaks substantially less (in terms of Shannon entropy).
   That is, to leak the entire 32 bits of h, one would have to declassify every single one of its bits, which would require at least 32 declassification actions. However, if one chooses to inefficiently declassify its equality to every single possible value instead (i.e., first declassify $h = 0$, then $h = 1$, etc.), the number of declassification actions could instead be 2^{32}. Crucially, there is no way to leak h entirely using less than 32 declassification operations.
3. Finally, we require a *permission* to perform a single declassification operation. That is, we give a declassification operation the specification $\vDash \{leak(1)\}\texttt{declassify}\ e\{low(e)\}$, where $leak(m)$ represents a permission to leak at most m bits of information, and e is a boolean value. We then give the program the permission $leak(n)$ in its precondition, allowing it to leak at most n bits of high information, and thereby limiting the number of declassification operations the program may perform.

This simple approach to limiting the amount of leaked information is sound, but imprecise: First, as already discussed, declassifying a boolean expression may leak less than a bit; second, declassification under high conditions does not have to be forbidden if the information flow resulting from high branches can be modeled precisely. Interestingly, high branches can even *lower* the amount of information leaked by a declassification action. As an example, the program `if` $(h = 256)$ $\{$`declassify` $h\}$ (where we write `declassify` h as a short hand for declassifying all its bits) leaks substantially less than 32 bits when considering Shannon entropy. On the other hand, however, `if` $(h1)$ $\{$`declassify` $h2\}$ will leak more than just $h2$.

As future work, one could investigate how this approach can be made sufficiently precise to verify realistic properties in real code. In particular, one could first investigate if (even with this crude approach) a smart placement and usage of declassification operations can already avoid some of its pitfalls, e.g. by declassifying $h = 256$ in the previously-shown program, to avoid having to conditionally declassify all bits in h and therefore be able to show that at most one bit is leaked instead. Subsequently, one could then extend the approach to more precisely calculate how much information is leaked by a particular declassification operation in cases where the basic approach is insufficient.

6.2.4. Verification in the Presence of Untrusted Code

In Chapter 5, we presented a verification technique for smart contracts that addresses the problem that such contracts generally run in the presence of untrusted code, ruling out (or requiring adaptations to) the use of traditional verification techniques like separation logic. However, the problem of executing verified code in the presence of other, untrusted code is by no means limited to smart contracts: Many applications consist of several components, and often verification is restricted to some subset of those components. For example, in a large application, one may only verify a specific, security-critical part of the code, simply because verifying the entire application would be too resource intensive. In other cases, applications that have a plugin infrastructure are built to be extensible with arbitrary outside code as long as said code implements a specific interface; thus, when verifying (parts of) the main component or a specific plugin, other code that will be running as part of the application is simply unknown.

Our solution for smart contracts also applies to some actor-based languages, but crucially depends on the fact that object data cannot be modified by outside code, which is not the case by default in most object-oriented languages. For individual programs, however, it may well be the case that a class' internal data is protected because its implementation ensures that no references to its internal state are ever leaked to the untrusted outside world, and that, conversely, the object itself does not capture any state to which pre-existing outside references may exist.

Thus, to make our technique applicable to ordinary object-oriented languages, one could supplement it with a verification technique for proving that internal object state is never modifiable from the outside, by ensuring that no references that would allow such modifications ever become reachable from outside state. The exact requirements of such a technique would of course be language-specific, and depend on the kind of guarantees a programming language provides: For example, in this setting, it is crucial to know if “private” object data can be accessed by outside objects in some way or not (e.g., using reflection), or if pointers can be forged, e.g. by performing pointer arithmetic (which would make such an approach entirely impossible).

Bibliography

[1] Pieter Agten, Bart Jacobs, and Frank Piessens. 'Sound Modular Verification of C Code Executing in an Unverified Context'. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. Ed. by Sriram K. Rajamani and David Walker. ACM, 2015, pp. 581–594. DOI: 10.1145/2676726.2676972 (cited on page 203).

[2] Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Pierre-Yves Strub. 'A relational logic for higher-order programs'. In: *PACMPL* 1.ICFP (2017), 21:1–21:29 (cited on page 117).

[3] Elvira Albert, Shelly Grossman, Noam Rinetzky, Clara Rodríguez-Núñez, Albert Rubio, and Mooly Sagiv. 'Taming callbacks for smart contract modularity'. In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 209:1–209:30. DOI: 10.1145/3428277 (cited on pages 165, 201).

[4] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 'Verifying Constant-Time Implementations'. In: *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*. Ed. by Thorsten Holz and Stefan Savage. USENIX Association, 2016, pp. 53–70 (cited on page 156).

[5] Leonardo Alt and Christian Reitwießner. 'SMT-Based Verification of Solidity Smart Contracts'. In: *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 11247. Lecture Notes in Computer Science. Springer, 2018, pp. 376–388. DOI: 10.1007/978-3-030-03427-6_28 (cited on page 201).

[6] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 'Syntax-guided synthesis'. In: *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. IEEE, 2013, pp. 1–8 (cited on page 120).

[7] Jesper Andersen, Ebbe Elsborg, Fritz Henglein, Jakob Grue Simonsen, and Christian Stefansen. 'Compositional specification of commercial contracts'. In: *Int. J. Softw. Tools Technol. Transf.* 8.6 (2006), pp. 485–516. DOI: 10.1007/s10009-006-0010-1 (cited on pages 202, 203).

[8] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. 'Decomposition instead of self-composition for proving the absence of timing channels'. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. 2017, pp. 362–375 (cited on pages 110, 111, 119).

[9] Krzysztof R. Apt. 'Ten Years of Hoare's Logic: A Survey - Part 1'. In: *ACM Trans. Program. Lang. Syst.* 3.4 (1981), pp. 431–483. DOI: 10.1145/357146.357150 (cited on page 11).

[10] Sivakumar Arumugam. *Serenuscoin contract*. Accessed on 2021-04-16. 2019. URL: https://github.com/serenuscoin/contracts (cited on page 197).

[11] Vytautas Astrauskas. 'Input-output verification in Viper'. MA thesis. ETH Zürich, 2016 (cited on pages 59, 61).

[12] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 'Leveraging Rust types for modular specification and verification'. In: *Proc. ACM Program. Lang.* 3.OOPSLA (2019), 147:1–147:30. DOI: 10.1145/3360573 (cited on pages 76, 123).

[13] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 'A Survey of Attacks on Ethereum Smart Contracts (SoK)'. In: *Principles of Security and Trust - 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. Ed. by Matteo Maffei and Mark Ryan. Vol. 10204. Lecture Notes in Computer Science. Springer, 2017, pp. 164–186. DOI: 10.1007/978-3-662-54455-6_8 (cited on page 201).

[14] Anindya Banerjee, Ramana Nagasamudram, David A. Naumann, and Mohammad Nikouei. 'A Relational Program Logic with Data Abstraction and Dynamic Framing'. In: *ACM Trans. Program. Lang. Syst.* (July 2022). Just Accepted. DOI: 10.1145/3551497 (cited on page 117).

[15] Anindya Banerjee and David A. Naumann. 'Secure Information Flow and Pointer Confinement in a Java-like Language'. In: *15th IEEE Computer Security Foundations Workshop (CSFW-15 2002), 24-26 June 2002, Cape Breton, Nova Scotia, Canada*. 2002, p. 253. DOI: 10.1109/CSFW.2002.1021820 (cited on page 91).

[16] Anindya Banerjee and David A. Naumann. 'Secure Information Flow and Pointer Confinement in a Java-like Language'. In: *15th IEEE Computer Security Foundations Workshop (CSFW-15 2002), 24-26 June 2002, Cape Breton, Nova Scotia, Canada*. 2002, p. 253 (cited on pages 110, 111).

[17] Anindya Banerjee, David A. Naumann, and Mohammad Nikouei. 'Relational Logic with Framing and Hypotheses'. In: *36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2016, December 13-15, 2016, Chennai, India*. 2016, 11:1–11:16 (cited on pages 117, 207).

[18] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 'Boogie: A Modular Reusable Verifier for Object-Oriented Programs'. In: *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*. Ed. by Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever. Vol. 4111. Lecture Notes in Computer Science. Springer, 2005, pp. 364–387. DOI: 10.1007/11804192_17 (cited on pages 4, 16, 67, 76, 116, 123).

[19] Michael Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. 'Verification of Object-Oriented Programs with Invariants'. In: *J. Object Technol.* 3.6 (2004), pp. 27–56. DOI: 10.5381/jot.2004.3.6.a2 (cited on page 165).

[20] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. 'The Coq proof assistant reference manual: Version 6.1'. PhD thesis. Inria, 1997 (cited on page 3).

[21] David Barrera, Laurent Chuat, Adrian Perrig, Raphael M. Reischuk, and Pawel Szalachowski. 'The SCION Internet Architecture'. In: *Commun. ACM* 60.6 (May 2017), pp. 56–65 (cited on page 73).

[22] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. 'CVC4'. In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 171–177. DOI: 10.1007/978-3-642-22110-1_14 (cited on pages 3, 16).

[23] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 'Relational Verification Using Product Programs'. In: *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*. Ed. by Michael J. Butler and Wolfram Schulte. Vol. 6664. Lecture Notes in Computer Science. Springer, 2011, pp. 200–214. DOI: 10.1007/978-3-642-21437-0_17 (cited on pages 3, 5, 79, 110, 111, 117, 123, 158).

[24] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 'Beyond 2-Safety: Asymmetric Product Programs for Relational Program Verification'. In: *Logical Foundations of Computer Science, International Symposium, LFCS 2013, San Diego, CA, USA, January 6-8, 2013. Proceedings*. Ed. by Sergei N. Artëmov and Anil Nerode. Vol. 7734. Lecture Notes in Computer Science. Springer, 2013, pp. 29–43. DOI: 10.1007/978-3-642-35722-0_3 (cited on pages 5, 79, 117, 118, 123).

[25] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. 'Secure information flow by self-composition'. In: *Mathematical Structures in Computer Science* 21.6 (2011), pp. 1207–1252. DOI: 10.1017/S0960129511000193 (cited on pages 3, 79, 101, 117, 123, 158).

[26] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 'Formal certification of code-based cryptographic proofs'. In: *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*. 2009, pp. 90–101 (cited on page 117).

[27] Gilles Barthe, David Pichardie, and Tamara Rezk. 'A certified lightweight non-interference Java bytecode verifier'. In: *Math. Struct. Comput. Sci.* 23.5 (2013), pp. 1032–1081. DOI: 10.1017/S0960129512000850 (cited on page 91).

[28] Nils Becker, Peter Müller, and Alexander J. Summers. 'The Axiom Profiler: Understanding and Debugging SMT Quantifier Instantiations'. In: *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I*. Ed. by Tomás Vojnar and Lijun Zhang. Vol. 11427. Lecture Notes in Computer Science. Springer, 2019, pp. 99–116. DOI: 10.1007/978-3-030-17462-0_6 (cited on page 77).

[29] Bernhard Beckert, Daniel Bruns, Vladimir Klebanov, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. 'Information Flow in Object-Oriented Software'. In: *Logic-Based Program Synthesis and Transformation, 23rd International Symposium, LOPSTR 2013, Madrid, Spain, September 18-19, 2013, Revised Selected Papers*. Ed. by Gopal Gupta and Ricardo Peña. Vol. 8901. Lecture Notes in Computer Science. Springer, 2013, pp. 19–37. DOI: 10.1007/978-3-319-14125-1_2 (cited on page 91).

[30] Nick Benton. 'Simple relational correctness proofs for static analyses and program transformations'. In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*. 2004, pp. 14–25 (cited on pages 3, 79, 117).

[31] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 'Smallfoot: Modular Automatic Assertion Checking with Separation Logic'. In: *Formal Methods for Components and Objects*. Ed. by Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 115–137 (cited on page 16).

[32] Lennart Beringer. 'Relational Decomposition'. In: *Interactive Theorem Proving - Second International Conference, ITP 2011, Berg en Dal, The Netherlands, August 22-25, 2011. Proceedings*. Ed. by Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk. Vol. 6898. Lecture Notes in Computer Science. Springer, 2011, pp. 39–54. DOI: 10.1007/978-3-642-22863-6_6 (cited on page 117).

[33] Lennart Beringer and Martin Hofmann. 'Secure information flow and program logics'. In: *20th IEEE Computer Security Foundations Symposium, CSF 2007, 6-8 July 2007, Venice, Italy*. IEEE Computer Society, 2007, pp. 233–248. DOI: 10.1109/CSF.2007.30 (cited on page 91).

[34] Adam Betts, Nathan Chong, Alastair F. Donaldson, Shaz Qadeer, and Paul Thomson. 'GPUVerify: a verifier for GPU kernels'. In: *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*. Ed. by Gary T. Leavens and Matthew B. Dwyer. ACM, 2012, pp. 113–132. DOI: 10.1145/2384616.2384625 (cited on pages 76, 119, 123).

[35] Sam Blackshear, Evan Cheng, David L Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Dario Russi Rain, Stephane Sezer, et al. *Move: A language with programmable*

resources. 2019. URL: https://developers.libra.org/docs/move-paper (cited on pages 161, 175, 202).

[36] Mathias Blarer. 'Static Analysis of GPU Kernel Performance Hyperproperties'. Bachelor's thesis. ETH Zürich, 2019 (cited on page 120).

[37] Blockchains LLC. *Decentralized Autonomous Organization (DAO) Framework*. Accessed on 2021-04-16. 2016. URL: https://github.com/blockchainsllc/DAO/blob/6967d70e0e11762c1c34830d7ef2b86e62ff868e/DAO.sol (cited on page 197).

[38] Stefan Blom and Marieke Huisman. 'The VerCors Tool for Verification of Concurrent Programs'. In: *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*. Ed. by Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun. Vol. 8442. Lecture Notes in Computer Science. Springer, 2014, pp. 127–131. DOI: 10.1007/978-3-319-06410-9_9 (cited on pages 76, 123, 133).

[39] P. Boström and P. Müller. 'Modular Verification of Finite Blocking in Non-terminating Programs'. In: *European Conference on Object-Oriented Programming (ECOOP)*. Ed. by J. T. Boyland. Vol. 37. LIPIcs. Schloss Dagstuhl, 2015, pp. 639–663 (cited on page 59).

[40] Pontus Boström and Peter Müller. 'Modular Verification of Finite Blocking in Non-terminating Programs'. In: *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*. Ed. by John Tang Boyland. Vol. 37. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, pp. 639–663. DOI: 10.4230/LIPIcs.ECOOP.2015.639 (cited on pages 62, 152).

[41] John Boyland. 'Checking Interference with Fractional Permissions'. In: *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*. Ed. by Radhia Cousot. Vol. 2694. Lecture Notes in Computer Science. Springer, 2003, pp. 55–72. DOI: 10.1007/3-540-44898-5_4 (cited on page 14).

[42] Christian Bräm. 'Verification of Advanced Properties for Real World Vyper Contracts'. MA thesis. ETH Zürich, 2020 (cited on page 196).

[43] Christian Bräm, Marco Eilers, Peter Müller, Robin Sierra, and Alexander J. Summers. 'Rich specifications for Ethereum smart contract verification'. In: *Proc. ACM Program. Lang.* 5.OOPSLA (2021), pp. 1–30. DOI: 10.1145/3485523 (cited on pages 8, 159).

[44] Qinxiang Cao and Zhongye Wang. 'Reentrancy? Yes. Reentrancy Bug? No'. In: *Dependable Software Engineering. Theories, Tools, and Applications - 6th International Symposium, SETTA 2020, Guangzhou, China, November 24-27, 2020, Proceedings*. Ed. by Jun Pang and Lijun Zhang. Vol. 12153. Lecture Notes in Computer Science. Springer, 2020, pp. 17–34. DOI: 10.1007/978-3-030-62822-2_2 (cited on page 201).

[45] Zhifei Chen, Lin Chen, and Baowen Xu. 'Hybrid Information Flow Analysis for Python Bytecode'. In: *11th Web Information System and Application Conference, WISA 2014, Tianjin, China, September 12-14, 2014*. IEEE, 2014, pp. 95–100. DOI: 10.1109/WISA.2014.26 (cited on page 157).

[46] Michael R. Clarkson and Fred B. Schneider. 'Hyperproperties'. In: *Journal of Computer Security* 18.6 (2010), pp. 1157–1210 (cited on pages 3, 80, 117).

[47] Michael J. Coblenz. 'Obsidian: a safer blockchain programming language'. In: *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*. Ed. by Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard. IEEE Computer Society, 2017, pp. 97–99. DOI: 10.1109/ICSE-C.2017.150 (cited on page 202).

[48] Ernie Cohen, Michal Moskal, Wolfram Schulte, and Stephan Tobies. 'Local Verification of Global Invariants in Concurrent Programs'. In: *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*. Ed. by Tayssir Touili, Byron Cook, and Paul B. Jackson. Vol. 6174. Lecture Notes in Computer Science. Springer, 2010, pp. 480–494. DOI: 10.1007/978-3-642-14295-6_42 (cited on pages 76, 123, 136, 137).

[49] Peter Collingbourne, Alastair F. Donaldson, Jeroen Ketema, and Shaz Qadeer. 'Interleaving and Lock-Step Semantics for Analysis and Verification of GPU Kernels'. In: *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. 2013, pp. 270–289. DOI: 10.1007/978-3-642-37036-6_16 (cited on page 119).

[50] Peter Collingbourne, Alastair F. Donaldson, Jeroen Ketema, and Shaz Qadeer. 'Interleaving and Lock-Step Semantics for Analysis and Verification of GPU Kernels'. In: *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. 2013, pp. 270–289. DOI: 10.1007/978-3-642-37036-6_16 (cited on page 150).

[51] David Costanzo and Zhong Shao. 'A Separation Logic for Enforcing Declarative Information Flow Control Policies'. In: *Principles of Security and Trust - Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*. 2014, pp. 179–198 (cited on pages 110, 111, 120).

[52] Patrick Cousot and Radhia Cousot. 'Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints'. In: *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*. Ed. by Robert M. Graham, Michael A. Harrison, and Ravi Sethi. ACM, 1977, pp. 238–252. DOI: 10.1145/512950.512973 (cited on page 120).

[53] Patrick Cousot and Nicolas Halbwachs. 'Automatic Discovery of Linear Restraints Among Variables of a Program'. In: *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*. Ed. by Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski. ACM Press, 1978, pp. 84–96. DOI: 10.1145/512760.512770 (cited on page 120).

[54] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 'Frama-C - A Software Analysis Perspective'. In: *Software Engineering and Formal Methods - 10th International Conference, SEFM 2012, Thessaloniki, Greece, October 1-5, 2012. Proceedings*. Ed. by George Eleftherakis, Mike Hinchey, and Mike Holcombe. Vol. 7504. Lecture Notes in Computer Science. Springer, 2012, pp. 233–247. DOI: 10.1007/978-3-642-33826-7_16 (cited on pages 76, 123).

[55] Thibault Dardinier, Gaurav Parthasarathy, and Peter Müller. 'Verification-Preserving Inlining in Automatic Separation Logic Verifiers (extended version)'. In: *CoRR* abs/2208.10456 (2022). DOI: 10.48550/arXiv.2208.10456 (cited on page 77).

[56] Ádám Darvas, Reiner Hähnle, and David Sands. 'A Theorem Proving Approach to Analysis of Secure Information Flow'. In: *Security in Pervasive Computing, Second International Conference, SPC 2005, Boppard, Germany, April 6-8, 2005, Proceedings*. 2005, pp. 193–209 (cited on pages 110, 111, 120).

[57] Zhenyue Deng and Geoffrey Smith. 'Lenient Array Operations for Practical Secure Information Flow'. In: *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*. 2004, p. 115 (cited on page 119).

[58] David Detlefs, Greg Nelson, and James B. Saxe. 'Simplify: a theorem prover for program checking'. In: *J. ACM* 52.3 (2005), pp. 365–473. DOI: 10.1145/1066100.1066102 (cited on pages 16, 154).

[59] Krishna Kishore Dhara and Gary T. Leavens. 'Forcing Behavioral Subtyping through Specification Inheritance'. In: *18th International Conference on Software Engineering, Berlin, Germany, March 25-29, 1996, Proceedings*. Ed. by H. Dieter Rombach, T. S. E. Maibaum, and Marvin V. Zelkowitz. IEEE Computer Society, 1996, pp. 258–267 (cited on page 39).

[60] Edsger W. Dijkstra. 'Guarded commands, non-determinancy and a calculus for the derivation of programs'. In: *Language Hierarchies and Interfaces, International Summer School, Marktoberdorf, Germany, July 23 - August 2, 1975*. Ed. by Friedrich L. Bauer and Klaus Samelson. Vol. 46. Lecture Notes in Computer Science. Springer, 1975, pp. 111–124. DOI: 10.1007/3-540-07994-7_51 (cited on page 16).

[61] David L. Dill, Wolfgang Grieskamp, Junkil Park, Shaz Qadeer, Meng Xu, and Jingyi Emma Zhong. 'Fast and Reliable Formal Verification of Smart Contracts with the Move Prover'. In: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022,*

Munich, Germany, April 2-7, 2022, Proceedings, Part I. Ed. by Dana Fisman and Grigore Rosu. Vol. 13243. Lecture Notes in Computer Science. Springer, 2022, pp. 183–200. DOI: 10.1007/978-3-030-99524-9_10 (cited on page 203).

[62] Danny Dolev and Andrew Chi-Chih Yao. 'On the security of public key protocols'. In: *IEEE Trans. Inf. Theory* 29.2 (1983), pp. 198–207. DOI: 10.1109/TIT.1983.1056650 (cited on page 75).

[63] Sophia Drossopoulou, Adrian Francalanza, Peter Müller, and Alexander J. Summers. 'A Unified Framework for Verification Techniques for Object Invariants'. In: *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*. Ed. by Jan Vitek. Vol. 5142. Lecture Notes in Computer Science. Springer, 2008, pp. 412–437. DOI: 10.1007/978-3-540-70592-5_18 (cited on pages 168, 169).

[64] Sophia Drossopoulou, James Noble, Julian Mackay, and Susan Eisenbach. 'Holistic Specifications for Robust Programs'. In: *Fundamental Approaches to Software Engineering - 23rd International Conference, FASE 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*. Ed. by Heike Wehrheim and Jordi Cabot. Vol. 12076. Lecture Notes in Computer Science. Springer, 2020, pp. 420–440. DOI: 10.1007/978-3-030-45234-6_21 (cited on page 203).

[65] Douglas D. Dunlop and Victor R. Basili. 'A Comparative Analysis of Functional Correctness'. In: *ACM Comput. Surv.* 14.2 (1982), pp. 229–244. DOI: 10.1145/356876.356881 (cited on pages 2, 11).

[66] Marco Eilers, Severin Meier, and Peter Müller. 'Product Programs in the Wild: Retrofitting Program Verifiers to Check Information Flow Security'. In: *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*. Ed. by Alexandra Silva and K. Rustan M. Leino. Vol. 12759. Lecture Notes in Computer Science. Springer, 2021, pp. 718–741. DOI: 10.1007/978-3-030-81685-8_34 (cited on pages 8, 77, 123).

[67] Marco Eilers and Peter Müller. 'Nagini: A Static Verifier for Python'. In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*. Ed. by Hana Chockler and Georg Weissenbacher. Vol. 10981. Lecture Notes in Computer Science. Springer, 2018, pp. 596–603. DOI: 10.1007/978-3-319-96145-3_33 (cited on page 9).

[68] Marco Eilers, Peter Müller, and Samuel Hitz. 'Modular Product Programs'. In: *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. Ed. by Amal Ahmed. Vol. 10801. Lecture Notes in Computer Science. Springer, 2018, pp. 502–529. DOI: 10.1007/978-3-319-89884-1_18 (cited on pages 8, 79).

[69] Marco Eilers, Peter Müller, and Samuel Hitz. 'Modular Product Programs'. In: *ACM Trans. Program. Lang. Syst.* 42.1 (2020), 3:1–3:37. DOI: 10.1145/3324783 (cited on pages 8, 79).

[70] Dima Elenbogen, Shmuel Katz, and Ofer Strichman. 'Proving mutual termination'. In: *Formal Methods in System Design* 47.2 (2015), pp. 204–229. DOI: 10.1007/s10703-015-0234-3 (cited on page 119).

[71] William Entriken, Dieter Shirley, Jacob Evans, and Nastassia Sachs. 'EIP-721: ERC-721 Non-Fungible Token Standard'. In: *Ethereum Improvement Proposals* 721 (2018) (cited on pages 182, 183).

[72] Gidon Ernst and Toby Murray. 'SecCSL: Security Concurrent Separation Logic'. In: *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*. Ed. by Isil Dillig and Serdar Tasiran. Vol. 11562. Lecture Notes in Computer Science. Springer, 2019, pp. 208–230. DOI: 10.1007/978-3-030-25543-5_13 (cited on pages 3, 5, 76, 125, 136, 145, 149, 154).

[73] Ethereum. *Solidity by example*. Accessed on 2021-04-16. 2021. URL: https://github.com/ethereum/solidity (cited on page 197).

[74] Ethereum. *Solidity documentation*. Accessed on 2020-01-11. 2021. URL: https://solidity.readthedocs.io/ (cited on page 162).

[75] Ethereum. *Vyper documentation*. Accessed on 2020-01-11. 2021. URL: https://vyper.readthedocs.io/ (cited on pages 161, 162).

[76] Ethereum. *Vyper example contracts*. Accessed on 2021-04-16. 2021. URL: https://github.com/vyperlang/vyper/tree/master/examples (cited on page 197).

[77] Manuel Fähndrich, Mike Barnett, and Francesco Logozzo. *Code Contracts*. http://research.microsoft.com/contracts. 2008 (cited on page 64).

[78] Azadeh Farzan and Anthony Vandikas. 'Automated Hypersafety Verification'. In: *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*. Ed. by Isil Dillig and Serdar Tasiran. Vol. 11561. Lecture Notes in Computer Science. Springer, 2019, pp. 200–218. DOI: 10.1007/978-3-030-25540-4_11 (cited on pages 118, 123, 136, 158).

[79] Josselin Feist, Gustavo Grieco, and Alex Groce. 'Slither: a static analysis framework for smart contracts'. In: *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2019, Montreal, QC, Canada, May 27, 2019*. IEEE, 2019, pp. 8–15. DOI: 10.1109/WETSEB.2019.00008 (cited on pages 160, 201).

[80] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. 'Automating regression verification'. In: *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*. Ed. by Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher. ACM, 2014, pp. 349–360. DOI: 10.1145/2642937.2642987 (cited on page 119).

[81] Jean-Christophe Filliâtre and Claude Marché. 'The Why/Krakatoa/Caduceus Platform for Deductive Program Verification'. In: *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*. Ed. by Werner Damm and Holger Hermanns. Vol. 4590. Lecture Notes in Computer

Science. Springer, 2007, pp. 173–177. DOI: 10.1007/978-3-540-73368-3_21 (cited on pages 76, 123).

[82] Jean-Christophe Filliâtre and Andrei Paskevich. 'Why3 - Where Programs Meet Provers'. In: *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. Ed. by Matthias Felleisen and Philippa Gardner. Vol. 7792. Lecture Notes in Computer Science. Springer, 2013, pp. 125–128. DOI: 10.1007/978-3-642-37036-6_8 (cited on pages 4, 16, 76, 117, 123).

[83] Sascha Forster. 'Static Verification of the SCION Router Implementation'. Bachelor's thesis. ETH Zürich, 2018 (cited on pages 73, 77).

[84] Levin Fritz and Jurriaan Hage. 'Cost Versus Precision for Approximate Typing for Python'. In: *PEPM*. 2017, pp. 89–98 (cited on page 76).

[85] Aymeric Fromherz, Abdelraouf Ouadjaout, and Antoine Miné. 'Static Value Analysis of Python Programs by Abstract Interpretation'. In: *NASA Formal Methods - 10th International Symposium, NFM 2018, Newport News, VA, USA, April 17-19, 2018, Proceedings*. Ed. by Aaron Dutle, César A. Muñoz, and Anthony Narkawicz. Vol. 10811. Lecture Notes in Computer Science. Springer, 2018, pp. 185–202. DOI: 10.1007/978-3-319-77935-5_14 (cited on page 76).

[86] Dan Frumin, Robbert Krebbers, and Lars Birkedal. 'Compositional Non-Interference for Fine-Grained Concurrent Programs'. In: *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 2021, pp. 1416–1433. DOI: 10.1109/SP40001.2021.00003 (cited on pages 5, 157).

[87] Briti Gangopadhyay, Harshit Soora, and Pallab Dasgupta. 'Hierarchical Program-Triggered Reinforcement Learning Agents for Automated Driving'. In: *IEEE Transactions on Intelligent Transportation Systems* (2021), pp. 1–10. DOI: 10.1109/TITS.2021.3096998 (cited on page 77).

[88] Philippa Gardner, Sergio Maffeis, and Gareth David Smith. 'Towards a program logic for JavaScript'. In: *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*. Ed. by John Field and Michael Hicks. ACM, 2012, pp. 31–44. DOI: 10.1145/2103656.2103663 (cited on page 77).

[89] Dennis Giffhorn and Gregor Snelting. 'A new algorithm for low-deterministic security'. In: *Int. J. Inf. Sec.* 14.3 (2015), pp. 263–287. DOI: 10.1007/s10207-014-0257-6 (cited on pages 3, 101, 110, 111, 119, 157).

[90] Éric Goubault, Jérémy Ledent, and Samuel Mimram. 'Concurrent Specifications Beyond Linearizability'. In: *22nd International Conference on Principles of Distributed Systems, OPODIS 2018, December 17-19, 2018, Hong Kong, China*. Ed. by Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira. Vol. 125. LIPIcs. Schloss

Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 28:1–28:16. DOI: 10.4230/LIPIcs.OPODIS.2018.28 (cited on pages 136, 137).

[91] John Graham-Cumming and Jeff W. Sanders. 'On the Refinement of Non-Interference'. In: *4th IEEE Computer Security Foundations Workshop - CSFW'91, Franconia, New Hampshire, USA, June 18-20, 1991, Proceedings*. IEEE Computer Society, 1991, pp. 35–42. DOI: 10.1109/CSFW.1991.151567 (cited on page 157).

[92] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 'Online detection of effectively callback free objects with applications to smart contracts'. In: *PACMPL* 2.POPL (2018), 48:1–48:28. DOI: 10.1145/3158136 (cited on pages 165, 201).

[93] Tony Grue, Sergei Vorobev, Jukka Lehtosalo, and Guido van Rossum. *PyAnnotate*. https://github.com/dropbox/pyannotate (cited on page 76).

[94] Osman Gazi Güçlütürk. *The DAO Hack Explained: Unfortunate Take-off of Smart Contracts*. Accessed on 2021-03-31. 2018. URL: https://medium.com/@oguclutürk/the-dao-hack-explained-unfortunate-take-off-of-smart-contracts-2bd8c8db3562 (cited on pages 159, 164, 199).

[95] Dwight Guth. 'A formal semantics of Python 3.3'. MA thesis. University of Illinois at Urbana-Champaign, July 2013 (cited on page 76).

[96] Ákos Hajdu and Dejan Jovanovic. 'solc-verify: A Modular Verifier for Solidity Smart Contracts'. In: *CoRR* abs/1907.04262 (2019) (cited on pages 160, 202).

[97] Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. 'MaxSMT-Based Type Inference for Python 3'. In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*. Ed. by Hana Chockler and Georg Weissenbacher. Vol. 10982. Lecture Notes in Computer Science. Springer, 2018, pp. 12–19. DOI: 10.1007/978-3-319-96142-2_2 (cited on pages 8, 76).

[98] Chris Hawblitzel, Ming Kawaguchi, Shuvendu K. Lahiri, and Henrique Rebêlo. 'Towards Modularly Comparing Programs Using Automated Theorem Provers'. In: *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*. Ed. by Maria Paola Bonacina. Vol. 7898. Lecture Notes in Computer Science. Springer, 2013, pp. 282–299. DOI: 10.1007/978-3-642-38574-2_20 (cited on pages 80, 92, 118, 119, 156).

[99] Daniel Hedin and David Sands. 'Timing Aware Information Flow Security for a JavaCard-like Bytecode'. In: *Electron. Notes Theor. Comput. Sci.* 141.1 (2005), pp. 163–182. DOI: 10.1016/j.entcs.2005.02.031 (cited on page 91).

[100] Cedric Hegglin. 'Counterexamples for a Rust Verifier'. Bachelor's thesis. ETH Zürich, 2021 (cited on page 77).

[101] Fritz Henglein, Christian Kjær Larsen, and Agata Murawska. 'A Formally Verified Static Analysis Framework for Compositional Contracts'. In: *Financial Cryptography and Data Security - FC 2020 International Workshops, AsiaUSEC, CoDeFi, VOTING, and WTSC, Kota Kinabalu, Malaysia, February 14, 2020, Revised Selected Papers*. Ed. by Matthew Bernhard, Andrea Bracciali, L. Jean Camp, Shin'ichiro Matsuo, Alana Maurushat, Peter B. Rønne, and Massimiliano Sala. Vol. 12063. Lecture Notes in Computer Science. Springer, 2020, pp. 599–619. DOI: 10.1007/978-3-030-54455-3_42 (cited on page 203).

[102] Stefan Heule, K. Rustan M. Leino, Peter Müller, and Alexander J. Summers. 'Abstract Read Permissions: Fractional Permissions without the Fractions'. In: *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*. Ed. by Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni. Vol. 7737. Lecture Notes in Computer Science. Springer, 2013, pp. 315–334. DOI: 10.1007/978-3-642-35873-9_20 (cited on page 62).

[103] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon M. Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. 'KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine'. In: *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*. IEEE Computer Society, 2018, pp. 204–217. DOI: 10.1109/CSF.2018.00022 (cited on pages 160, 202).

[104] Yoichi Hirai. 'Defining the Ethereum Virtual Machine for Interactive Theorem Provers'. In: *Financial Cryptography and Data Security - FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers*. Ed. by Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y. A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson. Vol. 10323. Lecture Notes in Computer Science. Springer, 2017, pp. 520–535. DOI: 10.1007/978-3-319-70278-0_33 (cited on page 202).

[105] C. A. R. Hoare. 'An Axiomatic Basis for Computer Programming'. In: *Commun. ACM* 12.10 (1969), pp. 576–580. DOI: 10.1145/363235.363259 (cited on pages 2, 11).

[106] Marieke Huisman, Pratik Worah, and Kim Sunesen. 'A Temporal Logic Characterisation of Observational Determinism'. In: *19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006), 5-7 July 2006, Venice, Italy*. IEEE Computer Society, 2006, p. 3. DOI: 10.1109/CSFW.2006.6 (cited on pages 147, 149).

[107] Akifumi Imanishi, Kohei Suenaga, and Atsushi Igarashi. 'A guess-and-assume approach to loop fusion for program verification'. In: *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, Los Angeles, CA, USA, January 8-9, 2018*. 2018, pp. 2–14 (cited on page 119).

[108] Bart Jacobs and Frank Piessens. 'Expressive modular fine-grained concurrency specification'. In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. Ed. by Thomas Ball

and Mooly Sagiv. ACM, 2011, pp. 271–282. doi: 10.1145/1926385.1926417 (cited on pages 136, 137).

[109] Bart Jacobs, Frank Piessens, K. Rustan M. Leino, and Wolfram Schulte. 'Safe Concurrency for Aggregate Objects with Invariants'. In: *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005), 7-9 September 2005, Koblenz, Germany*. Ed. by Bernhard K. Aichernig and Bernhard Beckert. IEEE Computer Society, 2005, pp. 137–147. doi: 10.1109/SEFM.2005.39 (cited on pages 136, 137).

[110] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 'VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java'. In: *NASA Formal Methods: Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*. Ed. by Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 41–55 (cited on pages 43, 61, 76).

[111] Cliff B. Jones. 'Developing methods for computer programs including a notion of interference'. PhD thesis. University of Oxford, UK, 1981 (cited on page 157).

[112] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 'Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning'. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. Ed. by Sriram K. Rajamani and David Walker. ACM, 2015, pp. 637–650. doi: 10.1145/2676726.2676980 (cited on page 157).

[113] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 'ZEUS: Analyzing Safety of Smart Contracts'. In: *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018 (cited on pages 160, 202).

[114] Ioannis T Kassios and Peter Müller. 'Specification and verification of closures'. In: *Technical Report/ETH Zurich, Department of Computer Science* 660 (2010) (cited on page 35).

[115] Ioannis T. Kassios. 'Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions'. In: *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*. Ed. by Jayadev Misra, Tobias Nipkow, and Emil Sekerinski. Vol. 4085. Lecture Notes in Computer Science. Springer, 2006, pp. 268–283. doi: 10.1007/11813040_19 (cited on pages 13, 165).

[116] Christian Knabenhans. 'Automatic Inference of Hyperproperties'. Bachelor's thesis. ETH Zürich, 2018 (cited on page 120).

[117] Ralf Küsters, Tomasz Truderung, Bernhard Beckert, Daniel Bruns, Michael Kirsten, and Martin Mohr. 'A Hybrid Approach for Proving Noninterference of Java Programs'. In: *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*. 2015, pp. 305–319 (cited on pages 110, 111).

[118] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. ‘SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs’. In: *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. Ed. by P. Madhusudan and Sanjit A. Seshia. Vol. 7358. Lecture Notes in Computer Science. Springer, 2012, pp. 712–717. DOI: 10.1007/978-3-642-31424-7_54 (cited on pages 123, 156).

[119] Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, and Chris Hawblitzel. ‘Differential assertion checking’. In: *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE’13, Saint Petersburg, Russian Federation, August 18-26, 2013*. Ed. by Bertrand Meyer, Luciano Baresi, and Mira Mezini. ACM, 2013, pp. 345–355. DOI: 10.1145/2491411.2491452 (cited on pages 80, 92, 118, 123, 156, 158).

[120] Enmei Lai and Wenjun Luo. ‘Static Analysis of Integer Overflow of Smart Contracts in Ethereum’. In: *ICCSP 2020: 4th International Conference on Cryptography, Security and Privacy, Nanjing, China, January 10-12, 2020*. ACM, 2020, pp. 110–115. DOI: 10.1145/3377644.3377650 (cited on pages 160, 201).

[121] Akash Lal and Shaz Qadeer. ‘Reachability Modulo Theories’. In: *Reachability Problems - 7th International Workshop, RP 2013, Uppsala, Sweden, September 24-26, 2013 Proceedings*. Ed. by Parosh Aziz Abdulla and Igor Potapov. Vol. 8169. Lecture Notes in Computer Science. Springer, 2013, pp. 23–44. DOI: 10.1007/978-3-642-41036-9_4 (cited on page 156).

[122] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, and Daniel M. Zimmerman. *JML reference manual*. 2008 (cited on pages 11, 168, 169).

[123] Hermann Lehner. ‘A formal definition of JML in Coq and its application to runtime assertion checking’. PhD thesis. ETH Zurich, 2011 (cited on page 63).

[124] Hermann Lehner and Peter Müller. ‘Formal Translation of Bytecode into BoogiePL’. In: *Electron. Notes Theor. Comput. Sci.* 190.1 (2007), pp. 35–50. DOI: 10.1016/j.entcs.2007.02.059 (cited on page 63).

[125] Jukka Lehtosalo et al. *Mypy - Optional Static Typing for Python*. http://mypy-lang.org. 2017 (cited on pages 26, 66).

[126] Jukka Lehtosalo et al. *Google LLC*. https://github.com/google/pytype/. 2018 (cited on page 76).

[127] K. Rustan M. Leino. ‘Dafny: An Automatic Program Verifier for Functional Correctness’. In: *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*. Ed. by Edmund M. Clarke and Andrei Voronkov. Vol. 6355. Lecture Notes in Computer Science. Springer, 2010, pp. 348–370. DOI: 10.1007/978-3-642-17511-4_20 (cited on pages 76, 123).

[128] K. Rustan M. Leino and Peter Müller. 'Object Invariants in Dynamic Contexts'. In: *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings*. Ed. by Martin Odersky. Vol. 3086. Lecture Notes in Computer Science. Springer, 2004, pp. 491–516. DOI: 10.1007/978-3-540-24851-4_22 (cited on page 3).

[129] K. Rustan M. Leino and Peter Müller. 'Object Invariants in Dynamic Contexts'. In: *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings*. Ed. by Martin Odersky. Vol. 3086. Lecture Notes in Computer Science. Springer, 2004, pp. 491–516. DOI: 10.1007/978-3-540-24851-4_22 (cited on page 173).

[130] K. Rustan M. Leino and Peter Müller. 'Using the Spec# Language, Methodology, and Tools to Write Bug-Free Programs'. In: *Advanced Lectures on Software Engineering, LASER Summer School 2007/2008*. Ed. by Peter Müller. Vol. 6029. Lecture Notes in Computer Science. Springer, 2008, pp. 91–139. DOI: 10.1007/978-3-642-13010-6_4 (cited on pages 43, 76, 123).

[131] K. Rustan M. Leino and Peter Müller. 'Verification of Equivalent-Results Methods'. In: *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. 2008, pp. 307–321 (cited on page 120).

[132] K. Rustan M. Leino and Peter Müller. 'A Basis for Verifying Multi-threaded Programs'. In: *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*. Ed. by Giuseppe Castagna. Vol. 5502. Lecture Notes in Computer Science. Springer, 2009, pp. 378–393. DOI: 10.1007/978-3-642-00590-9_27 (cited on pages 12, 15, 56, 137).

[133] K. Rustan M. Leino, Peter Müller, and Jan Smans. 'Verification of Concurrent Programs with Chalice'. In: *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures*. Ed. by Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri. Vol. 5705. Lecture Notes in Computer Science. Springer, 2009, pp. 195–222. DOI: 10.1007/978-3-642-03829-7_7 (cited on pages 133, 136).

[134] Ivan Levkivskyi, Jukka Lehtosalo, and Łukasz Langa. *PEP 544: Protocols: Structural subtyping (static duck typing)*. https://www.python.org/dev/peps/pep-0544/. 2017 (cited on pages 25, 36).

[135] Guodong Li, Esben Andreasen, and Indradeep Ghosh. 'SymJS: automatic symbolic testing of JavaScript web applications'. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. Ed. by Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey. ACM, 2014, pp. 449–459. DOI: 10.1145/2635868.2635913 (cited on page 77).

[136] Peng Li and Steve Zdancewic. 'Downgrading policies and relaxed noninterference'. In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*. 2005, pp. 158–170 (cited on page 120).

[137] Barbara Liskov and Jeannette M. Wing. 'Specifications and Their Use in Defining Subtypes'. In: *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Eighth Annual Conference, Washington, DC, USA, September 26 - October 1, 1993, Proceedings*. Ed. by Timlynn Babitsky and Jim Salmons. ACM, 1993, pp. 16–28. DOI: 10.1145/165854.165863 (cited on page 168).

[138] Barbara Liskov and Jeannette M. Wing. 'A Behavioral Notion of Subtyping'. In: *ACM Trans. Program. Lang. Syst.* 16.6 (1994), pp. 1811–1841. DOI: 10.1145/197320.197383 (cited on pages 33, 39, 130).

[139] Ye Liu, Yi Li, Shang-Wei Lin, and Rong Zhao. 'Towards automated verification of smart contract fairness'. In: *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*. Ed. by Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann. ACM, 2020, pp. 666–677. DOI: 10.1145/3368089.3409740 (cited on page 203).

[140] Gavin Lowe. 'A Hierarchy of Authentication Specification'. In: *10th Computer Security Foundations Workshop (CSFW '97), June 10-12, 1997, Rockport, Massachusetts, USA*. IEEE Computer Society, 1997, pp. 31–44. DOI: 10.1109/CSFW.1997.596782 (cited on page 74).

[141] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 'Making Smart Contracts Smarter'. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. Ed. by Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi. ACM, 2016, pp. 254–269. DOI: 10.1145/2976749.2978309 (cited on page 201).

[142] Eva Maia, Nelma Moreira, and Rogério Reis. 'A Static Type Inference for Python'. In: *DYLA*. 2012 (cited on page 76).

[143] Petar Maksimovic, Sacha-Élie Ayoun, José Fragoso Santos, and Philippa Gardner. 'Gillian, Part II: Real-World Verification for JavaScript and C'. In: *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*. Ed. by Alexandra Silva and K. Rustan M. Leino. Vol. 12760. Lecture Notes in Computer Science. Springer, 2021, pp. 827–850. DOI: 10.1007/978-3-030-81688-9_38 (cited on page 77).

[144] Daniel Matichuk, Toby C. Murray, June Andronick, D. Ross Jeffery, Gerwin Klein, and Mark Staples. 'Empirical Study Towards a Leading Indicator for Cost of Formal Software Verification'. In: *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. Ed. by Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum. IEEE Computer Society, 2015, pp. 722–732. DOI: 10.1109/ICSE.2015.85 (cited on page 12).

[145] Laurent Mauborgne and Xavier Rival. 'Trace Partitioning in Abstract Interpretation Based Static Analyzers'. In: *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*. Ed. by Shmuel Sagiv. Vol. 3444. Lecture Notes in Computer Science. Springer, 2005, pp. 5–20. DOI: 10.1007/978-3-540-31987-0_2 (cited on page 120).

[146] Anastasia Mavridou, Aron Laszka, Emmanouela Stachtiari, and Abhishek Dubey. 'VeriSolid: Correct-by-Design Smart Contracts for Ethereum'. In: *Financial Cryptography and Data Security - 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers*. Ed. by Ian Goldberg and Tyler Moore. Vol. 11598. Lecture Notes in Computer Science. Springer, 2019, pp. 446–465. DOI: 10.1007/978-3-030-32101-7_27 (cited on page 202).

[147] John McCall, Doug Gregor, Konrad Malawski, and Chris Lattner. *SE-0306: Actors*. Accessed on 2021-08-07. 2021. URL: https://github.com/apple/swift-evolution/blob/main/proposals/0306-actors.md (cited on page 166).

[148] Severin Meier. 'Verification of information flow security for Python programs'. MA thesis. ETH Zürich, 2018 (cited on pages 77, 152).

[149] Mark S. Miller. 'Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control'. PhD thesis. Johns Hopkins University, May 2006 (cited on page 203).

[150] Mark S. Miller, Chip Morningstar, and Bill Frantz. 'Capability-Based Financial Instruments'. In: *Financial Cryptography, 4th International Conference, FC 2000 Anguilla, British West Indies, February 20-24, 2000, Proceedings*. Ed. by Yair Frankel. Vol. 1962. Lecture Notes in Computer Science. Springer, 2000, pp. 349–378. DOI: 10.1007/3-540-45472-1_24 (cited on page 203).

[151] Vittorio Minacori. 'EIP-1363: ERC-1363 Payable Token'. In: *Ethereum Improvement Proposals* 1363 (2020) (cited on pages 165, 199, 200).

[152] Vittorio Minacori. *ERC-1363 Payable Token*. Accessed on 2021-04-16. 2021. URL: https://github.com/vittominacori/erc1363-payable-token (cited on page 197).

[153] Antoine Miné. 'The octagon abstract domain'. In: *High. Order Symb. Comput.* 19.1 (2006), pp. 31–100. DOI: 10.1007/s10990-006-8609-1 (cited on page 120).

[154] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. 'Static Type Analysis by Abstract Interpretation of Python Programs'. In: *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Ed. by Robert Hirschfeld and Tobias Pape. Vol. 166. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, 17:1–17:29. DOI: 10.4230/LIPIcs.ECOOP.2020.17 (cited on page 76).

[155] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 'Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts'. In: *CoRR* abs/1907.03890 (2019) (cited on page 201).

[156] Leonardo Mendonça de Moura and Nikolaj Bjørner. 'Z3: An Efficient SMT Solver'. In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. doi: 10.1007/978-3-540-78800-3_24 (cited on pages 3, 10, 16, 67, 195).

[157] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*. Vol. 2262. Lecture Notes in Computer Science. Springer, 2002 (cited on pages 3, 13, 165).

[158] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 'Automatic Verification of Iterated Separating Conjunctions Using Symbolic Execution'. In: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*. 2016, pp. 405–425 (cited on page 110).

[159] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 'Viper: A Verification Infrastructure for Permission-Based Reasoning'. In: *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*. Ed. by Barbara Jobstmann and K. Rustan M. Leino. Vol. 9583. Lecture Notes in Computer Science. Springer, 2016, pp. 41–62. doi: 10.1007/978-3-662-49122-5_2 (cited on pages 4, 10, 16–18, 20, 64, 76, 81, 109, 123, 195).

[160] Toby C. Murray, Robert Sison, and Kai Engelhardt. 'COVERN: A Logic for Compositional Verification of Information Flow Control'. In: *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*. IEEE, 2018, pp. 16–30. doi: 10.1109/EuroSP.2018.00010 (cited on pages 5, 81, 102, 124, 145, 154, 157).

[161] Andrew C Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. 'Jif: Java information flow'. In: *Software release. Located at http://www. cs. cornell. edu/jif* 2005 (2001) (cited on page 157).

[162] Ellie Myler and George Broadbent. 'ISO 17799: Standard for security'. In: *Information Management* 40.6 (2006), p. 43 (cited on page 1).

[163] Ramana Nagasamudram and David A. Naumann. 'Alignment Completeness for Relational Hoare Logics'. In: *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*. IEEE, 2021, pp. 1–13. doi: 10.1109/LICS52264.2021.9470690 (cited on page 117).

[164] David A. Naumann. 'From Coupling Relations to Mated Invariants for Checking Information Flow'. In: *Computer Security - ESORICS 2006, 11th European Symposium on Research in Computer Security, Hamburg, Germany, September 18-20, 2006, Proceedings*. 2006, pp. 279–296 (cited on pages 104, 110, 111, 117).

[165] David A. Naumann. 'From Coupling Relations to Mated Invariants for Checking Information Flow'. In: *Computer Security - ESORICS 2006, 11th European Symposium on Research in Computer Security, Hamburg, Germany, September 18-20, 2006, Proceedings*. Ed. by Dieter Gollmann, Jan Meier, and Andrei Sabelfeld. Vol. 4189. Lecture Notes in Computer Science. Springer, 2006, pp. 279–296. DOI: 10.1007/11863908_18 (cited on page 91).

[166] Keerthi Nelaturu, Anastasia Mavridou, Andreas G. Veneris, and Aron Laszka. 'Verified Development and Deployment of Multiple Interacting Smart Contracts with VeriSolid'. In: *IEEE International Conference on Blockchain and Cryptocurrency, ICBC 2020, Toronto, ON, Canada, May 2-6, 2020*. IEEE, 2020, pp. 1–9. DOI: 10.1109/ICBC48266.2020.9169428 (cited on page 202).

[167] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 'Finding The Greedy, Prodigal, and Suicidal Contracts at Scale'. In: *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. ACM, 2018, pp. 653–663. DOI: 10.1145/3274694.3274743 (cited on page 201).

[168] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science. Springer, 2002 (cited on page 3).

[169] Peter W. O'Hearn. 'Resources, concurrency, and local reasoning'. In: *Theor. Comput. Sci.* 375.1-3 (2007), pp. 271–307. DOI: 10.1016/j.tcs.2006.12.035 (cited on pages 3, 12, 15, 134, 136, 152).

[170] Susan S. Owicki and David Gries. 'An Axiomatic Proof Technique for Parallel Programs I'. In: *Acta Informatica* 6 (1976), pp. 319–340. DOI: 10.1007/BF00268134 (cited on page 157).

[171] Matthew Parkinson and Gavin Bierman. 'Separation Logic and Abstraction'. In: *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '05. Long Beach, California, USA: ACM, 2005, pp. 247–258 (cited on pages 36, 42, 62).

[172] Willem Penninckx, Bart Jacobs, and Frank Piessens. 'Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs'. In: *Programming Languages and Systems: 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings*. Ed. by Jan Vitek. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 158–182 (cited on pages 3, 61, 62).

[173] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. *VerX smart contract verification benchmarks*. Accessed on 2021-04-16. 2019. URL: https://github.com/eth-sri/verx-benchmarks (cited on page 197).

[174] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin T. Vechev. 'VerX: Safety Verification of Smart Contracts'. In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1661–1677. DOI: 10.1109/SP40000.2020.00024 (cited on pages 160, 165, 199, 202).

[175] Lauren Pick, Grigory Fedyukovich, and Aarti Gupta. 'Exploiting Synchrony and Symmetry in Relational Verification'. In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*. 2018, pp. 164–182 (cited on pages 111, 114, 117).

[176] Lauren Pick, Grigory Fedyukovich, and Aarti Gupta. 'Automating Modular Verification of Secure Information Flow'. In: *2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020*. IEEE, 2020, pp. 158–168. DOI: 10.34727/2020/isbn.978-3-85448-042-6_23 (cited on page 120).

[177] Arnd Poetzsch-Heffter. 'Specification and verification of object-oriented programs'. Habilitation thesis. TU München, 1997 (cited on pages 3, 11).

[178] Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. 'Python: The Full Monty'. In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*. OOPSLA '13. Indianapolis, Indiana, USA: ACM, 2013, pp. 217–232 (cited on page 63).

[179] Adi Prabawa, Mahmudul Faisal Al Ameen, Benedict Lee, and Wei-Ngan Chin. 'A Logical System for Modular Information Flow Verification'. In: *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings*. 2018, pp. 430–451 (cited on pages 3, 120).

[180] John C. Reynolds. 'Separation Logic: A Logic for Shared Mutable Data Structures'. In: *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817 (cited on pages 3, 13, 15, 91, 159, 165, 189).

[181] Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa. *PEP 484: Type Hints*. https://www.python.org/dev/peps/pep-0484/. 2014 (cited on pages 9, 25, 36).

[182] Guido van Rossum and Ivan Levkivskyi. *PEP 483: The Theory of Type Hints*. https://www.python.org/dev/peps/pep-0483/. 2014 (cited on pages 25, 36).

[183] Grigore Rosu and Traian-Florin Serbanuta. 'An overview of the K semantic framework'. In: *J. Log. Algebr. Program.* 79.6 (2010), pp. 397–434 (cited on page 76).

[184] R.B. Rubbens. 'Improving Support for Java Exceptions and Inheritance in VerCors'. MA thesis. University of Twente, May 2020 (cited on pages 56, 77).

[185] Alejandro Russo, John Hughes, David A. Naumann, and Andrei Sabelfeld. 'Closing Internal Timing Channels by Transformation'. In: *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues, 11th Asian Computing Science Conference, Tokyo, Japan, December 6-8, 2006, Revised Selected Papers*. Ed. by Mitsu Okada and Ichiro Satoh. Vol. 4435. Lecture Notes in Computer Science. Springer, 2006, pp. 120–135. doi: 10.1007/978-3-540-77505-8_10 (cited on page 158).

[186] Andrei Sabelfeld and Andrew C. Myers. 'A Model for Delimited Information Release'. In: *Software Security - Theories and Systems, Second Mext-NSF-JSPS International Symposium, ISSS 2003, Tokyo, Japan, November 4-6, 2003, Revised Papers*. 2003, pp. 174–191 (cited on pages 105, 120).

[187] Andrei Sabelfeld and David Sands. 'Probabilistic Noninterference for Multi-Threaded Programs'. In: *Proceedings of the 13th IEEE Computer Security Foundations Workshop, CSFW '00, Cambridge, England, UK, July 3-5, 2000*. IEEE Computer Society, 2000, pp. 200–214. doi: 10.1109/CSFW.2000.856937 (cited on pages 135, 136).

[188] Andrei Sabelfeld and David Sands. 'Dimensions and Principles of Declassification'. In: *18th IEEE Computer Security Foundations Workshop, (CSFW-18 2005), 20-22 June 2005, Aix-en-Provence, France*. 2005, pp. 255–269 (cited on pages 105, 120).

[189] José Fragoso Santos, Petar Maksimovic, Daiva Naudziuniene, Thomas Wood, and Philippa Gardner. 'JaVerT: JavaScript verification toolchain'. In: *Proc. ACM Program. Lang.* 2.POPL (2018), 50:1–50:33. doi: 10.1145/3158138 (cited on page 77).

[190] José Fragoso Santos, Petar Maksimovic, Gabriela Sampaio, and Philippa Gardner. 'JaVerT 2.0: compositional symbolic execution for JavaScript'. In: *Proc. ACM Program. Lang.* 3.POPL (2019), 66:1–66:31. doi: 10.1145/3290379 (cited on page 77).

[191] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 'A Symbolic Execution Framework for JavaScript'. In: *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society, 2010, pp. 513–528. doi: 10.1109/SP.2010.38 (cited on page 77).

[192] Christoph Scheben and Peter H. Schmitt. 'Verification of Information Flow Properties of Java Programs without Approximations'. In: *Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2011, Turin, Italy, October 5-7, 2011, Revised Selected Papers*. Ed. by Bernhard Beckert, Ferruccio Damiani, and Dilian Gurov. Vol. 7421. Lecture Notes in Computer Science. Springer, 2011, pp. 232–249. doi: 10.1007/978-3-642-31762-0_15 (cited on page 120).

[193] Benjamin Schmid. 'Abstract Read Permission Support for an Automatic Python Verifier'. Bachelor's thesis. 2018 (cited on pages 62, 77).

[194] Daniel Schoepe, Toby Murray, and Andrei Sabelfeld. 'VERONICA: Expressive and Precise Concurrent Information Flow Security'. In: *33rd IEEE Computer Security Foundations Symposium, CSF 2020, Boston, MA, USA, June 22-26, 2020*. IEEE, 2020, pp. 79–94. DOI: 10.1109/CSF49147.2020.00014 (cited on pages 5, 154, 157).

[195] Koushik Sen, George C. Necula, Liang Gong, and Wontae Choi. 'MultiSE: multi-path symbolic execution using value summaries'. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. Ed. by Elisabetta Di Nitto, Mark Harman, and Patrick Heymans. ACM, 2015, pp. 842–853. DOI: 10.1145/2786805.2786830 (cited on page 77).

[196] Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. 'Safer smart contract programming with Scilla'. In: *PACMPL* 3.OOPSLA (2019), 185:1–185:30. DOI: 10.1145/3360611 (cited on page 202).

[197] Ron Shemer, Arie Gurfinkel, Sharon Shoham, and Yakir Vizel. 'Property Directed Self Composition'. In: *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*. Ed. by Isil Dillig and Serdar Tasiran. Vol. 11561. Lecture Notes in Computer Science. Springer, 2019, pp. 161–179. DOI: 10.1007/978-3-030-25540-4_9 (cited on page 118).

[198] Robin Sierra. 'Verification of Ethereum Smart Contracts Written in Vyper'. MA thesis. ETH Zürich, 2019 (cited on pages 187, 196).

[199] Jan Smans, Bart Jacobs, and Frank Piessens. 'Implicit dynamic frames'. In: *ACM Trans. Program. Lang. Syst.* 34.1 (2012), 2:1–2:58. DOI: 10.1145/2160910.2160911 (cited on pages 13, 62, 91, 152).

[200] Yannis Smaragdakis, Neville Grech, Sifis Lagouvardos, Konstantinos Triantafyllou, and Ilias Tsatiris. 'Symbolic value-flow static analysis: deep, precise, complete modeling of Ethereum smart contracts'. In: *Proc. ACM Program. Lang.* 5.OOPSLA (2021), pp. 1–30. DOI: 10.1145/3485540 (cited on page 202).

[201] Geoffrey Smith. 'A New Type System for Secure Information Flow'. In: *14th IEEE Computer Security Foundations Workshop (CSFW-14 2001), 11-13 June 2001, Cape Breton, Nova Scotia, Canada*. 2001, pp. 115–125 (cited on pages 145, 157).

[202] Geoffrey Smith. 'Improved typings for probabilistic noninterference in a multi-threaded language'. In: *J. Comput. Secur.* 14.6 (2006), pp. 591–623 (cited on page 157).

[203] Geoffrey Smith. 'Principles of Secure Information Flow Analysis'. In: *Malware Detection*. 2007, pp. 291–307. DOI: 10.1007/978-0-387-44599-1_13 (cited on pages 3, 157).

[204] Geoffrey Smith. 'Principles of Secure Information Flow Analysis'. In: *Malware Detection*. 2007, pp. 291–307 (cited on pages 110, 111, 119).

[205] Geoffrey Smith and Dennis M. Volpano. 'Secure Information Flow in a Multi-Threaded Imperative Language'. In: *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*. Ed. by David B. MacQueen and Luca Cardelli. ACM, 1998, pp. 355–364. DOI: 10.1145/268946.268975 (cited on pages 137, 142, 157).

[206] Marcelo Sousa and Isil Dillig. 'Cartesian Hoare logic for verifying k-safety properties'. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. 2016, pp. 57–69 (cited on pages 5, 79, 111, 112, 114, 117).

[207] Christoph Sprenger, Tobias Klenze, Marco Eilers, Felix A. Wolf, Peter Müller, Martin Clochard, and David A. Basin. 'Igloo: soundly linking compositional refinement and separation logic for distributed system verification'. In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 152:1–152:31. DOI: 10.1145/3428220 (cited on pages 8, 61, 74, 77).

[208] Deian Stefan, Alejandro Russo, Pablo Buiras, Amit Levy, John C. Mitchell, and David Mazières. 'Addressing covert termination and timing channels in concurrent information flow systems'. In: *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*. Ed. by Peter Thiemann and Robby Bruce Findler. ACM, 2012, pp. 201–214. DOI: 10.1145/2364527.2364557 (cited on page 158).

[209] Samuel Steffen, Benjamin Bichsel, Mario Gersbach, Noa Melchior, Petar Tsankov, and Martin T. Vechev. 'zkay: Specifying and Enforcing Data Privacy in Smart Contracts'. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. Ed. by Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz. ACM, 2019, pp. 1759–1776. DOI: 10.1145/3319535.3363222 (cited on page 159).

[210] Jon Stephens, Kostas Ferles, Benjamin Mariano, Shuvendu K. Lahiri, and Isil Dillig. 'SmartPulse: Automated Checking of Temporal Properties in Smart Contracts'. In: *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 2021, pp. 555–571. DOI: 10.1109/SP40001.2021.00085 (cited on page 202).

[211] Tachio Terauchi and Alexander Aiken. 'Secure Information Flow as a Safety Problem'. In: *Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings*. 2005, pp. 352–367 (cited on pages 80, 82, 105, 110, 111, 117).

[212] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 'SmartCheck: Static Analysis of Ethereum Smart Contracts'. In: *1st IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2018, Gothenburg, Sweden, May 27 - June 3, 2018*. ACM, 2018, pp. 9–16 (cited on pages 160, 201).

[213] Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. 'Securify: Practical Security Analysis of Smart Contracts'. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. 2018, pp. 67–82. DOI: 10.1145/3243734.3243780 (cited on pages 160, 201).

[214] Uniswap. *Uniswap version 1*. Accessed on 2021-04-16. 2019. URL: https://github.com/Uniswap/uniswap-v1 (cited on page 197).

[215] Caterina Urban. 'What Programs Want: Automatic Inference of Input Data Specifications'. In: *CoRR* abs/2007.10688 (2020) (cited on page 76).

[216] Viktor Vafeiadis. 'Concurrent Separation Logic and Operational Semantics'. In: *Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics, MFPS 2011, Pittsburgh, PA, USA, May 25-28, 2011*. Ed. by Michael W. Mislove and Joël Ouaknine. Vol. 276. Electronic Notes in Theoretical Computer Science. Elsevier, 2011, pp. 335–351. DOI: 10.1016/j.entcs.2011.09.029 (cited on page 62).

[217] Viktor Vafeiadis and Chinmay Narayan. 'Relaxed separation logic: a program logic for C11 concurrency'. In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. Ed. by Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes. ACM, 2013, pp. 867–884. DOI: 10.1145/2509136.2509532 (cited on page 157).

[218] Marco Vassena, Gary Soeller, Peter Amidon, Matthew Chan, John Renner, and Deian Stefan. 'Foundations for Parallel Information Flow Control Runtime Systems'. In: *Principles of Security and Trust - 8th International Conference, POST 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*. Ed. by Flemming Nielson and David Sands. Vol. 11426. Lecture Notes in Computer Science. Springer, 2019, pp. 1–28. DOI: 10.1007/978-3-030-17138-4_1 (cited on page 158).

[219] Frédéric Vogels, Bart Jacobs, and Frank Piessens. 'Featherweight VeriFast'. In: *Log. Methods Comput. Sci.* 11.3 (2015). DOI: 10.2168/LMCS-11(3:19)2015 (cited on page 63).

[220] Fabian Vogelsteller and Vitalik Buterin. 'EIP-20: ERC-20 Token Standard'. In: *Ethereum Improvement Proposals* 20 (2015) (cited on pages 164, 170).

[221] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. 'A Sound Type System for Secure Flow Analysis'. In: *J. Comput. Secur.* 4.2/3 (1996), pp. 167–188. DOI: 10.3233/JCS-1996-42-304 (cited on page 101).

[222] Dennis M. Volpano and Geoffrey Smith. 'Probabilistic Noninterference in a Concurrent Language'. In: *Proceedings of the 11th IEEE Computer Security Foundations Workshop, Rockport, Massachusetts, USA, June 9-11, 1998*. IEEE Computer Society, 1998, pp. 34–43. DOI: 10.1109/CSFW.1998.683153 (cited on page 144).

[223] David Wagner. *An attack on a mint*. Accessed on 2022-09-21. 2008. URL: https://web.archive.org/web/20160620145146/http://www.eros-os.org/pipermail/e-lang/2008-March/012516.html (cited on page 164).

[224] Benjamin Weber. 'Automatic Verification of Closures and Lambda-Functions in Python'. MA thesis. ETH Zürich, 2017 (cited on pages 62, 71, 77).

[225] Scott Wesley, Maria Christakis, Jorge A. Navas, Richard J. Trefler, Valentin Wüstholz, and Arie Gurfinkel. 'Compositional Verification of Smart Contracts Through Communication Abstraction'. In: *Static Analysis - 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17-19, 2021, Proceedings*. Ed. by Cezara Dragoi, Suvam Mukherjee, and Kedar S. Namjoshi. Vol. 12913. Lecture Notes in Computer Science. Springer, 2021, pp. 429–452. DOI: 10.1007/978-3-030-88806-0_21 (cited on page 202).

[226] Scott Wesley, Maria Christakis, Jorge A. Navas, Richard J. Trefler, Valentin Wüstholz, and Arie Gurfinkel. 'Verifying Solidity Smart Contracts via Communication Abstraction in SmartACE'. In: *Verification, Model Checking, and Abstract Interpretation - 23rd International Conference, VMCAI 2022, Philadelphia, PA, USA, January 16-18, 2022, Proceedings*. Ed. by Bernd Finkbeiner and Thomas Wies. Vol. 13182. Lecture Notes in Computer Science. Springer, 2022, pp. 425–449. DOI: 10.1007/978-3-030-94583-1_21 (cited on page 160).

[227] Erik Wittern, Annie T. T. Ying, Yunhui Zheng, Julian Dolby, and Jim Alain Laredo. 'Statically checking web API requests in JavaScript'. In: *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. Ed. by Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard. IEEE, 2017, pp. 244–254. DOI: 10.1109/ICSE.2017.30 (cited on page 77).

[228] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João C. Pereira, and Peter Müller. 'Gobra: Modular Specification and Verification of Go Programs'. In: *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*. Ed. by Alexandra Silva and K. Rustan M. Leino. Vol. 12759. Lecture Notes in Computer Science. Springer, 2021, pp. 367–379. DOI: 10.1007/978-3-030-81685-8_17 (cited on pages 76, 123).

[229] Gavin Wood et al. 'Ethereum: A secure decentralised generalised transaction ledger'. In: *Ethereum project yellow paper* 151.2014 (2014), pp. 1–32 (cited on pages 159, 162).

[230] Pengbo Yan and Toby Murray. 'SecRSL: security separation logic for C11 release-acquire concurrency'. In: *Proc. ACM Program. Lang.* 5.OOPSLA (2021), pp. 1–26. DOI: 10.1145/3485476 (cited on page 157).

[231] Hongseok Yang. 'Relational separation logic'. In: *Theor. Comput. Sci.* 375.1-3 (2007), pp. 308–334 (cited on pages 79, 117).

[232] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 'Finding and Understanding Bugs in C Compilers'. In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. San Jose, California, USA: Association for Computing Machinery, 2011, pp. 283–294. DOI: 10.1145/1993498.1993532 (cited on page 62).

[233] Steve Zdancewic and Andrew C. Myers. 'Observational Determinism for Concurrent Program Security'. In: *16th IEEE Computer Security Foundations Workshop (CSFW-16 2003), 30 June - 2 July 2003, Pacific Grove, CA, USA*. IEEE Computer Society, 2003, p. 29. DOI: 10.1109/CSFW.2003.1212703 (cited on pages 147, 149, 157).

[234] Jingyi Emma Zhong, Kevin Cheang, Shaz Qadeer, Wolfgang Grieskamp, Sam Blackshear, Junkil Park, Yoni Zohar, Clark W. Barrett, and David L. Dill. 'The Move Prover'. In: *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*. Ed. by Shuvendu K. Lahiri and Chao Wang. Vol. 12224. Lecture Notes in Computer Science. Springer, 2020, pp. 137–150. DOI: 10.1007/978-3-030-53288-8_7 (cited on page 202).

Appendix

Proofs for Chapter 3 A.

A.1. Appendix

A.1.1. Preliminaries

In this section, we establish some general properties of the relations we defined as well as our language semantics. The proofs for virtually all of them are obvious (and we only describe them briefly); nevertheless, we explicitly name the properties that we rely on in the subsequent proofs of our main lemmas and theorems.

We start by defining two properties of the $\in_i$ relation w.r.t. store substitutions that follow immediately from its definition:

First, the relation is preserved by matching store substitutions:

Proposition A.1.1 *If* $\sigma_i \in_i \underline{\sigma}$ *then* $\sigma_i[x \mapsto v] \in_i \underline{\sigma}[x^{(i)} \mapsto v]$.

Second, the relation is preserved by substitions of variables belonging to other executions:

Proposition A.1.2 *If* $\sigma_i \in_i \underline{\sigma}$ *and* $j \neq i$ *then* $\sigma_i \in_i \underline{\sigma}[x^{(j)} \mapsto v]$.

Similarly, we define the following three properties for the $\preccurlyeq_i^V$ relation that also follow from its definition and, in the second case, the injectivity of the renaming:

First, the relation is transitive:

Proposition A.1.3 *If* $\underline{\sigma} \preccurlyeq_i^{V_1} \underline{\sigma}'' \wedge \underline{\sigma}'' \preccurlyeq_i^{V_2} \underline{\sigma}'$, *where* $V_1 \subseteq V$ *and* $V_2 \subseteq V$, *then* $\underline{\sigma} \preccurlyeq_i^V \underline{\sigma}'$.

Second, it is preserved by substitions of variables belonging to other executions:

Proposition A.1.4 *If* $\underline{\sigma} \preccurlyeq_i^V \underline{\sigma}'$ *then* $\underline{\sigma} \preccurlyeq_i^V \underline{\sigma}'[x^{(j)} \mapsto v]$ *for any* $j \neq i$.

Third, the relation always holds for a store holding only the excluded variables:

Proposition A.1.5 *If* $\{x_1, \ldots, x_n\} \subseteq V$ *then* $\underline{\sigma} \preccurlyeq_i^V \underline{\sigma}[x_1 \mapsto v_1] \ldots [x_n \mapsto v_n]$ *for any* $i, v_1, \ldots, v_n$.

Finally, we prove a lemma stating that the latter relation preserves the former:

Lemma A.1.6 *If* $\sigma \in_i \underline{\sigma}$ *and* $\underline{\sigma} \preccurlyeq_i^{\underline{s}} \underline{\sigma}'$ *then* $\sigma \in_i \underline{\sigma}'$.

Proof. Since all names in *freshvars*($\underline{s}$) are distinct from all names in PVar, and $\underline{\sigma}$ and $\underline{\sigma}'$ therefore agree on the values of the i-renamed variables in RPVar by the definition of $\preccurlyeq_i^V$, this follows from the definition of $\in_i$. □

Throughout the proofs of the following theorems and lemmas, we repeatedly make use of the following properties of our language semantics.

Firstly, the (normal or abnormal) termination of a sequential composition of statements implies the (normal or abnormal) termination of the substatements and vice versa:

Lemma A.1.7

1. $\langle s_1; s_2, \sigma\rangle \longrightarrow^l \langle \mathtt{skip}, \sigma'\rangle$ *implies that for some* l_1, l_2, σ'', *we have* $\langle s_1; s_2, \sigma\rangle \longrightarrow^{l_1} \langle \mathtt{skip}; s_2, \sigma''\rangle \longrightarrow \langle s_2, \sigma''\rangle \longrightarrow^{l_2} \langle \mathtt{skip}, \sigma'\rangle$ *and* $l = 1 + l_1 + l_2$.
2. $\langle s_1; s_2, \sigma\rangle \longrightarrow^l \langle \mathtt{skip}; s_2, \sigma'\rangle$ *implies that* $\langle s_1, \sigma\rangle \longrightarrow^l \langle \mathtt{skip}, \sigma'\rangle$.
3. *If* $\langle s_1; s_2, \sigma\rangle \longrightarrow^l \langle \dot{s}, \sigma'\rangle$ *and* $\dot{s} \neq \mathtt{skip}$ *then either* $\langle s_1, \sigma\rangle \longrightarrow^{l-1} \langle \dot{s}, \sigma'\rangle$ *or* $\langle s_1, \sigma\rangle \longrightarrow^{l_1} \langle \mathtt{skip}, \sigma''\rangle \wedge \langle s_2, \sigma''\rangle \longrightarrow^{l_2} \langle \dot{s}, \sigma'\rangle$ *for some* l_1, l_2 *s.t.* $l = l_1 + l_2 + 1$ *and* $l_1 \geq 0$ *and* $l_2 \geq 0$.

Proof. All three statements are proved by induction on l. For the first statement, the total number l of steps results from l_1 steps using Seq1, one step using Seq2, and the subsequent execution of s_2 in l_2 steps. □

Similarly, we show that the (normal or abnormal) termination of a `frame` statement implies that the framed statement terminates (normally or abnormally) from the framed store:

Lemma A.1.8

1. $\langle x_1, \ldots, x_m{:=}\mathtt{frame}_{p_1,\ldots,p_m}(s, \sigma_f), \sigma\rangle \longrightarrow^l \langle \mathtt{skip}, \sigma'\rangle$ *implies that* $l > 0$ *and* $\langle s, \sigma_f\rangle \longrightarrow^{l-1} \langle \mathtt{skip}, \sigma'_f\rangle$ *for some* σ'_f *s.t.* $\sigma' = \sigma[x_1 \mapsto \sigma'_f(p_1)] \ldots [x_m \mapsto \sigma'_f(p_m)]$.
2. $\langle x_1, \ldots, x_m{:=}\mathtt{frame}_{p_1,\ldots,p_m}(s, \sigma_f), \sigma\rangle \longrightarrow^l \langle \dot{s}, \sigma'\rangle$, *where* $\dot{s} \neq \mathtt{skip}$, *implies that* $l > 0$ *and* $\langle s, \sigma_f\rangle \longrightarrow^{l-1} \langle \dot{s}, \sigma'_f\rangle$ *for some* σ'_f *and* $\sigma' = \sigma$.

Proof. By induction on l. □

Finally, we have a simple fact about expression evaluation:

Lemma A.1.9 *If* $e \Downarrow_\sigma v$ *and* $x \notin fv(e)$ *then* $e \Downarrow_{\sigma[x \mapsto v']} v$

Proof. By induction on the structure of e. □

The next lemmas describe properties of sequential compositions of a number of (conditional) assignments, havocs, assertions, and assumes, as they are used in the product construction. For each, we show first that any such statement always has a terminating trace, and subsequently, that *all* traces of such statements end in configurations that have certain properties. It is necessary to make this distinction because our language is non-deterministic, and showing that there is one terminating trace that has certain properties is not equivalent to showing properties of *all* possible traces of a statement.

First, we have two lemmas about a sequence of k assignments.

Lemma A.1.10 *For all* x, e, k *and* $\underline{\sigma}$, $\langle \bigodot_{i=i_0}^{k} x^{(i)}{:=}e^{(i)}, \underline{\sigma} \rangle \longrightarrow^* \langle \texttt{skip}, \underline{\sigma}' \rangle$ *for some* $\underline{\sigma}'$.

Proof. If $i_0 > k$ then $(\bigodot_{i=i_0}^{k} s) \equiv \texttt{skip}$ and the conclusion holds trivially. Otherwise the proof goes by induction on $k - i_0$. □

Lemma A.1.11 *For all* $x, e, k, \dot{s}, \underline{\sigma}$ *and* $\underline{\sigma}'$, *if* $\langle \bigodot_{i=i_0}^{k} x^{(i)}{:=}e^{(i)}, \underline{\sigma} \rangle \longrightarrow^* \langle \dot{s}, \underline{\sigma}' \rangle$ *then* $\dot{s} = \texttt{skip}$ *and* $\underline{\sigma}' = \underline{\sigma}[x^{(i_0)} \mapsto v_{i_0}] \ldots [x^{(k)} \mapsto v_k]$ *and* $\forall i \in \{i_0, \ldots, k\}.\, e^{(i)} \Downarrow_{\underline{\sigma}} v_i$.

Proof. If $i_0 > k$ then $(\bigodot_{i=i_0}^{k} s) \equiv \texttt{skip}$ and the conclusion holds trivially. Otherwise the proof goes by induction on $k - i_0$, using Lemma A.1.9 and injectivity of the renaming function. □

Next, we have two lemmas about a sequence of k *conditional* assignments.

Lemma A.1.12 *Let* $A \subseteq \{1, \ldots, k\}$ *and* $\underline{\sigma}(p^{(i)}) = \top$ *for all* $i \in A$ *and* $\underline{\sigma}(p^{(i)}) = \bot$ *for all* $i \in \{1, \ldots, k\} \setminus A$.

Then $\langle \bigodot_{i=i_0}^{k} \texttt{if}\,(p^{(i)})\,\{x^{(i)}{:=}e^{(i)}\}, \underline{\sigma} \rangle \longrightarrow^* \langle \texttt{skip}, \underline{\sigma}' \rangle$ *for some* $\underline{\sigma}'$.

Proof. If $i_0 > k$ then $(\bigodot_{i=i_0}^{k} s) \equiv \texttt{skip}$ and the conclusion holds trivially. Otherwise the proof goes by induction on $k - i_0$. □

Lemma A.1.13 *Let* $A \subseteq \{1, \ldots, k\}$ *and* $\underline{\sigma}(p^{(i)}) = \top$ *for all* $i \in A$ *and* $\underline{\sigma}(p^{(i)}) = \bot$ *for all* $i \in \{1, \ldots, k\} \setminus A$.

Further assume that $\langle \bigodot_{i=i_0}^{k} \texttt{if}\,(p^{(i)})\,\{x^{(i)}{:=}e^{(i)}\}, \underline{\sigma} \rangle \longrightarrow^* \langle \dot{s}, \underline{\sigma}' \rangle$.

Then $\dot{s} = \texttt{skip}$ *and* $\underline{\sigma}' = \underline{\sigma}[x^{(j_1)} \mapsto v_{j_1}] \ldots [x^{(j_n)} \mapsto v_{j_n}]$ *for* $\{j_1, \ldots, j_n\} = A$ *s.t.* $j_i < j_{i+i}$ *and* $e^{(j_n)} \Downarrow_{\underline{\sigma}} v_{j_n}$.

Proof. If $i_0 > k$ then $(\bigodot_{i=i_0}^{k} s) \equiv \texttt{skip}$ and the conclusion holds trivially. Otherwise the proof goes by induction on $k - i_0$, using Lemma A.1.9 and injectivity of the renaming function. □

Now, we have the analogous two lemmas about a sequence of k conditional `havoc` statements, which of course behave like non-deterministic assignments.

Lemma A.1.14 *Let* $A \subseteq \{1, \ldots, k\}$ *and* $\underline{\sigma}(p^{(i)}) = \top$ *for all* $i \in A$ *and* $\underline{\sigma}(p^{(i)}) = \bot$ *for all* $i \in \{1, \ldots, k\} \setminus A$.

Then $\langle \bigodot_{i=i_0}^{k} \texttt{if}\,(p^{(i)})\,\{\texttt{havoc}\ x^{(i)}\}, \underline{\sigma} \rangle \to^{*} \langle \texttt{skip}, \underline{\sigma}' \rangle$ *for some* $\underline{\sigma}'$.

Proof. If $i_0 > k$ then $(\bigodot_{i=i_0}^{k} s) \equiv \texttt{skip}$ and the conclusion holds trivially. Otherwise the proof goes by induction on $k - i_0$. □

Lemma A.1.15 *Let* $A \subseteq \{1, \ldots, k\}$ *and* $\underline{\sigma}(p^{(i)}) = \top$ *for all* $i \in A$ *and* $\underline{\sigma}(p^{(i)}) = \bot$ *for all* $i \in \{1, \ldots, k\} \setminus A$.

Further assume that $\langle \bigodot_{i=i_0}^{k} \texttt{if}\,(p^{(i)})\,\{\texttt{havoc}\ x^{(i)}\}, \underline{\sigma} \rangle \to^{*} \langle \dot{s}, \underline{\sigma}' \rangle$.

Then $\dot{s} = \texttt{skip}$ *and* $\underline{\sigma}' = \underline{\sigma}[x^{(j_1)} \mapsto v_{j_1}] \ldots [x^{(j_n)} \mapsto v_{j_n}]$ *for* $\{j_1, \ldots, j_n\} = A$ *and any* $v_{j_1}, \ldots, v_{j_n}$.

Proof. If $i_0 > k$ then $(\bigodot_{i=i_0}^{k} s) \equiv \texttt{skip}$ and the conclusion holds trivially. Otherwise the proof goes by induction on $k - i_0$, using injectivity of the renaming function. □

Now two lemmas about a sequence of k `assert` statements.

Lemma A.1.16 $\langle \bigodot_{i=i_0}^{k} \texttt{if}\,(p^{(i)})\,\{\texttt{assert}\ e^{(i)}\}, \underline{\sigma} \rangle \to^{*} \langle \dot{s}, \underline{\sigma}' \rangle$ *for some* $\dot{s} \neq \texttt{error}$ *if and only if* $\underline{\sigma}' = \underline{\sigma}$ *and* $\dot{s} = \texttt{skip}$ *and for all* $i \in \{i_0, \ldots, k\}$ *s.t.* $p^{(i)} \Downarrow_{\underline{\sigma}} \top$ *we have* $e^{(i)} \Downarrow_{\underline{\sigma}} \top$.

Proof. Both directions of the biimplication are straightforward by induction on $k - i_0$. □

Lemma A.1.17 $\langle \bigodot_{i=i_0}^{k} \texttt{if}\,(p^{(i)})\,\{\texttt{assert}\ e^{(i)}\}, \underline{\sigma} \rangle \to^{*} \langle \texttt{error}, \underline{\sigma}' \rangle$ *if and only if* $\underline{\sigma}' = \underline{\sigma}$ *and for some non-empty set* $A \subseteq \{i_0, \ldots, k\}$ *we have* $\forall i \in A.\, p^{(i)} \Downarrow_{\underline{\sigma}} \top \wedge e^{(i)} \Downarrow_{\underline{\sigma}} \bot$.

Proof. Like for the previous lemma, both directions of the biimplication are proved by induction on $k - i_0$. □

And we have the analogous two lemmas about a sequence of k `assume` statements.

Lemma A.1.18 $\langle \bigodot_{i=i_0}^{k} \texttt{if}\,(p^{(i)})\,\{\texttt{assume}\ e^{(i)}\}, \underline{\sigma} \rangle \to^{*} \langle \dot{s}, \underline{\sigma}' \rangle$ *for some* $\dot{s} \neq \texttt{magic}$ *if and only if* $\underline{\sigma}' = \underline{\sigma}$ *and* $\dot{s} = \texttt{skip}$ *and for all* $i \in \{i_0, \ldots, k\}$ *s.t.* $p^{(i)} \Downarrow_{\underline{\sigma}} \top$ *we have* $e^{(i)} \Downarrow_{\underline{\sigma}} \top$.

Proof. Analogous to the proof of Lemma A.1.16, by induction on $k - i_0$. □

Lemma A.1.19 $\langle \bigodot_{i=i_0}^{k} \mathtt{if}\,(p^{(i)})\,\{\mathtt{assume}\; e^{(i)}\}, \underline{\sigma}\rangle \longrightarrow^{*} \langle \mathtt{magic}, \underline{\sigma}'\rangle$ *if and only if* $\underline{\sigma}' = \underline{\sigma}$ *and for some non-empty set* $A \subseteq \{i_0, \ldots, k\}$ *we have* $\forall i \in A.\, p^{(i)} \Downarrow_{\underline{\sigma}} \top \wedge e^{(i)} \Downarrow_{\underline{\sigma}} \bot$.

Proof. Analogous to the proof of Lemma A.1.17, by induction on $k - i_0$. □

A.1.2. Proof of Thm. 3.4.2

Theorem 3.4.2 *Assume that* $match(\Phi, \underline{\Phi})$ *and that for some sets* $A \subseteq \{1, \ldots, k\}$ *and* $I = \{1, \ldots, k\} \setminus A$ *we have that* $\forall i \in A.\, p^{(i)} \Downarrow_{\underline{\sigma}} \top \wedge \sigma_i \in_i \underline{\sigma}$ *and* $\forall i \in I.\, p^{(i)} \Downarrow_{\underline{\sigma}} \bot$. *Let* $\{p^{(1)}, \ldots, p^{(k)}\} \cap (RPV\textsc{ar} \cup freshvars(\underline{s})) = \emptyset$ *and* $\underline{s} = [\![s]\!]_k^{\mathring{p}}$ *and* $\langle \underline{s}, \underline{\sigma}\rangle \longrightarrow^{l} \langle \mathtt{skip}, \underline{\sigma}'\rangle$ *under* $\underline{\Phi}$.

Then

1. *for all* $i \in A$, $\langle s, \sigma_i\rangle \longrightarrow^{*} \langle \mathtt{skip}, \sigma_i'\rangle$ *under* Φ *for some* σ_i' *s.t.* $\sigma_i' \in_i \underline{\sigma}'$, *and* $\underline{\sigma} \precsim_i^{\underline{s}} \underline{\sigma}'$,
2. *for all* $i \in I$, $\underline{\sigma} \preccurlyeq_i^{\underline{s}} \underline{\sigma}'$.

Proof. By strong induction on the length l of the derivation $\langle [\![s]\!]_k^{\mathring{p}}, \underline{\sigma}\rangle \longrightarrow^{l} \langle \mathtt{skip}, \underline{\sigma}'\rangle$.

We perform a case split on the structure of s, which determines the structure of the product $\underline{s}$. Note that our theorem only applies to traces of statements that are modular products of some statements, not to arbitrary statements. We therefore generally perform as many steps of $\underline{s}$ as necessary to get to a configuration where the statement is again a product program, so that we can apply the induction hypothesis.

- ▸ $s \equiv x{:=}e$: Then $\underline{s}$ has the form $\bigodot_{i=1}^{k} \mathtt{if}\,(p^{(i)})\,\{x^{(i)}{:=}e^{(i)}\}$. By Lemma A.1.13, we have that $\underline{\sigma}' = \underline{\sigma}[x^{(j_1)} \mapsto v_{j_1}] \ldots [x^{(j_n)} \mapsto v_{j_n}]$ for $\{j_1, \ldots, j_n\} = A$ s.t. $e^{(j_n)} \Downarrow_{\underline{\sigma}} v_{j_n}$.
  For any $i \in A$, we then have for some o that $j_o = i$ and therefore $e^{(i)} \Downarrow_{\underline{\sigma}} v_i$. Since we also have that $fv(e) \subseteq$ PV<span style="font-variant:small-caps">ar</span>, by Lemma 3.4.1 we have that $e \Downarrow_{\sigma_i} v_i$. Then by Assign, we get that $\langle x{:=}e, \sigma_i\rangle \longrightarrow \langle \mathtt{skip}, \sigma_i[x \mapsto v_i]\rangle$.
  We therefore have to show that $\sigma_i[x \mapsto v_i] \in_i \underline{\sigma}[x^{(j_1)} \mapsto v_{j_1}] \ldots [x^{(j_o)} \mapsto v_{j_o}] \ldots [x^{(j_n)} \mapsto v_{j_n}]$:

  $$\begin{aligned}
  &\sigma_i \in_i \underline{\sigma} \\
  \Rightarrow &\sigma_i \in_i \underline{\sigma}[x^{(j_1)} \mapsto v_{j_1}] \ldots [x^{(j_{o-1})} \mapsto v_{j_{o-1}}] && \text{(Prop. A.1.2)} \\
  \Rightarrow &\sigma_i[x \mapsto v_i] \in_i \underline{\sigma}[x^{(j_1)} \mapsto v_{j_1}] \ldots [x^{(j_o)} \mapsto v_{j_o}] && \text{(Prop. A.1.1)} \\
  \Rightarrow &\sigma_i[x \mapsto v_i] \in_i \underline{\sigma}[x^{(j_1)} \mapsto v_{j_1}] \ldots [x^{(j_o)} \mapsto v_{j_o}] \ldots [x^{(j_n)} \mapsto v_{j_n}] \\
  & && \text{(Prop. A.1.2)}
  \end{aligned}$$

  Since $x^{(i)} \in$ RPV<span style="font-variant:small-caps">ar</span>, we also have $\underline{\sigma} \precsim_i^{\underline{s}} \underline{\sigma}'$ by Prop. A.1.5 (a).
  For all $i \in I$, we have that $\underline{\sigma} \preccurlyeq_i^{\underline{s}} \underline{\sigma}'$ by Prop. A.1.4 (b).
- ▸ $s \equiv \mathtt{havoc}\; x$: Largely analogous to the previous case.
  $\underline{s}$ must have the form $\bigodot_{i=1}^{k} \mathtt{if}\,(p^{(i)})\,\{\mathtt{havoc}\; x^{(i)}\}$. By Lemma A.1.15, we have that $\underline{\sigma}' = \underline{\sigma}[x^{(j_1)} \mapsto v_{j_1}] \ldots [x^{(j_n)} \mapsto v_{j_n}]$ for $\{j_1, \ldots, j_n\} = A$ and some $v_{j_1}, \ldots, v_{j_n}$. For any $i \in A$, we then have for some

o that $j_o = i$. Then by Havoc, choosing the value v_i, we get that $\langle \texttt{havoc}\, x, \sigma_i \rangle \rightarrow \langle \texttt{skip}, \sigma_i[x \mapsto v_i] \rangle$. By the same argument as above, we get that $\sigma_i[x \mapsto v_i] \in_i \underline{\sigma}[x^{(j_1)} \mapsto v_{j_1}] \ldots [x^{(j_o)} \mapsto v_{j_o}] \ldots [x^{(j_n)} \mapsto v_{j_n}]$. Since $x^{(i)} \in \text{RPVar}$, we also have $\underline{\sigma} \precsim_i^{s} \underline{\sigma}'$ by Prop. A.1.5 (a).
For all $i \in I$, we have that $\underline{\sigma} \preccurlyeq_i^{s} \underline{\sigma}'$ by Prop. A.1.4, since there is no j_o s.t. $j_o = i$ (b).

- $s \equiv \texttt{skip}$: Then we must have that $\underline{s} = \texttt{skip}$, and therefore $\underline{\sigma}' = \underline{\sigma}$. Additionally, for all $i \in A$, we have $\sigma_i = \sigma_i'$ and therefore $\sigma_i' \in_i \underline{\sigma}'$, and we have $\underline{\sigma} \precsim_i^{s} \underline{\sigma}'$ by reflexivity of $\preccurlyeq_i^{V}$ (a). Similarly, we have $\underline{\sigma} \preccurlyeq_i^{s} \underline{\sigma}'$ for all $i \in I$ because of reflexivity of $\preccurlyeq_i^{V}$ (b).
- $s \equiv \texttt{assert}\, e$: Then $\underline{s}$ has the form $\bigodot_{i=1}^{k} \texttt{if}\,(p^{(i)})\,\{\texttt{assert}\, e^{(i)}\}$. By Lemma A.1.16, we have $\underline{\sigma}' = \underline{\sigma}$ and for all $i \in A$ that $e^{(i)} \Downarrow_{\underline{\sigma}} \top$. Then for all such i, since $\mathit{fv}(e) \subseteq \text{PVar}$, we have $e \Downarrow_{\sigma_i} \top$ by Lemma 3.4.1. Then by Assert1 we can construct $\langle \texttt{assert}\, e, \sigma_i \rangle \rightarrow \langle \texttt{skip}, \sigma_i \rangle$. Therefore $\sigma_i' = \sigma_i$ which trivially implies that $\sigma_i' \in_i \underline{\sigma}'$. Additionally, we have $\underline{\sigma} \precsim_i^{s} \underline{\sigma}'$ by reflexivity of $\preccurlyeq_i^{V}$ (a).
For all $i \in I$, we have $\underline{\sigma} \preccurlyeq_i^{s} \underline{\sigma}'$ by reflexivity of $\preccurlyeq_i^{V}$ (b).
- $s \equiv \texttt{assume}\, e$: This case is analogous to the previous case, using Lemma A.1.18 and Assume1 instead of Lemma A.1.16 and Assert1.
- $s \equiv \texttt{if}\,(e)\,\{s_1\}\,\texttt{else}\,\{s_2\}$:
Then $\underline{s} = \bigodot_{i=1}^{k}(t^{(i)}{:=}p^{(i)} \land e^{(i)}); \bigodot_{i=1}^{k}(f^{(i)}{:=}p^{(i)} \land \neg e^{(i)}); \underline{s_1}; \underline{s_2}$, where $\underline{s_1} = \llbracket s_1 \rrbracket_k^{\mathring{t}}$ and $\underline{s_2} = \llbracket s_2 \rrbracket_k^{\mathring{f}}$ and $\mathit{freshvars}(\underline{s}) = \{\mathring{t}, \mathring{f}\} \cup \mathit{freshvars}(\underline{s_1}) \cup \mathit{freshvars}(\underline{s_2})$.
Then by repeated application of Lemma A.1.7.1 we must have that

$$\begin{aligned} &\langle \bigodot_{i=1}^{k}(t^{(i)}{:=}p^{(i)} \land e^{(i)}); \bigodot_{i=1}^{k}(f^{(i)}{:=}p^{(i)} \land \neg e^{(i)}); \underline{s_1}; \underline{s_2}, \underline{\sigma} \rangle \\ &\rightarrow^{l_1} \langle \texttt{skip}; \bigodot_{i=1}^{k}(f^{(i)}{:=}p^{(i)} \land \neg e^{(i)}); \underline{s_1}; \underline{s_2}, \underline{\sigma}'' \rangle && \text{(for some } l_1, \underline{\sigma}''\text{)} \\ &\rightarrow^{1} \langle \bigodot_{i=1}^{k}(f^{(i)}{:=}p^{(i)} \land \neg e^{(i)}); \underline{s_1}; \underline{s_2}, \underline{\sigma}'' \rangle \\ &\rightarrow^{l_2} \langle \texttt{skip}; \underline{s_1}; \underline{s_2}, \underline{\sigma}''' \rangle && \text{(for some } l_2, \underline{\sigma}'''\text{)} \\ &\rightarrow^{1} \langle \underline{s_1}; \underline{s_2}, \underline{\sigma}''' \rangle \\ &\rightarrow^{l_3} \langle \texttt{skip}; \underline{s_2}, \underline{\sigma}'''' \rangle \\ &\rightarrow^{1} \langle \underline{s_2}, \underline{\sigma}'''' \rangle && \text{(for some } l_3, \underline{\sigma}''''\text{)} \\ &\rightarrow^{l_4} \langle \texttt{skip}, \underline{\sigma}' \rangle && \text{(for some } l_4\text{)} \end{aligned}$$

s.t. $l = l_1 + l_2 + l_3 + l_4 + 3$ and all $l_i \geq 0$.
We first identify properties of $\underline{\sigma}'''$ in order to be able to apply the induction hypothesis. By Lemma A.1.7.2, we get that $\langle \bigodot_{i=1}^{k}(t^{(i)}{:=}p^{(i)} \land e^{(i)}), \underline{\sigma} \rangle \rightarrow^{l_1} \langle \texttt{skip}, \underline{\sigma}'' \rangle$. By Lemma A.1.11 $\underline{\sigma}'' = \underline{\sigma}[t^{(1)} \mapsto v_1] \ldots [t^{(k)} \mapsto v_k]$ where $\forall i \in \{1, \ldots, k\}.\, p^{(i)} \land e^{(i)} \Downarrow_{\underline{\sigma}} v_i$. By the same argument, we get that $\underline{\sigma}''' = \underline{\sigma}''[f^{(1)} \mapsto v_1'] \ldots [f^{(k)} \mapsto v_k']$ where $\forall i \in \{1, \ldots, k\}.\, p^{(i)} \land \neg e^{(i)} \Downarrow_{\underline{\sigma}''} v_i'$. By Lemma A.1.9, we also have $\forall i \in \{1, \ldots, k\}.\, p^{(i)} \land \neg e^{(i)} \Downarrow_{\underline{\sigma}'} v_i'$, since $\mathring{t}$ and $\mathring{f}$ are not in $\mathit{fv}(p^{(i)} \land \neg e^{(i)})$. Since $\mathring{t}$ and $\mathring{f}$ are in $\mathit{freshvars}(\underline{s})$, by Prop. A.1.5 we have $\underline{\sigma} \precsim_i^{s} \underline{\sigma}'''$ for all $i \in \{1, \ldots, k\}$. Therefore by Lemma A.1.6 we have that $\sigma_i \in_i \underline{\sigma}'''$ for all $i \in A$.

Let $A_1 = \{i | i \in A \wedge e^{(i)} \Downarrow_{\underline{\sigma}} \top\}$ and $A_2 = \{i | i \in A \wedge e^{(i)} \Downarrow_{\underline{\sigma}} \bot\}$. Note that $A = A_1 \cup A_2$. Let $I_1 = I \cup A_2$ and $I_2 = I \cup A_1$. By the rules for expression evaluation, we have that for all $i \in A_1, p^{(i)} \wedge e^{(i)} \Downarrow_{\underline{\sigma}} \top$ and therefore $t^{(i)} \Downarrow_{\underline{\sigma}'''} \top$, and for all $i \in I_1$, $p^{(i)} \wedge e^{(i)} \Downarrow_{\underline{\sigma}} \bot$ and therefore $t^{(i)} \Downarrow_{\underline{\sigma}'''} \bot$. Likewise, we have for all $i \in A_2$, $p^{(i)} \wedge \neg e^{(i)} \Downarrow_{\underline{\sigma}} \top$ and therefore $f^{(i)} \Downarrow_{\underline{\sigma}'''} \top$ and for all $i \in I_2$, $p^{(i)} \wedge \neg e^{(i)} \Downarrow_{\underline{\sigma}} \bot$ and therefore $f^{(i)} \Downarrow_{\underline{\sigma}'''} \bot$.

We can now apply the induction hypothesis twice. By Lemma A.1.7.2, $\langle \underline{s_1}, \underline{\sigma}''' \rangle \rightarrow^{l_3} \langle \texttt{skip}, \underline{\sigma}'''' \rangle$. Then for $\underline{s_1}, \underline{\sigma}''', A_1, I_1$ we may assume the induction hypothesis (IH1) and get that for all $i \in A_1$, $\underline{\sigma}''' \precsim^{s_1}_{i} \underline{\sigma}''''$, and for all $i \in I_1$, $\underline{\sigma}''' \preccurlyeq^{s_1}_{i} \underline{\sigma}''''$. Therefore we have that for all $i \in \{1, \dots, k\}$, $\underline{\sigma}''''(f^{(i)}) = \underline{\sigma}'''(f^{(i)})$. Therefore for all $i \in A_2$, $\underline{\sigma}''''(f^{(i)}) = \top$ and by Lemma A.1.6 $\sigma_i \in_i \underline{\sigma}''''$, and for all $i \in I_2$, $\underline{\sigma}''''(f^{(i)}) = \bot$. Therefore we may assume the induction hypothesis for $\underline{s_2}, \underline{\sigma}'''', A_2, I_2$ (IH2).

Based on this, we first show (b). We have for all $i \in I$ that $\underline{\sigma}(p^{(i)}) = \bot$, and since $I \subseteq I_1$ and $I \subseteq I_2$, $\underline{\sigma}'''(t^{(i)}) = \bot$ and $\underline{\sigma}'''(f^{(i)}) = \bot$. By (IH1), we have that $\underline{\sigma}''' \preccurlyeq^{s_1}_{i} \underline{\sigma}''''$ and therefore $\underline{\sigma}''''(f^{(i)}) = \bot$. By (IH2), we have that $\underline{\sigma}'''' \preccurlyeq^{s_2}_{i} \underline{\sigma}'$. Since $t^{(i)} \in \mathit{freshvars}(\underline{s})$ and $f^{(i)} \in \mathit{freshvars}(\underline{s})$, we have $\underline{\sigma} \preccurlyeq^{s}_{i} \underline{\sigma}'''$ by Prop. A.1.5. By Prop. A.1.3, we get $\underline{\sigma} \preccurlyeq^{s}_{i} \underline{\sigma}'$.

We proceed to show (a) for all $i \in A_1$. For all $i \in A_1$, we have $e^{(i)} \Downarrow_{\underline{\sigma}} \top$ and $\underline{\sigma}'''(t^{(i)}) = \top$ and $\underline{\sigma}'''(f^{(i)}) = \bot$. Since $A_1 \subseteq A$, we also have $\sigma_i \in_i \underline{\sigma}$. Since $\sigma_i \in_i \underline{\sigma}'''$, by Lemma 3.4.1 we get $e \Downarrow_{\sigma_i} \top$. By Cond1, we get that $\langle s, \sigma_i \rangle \rightarrow \langle s_1, \sigma_i \rangle$. By (IH1), we get $\langle s_1, \sigma_i \rangle \rightarrow^* \langle \texttt{skip}, \sigma_i'''' \rangle$ for some σ_i'''' s.t. $\sigma_i'''' \in_i \underline{\sigma}''''$, and $\underline{\sigma}''''(f^{(i)}) = \bot$. By (IH2), we have $\underline{\sigma}'''' \preccurlyeq^{s_2}_{i} \underline{\sigma}'$, and because of Lemma A.1.6, this implies $\sigma_i' \in_i \underline{\sigma}'$. We still need to show that $\underline{\sigma} \precsim^{s}_{i} \underline{\sigma}'$: Since $\underline{\sigma} \preccurlyeq^{s}_{i} \underline{\sigma}'''$, by (IH1) we have that $\underline{\sigma}''' \precsim^{s_1}_{i} \underline{\sigma}''''$. By (IH2) we have $\underline{\sigma}'''' \preccurlyeq^{s_2}_{i} \underline{\sigma}'$. Then by using Prop. A.1.3 twice, we get $\underline{\sigma} \precsim^{s}_{i} \underline{\sigma}'$.

We now show (a) for all $i \in A_2$, after which we are done because $A = A_1 \cup A_2$. For all $i \in A_2$ we have $e^{(i)} \Downarrow_{\underline{\sigma}} \bot$ and $\underline{\sigma}'''(t^{(i)}) = \bot$ and $\underline{\sigma}'''(f^{(i)}) = \top$. Since $A_2 \subseteq A$, we also have $\sigma_i \in_i \underline{\sigma}$. Since $\sigma_i \in_i \underline{\sigma}'''$, by Lemma 3.4.1 we get that $e \Downarrow_{\sigma_i} \bot$. Therefore by Cond2 $\langle s, \sigma_i \rangle \rightarrow \langle s_2, \sigma_i \rangle$. By (IH1), $\underline{\sigma}''' \preccurlyeq^{s_1}_{i} \underline{\sigma}''''$, which implies $\underline{\sigma}''''(f^{(i)}) = \top$ and $\sigma_i \in_i \underline{\sigma}''''$ by Lemma A.1.6. Then by (IH2), $\langle s_2, \sigma_i \rangle \rightarrow^* \langle \texttt{skip}, \sigma_i' \rangle$ for some σ_i' s.t. $\sigma_1' \in_i \underline{\sigma}'$. Additionally, as above, we have $\underline{\sigma} \preccurlyeq^{s}_{i} \underline{\sigma}'''$. By (IH1), we get $\underline{\sigma}''' \preccurlyeq^{s_1}_{i} \underline{\sigma}''''$, and by (IH2), $\underline{\sigma}'''' \precsim^{s_2}_{i} \underline{\sigma}'$. Again, we get $\underline{\sigma} \precsim^{s}_{i} \underline{\sigma}'$ by twice using Prop. A.1.3.

▸ $s \equiv \texttt{while}\ (e)\ \{s_l\}$:

Then $\underline{s}$ must have the form $\texttt{while}\ (\bigvee_{i=1}^{k} (p^{(i)} \wedge e^{(i)}))\ \{\bigodot_{i=1}^{k} (t^{(i)} := p^{(i)} \wedge e^{(i)}); \underline{s_l}\}$ where $\underline{s_l} = [\![s_l]\!]^{\mathring{t}}_{k}$ and $\mathit{freshvars}(\underline{s}) = \{\mathring{t}\} \cup \mathit{freshvars}(\underline{s_l})$. $\underline{s}$ must therefore progress by either Whl1 or Whl2.

- If it progresses by Whl2, then $\bigvee_{i=1}^{k} (p^{(i)} \wedge e^{(i)}) \Downarrow_{\underline{\sigma}} \bot$ and $\langle \underline{s}, \underline{\sigma} \rangle \rightarrow \langle \texttt{skip}, \underline{\sigma} \rangle$ and therefore $\underline{\sigma}' = \underline{\sigma}$. Then by the rules of expression evaluation, for all $i \in \{1, \dots, k\}$ we must have that $p^{(i)} \wedge e^{(i)} \Downarrow_{\underline{\sigma}} \bot$, and therefore for all $i \in A$, we have that $e^{(i)} \Downarrow_{\underline{\sigma}} \bot$. By Lemma 3.4.1, since $\mathit{fv}(e) \subseteq$ PVar, we have $e \Downarrow_{\sigma_i} \bot$. Then using Whl2 we can construct $\langle \texttt{while}\ (e)\ \{s_l\}, \sigma_i \rangle \rightarrow \langle \texttt{skip}, \sigma_i \rangle$. Then we have $\sigma_i' = \sigma_i$ and therefore trivially $\sigma_i' \in_i$

$\underline{\sigma}'$ as well as $\underline{\sigma} \precsim_i^{\underline{s}} \underline{\sigma}'$ by reflexivity of $\preccurlyeq_i^V$. Similarly, for all $i \in I$, we have $\underline{\sigma} \preccurlyeq_i^{\underline{s}} \underline{\sigma}'$ because of reflexivity of $\preccurlyeq_i^V$, and we are done.

- If the product progresses by Whl1 then $\bigvee_{i=1}^k (p^{(i)} \wedge e^{(i)}) \Downarrow_{\underline{\sigma}} \top$ and $\langle \underline{s}, \underline{\sigma} \rangle \rightarrow \langle \bigodot_{i=1}^k (t^{(i)}{:=}p^{(i)} \wedge e^{(i)}); \underline{s_l}; \underline{s}, \underline{\sigma} \rangle$.
  We must then have that $\langle \bigodot_{i=1}^k (t^{(i)}{:=}p^{(i)} \wedge e^{(i)}); \underline{s_l}; \underline{s}, \underline{\sigma} \rangle \rightarrow^{l-1} \langle \texttt{skip}, \underline{\sigma}' \rangle$ and by repeated application of Lemma A.1.7.1, we must have

$$\begin{aligned}
&\langle \bigodot_{i=1}^{k} (t^{(i)}{:=}p^{(i)} \wedge e^{(i)}); \underline{s_l}; \underline{s}, \underline{\sigma} \rangle \\
&\rightarrow^{l_1} \langle \texttt{skip}; \underline{s_l}; \underline{s}, \underline{\sigma}'' \rangle && \text{(for some } l_1, \underline{\sigma}''\text{)} \\
&\rightarrow^{1} \langle \underline{s_l}; \underline{s}, \underline{\sigma}'' \rangle \\
&\rightarrow^{l_2} \langle \texttt{skip}; \underline{s}, \underline{\sigma}''' \rangle && \text{(for some } l_2, \underline{\sigma}'''\text{)} \\
&\rightarrow^{1} \langle \underline{s}, \underline{\sigma}''' \rangle \\
&\rightarrow^{l_3} \langle \texttt{skip}, \underline{\sigma}' \rangle && \text{(for some } l_3\text{)}
\end{aligned}$$

  s.t. $l = 3 + l_1 + l_2 + l_3$ and all $l_i \geq 0$.
  Our goal is to apply the induction hypothesis twice, once to the execution of $\underline{s_l}$ and once to the execution of $\underline{s}$ from $\underline{\sigma}'''$. By Lemma A.1.7.2 we get $\langle \bigodot_{i=1}^k (t^{(i)}{:=}p^{(i)} \wedge e^{(i)}), \underline{\sigma} \rangle \rightarrow^{l_1} \langle \texttt{skip}, \underline{\sigma}'' \rangle$. Therefore by Lemma A.1.11 we get that $\underline{\sigma}'' = \underline{\sigma}[t^{(1)} \mapsto v_1] \ldots [t^{(k)} \mapsto v_k]$ and $\forall i \in \{1, \ldots, k\}. t^{(i)} \Downarrow_{\underline{\sigma}} v_i$. By Prop. A.1.5 we have that $\underline{\sigma} \preccurlyeq^{\underline{s}} \underline{\sigma}''$, since $\mathring{t}$ are in *freshvars*($\underline{s}$).
  Let $A_l = \{i | i \in A \wedge e^{(i)} \Downarrow_{\underline{\sigma}} \top\}$ and $I_l = \{1, \ldots, k\} \setminus A_l$. We have that for all $i \in A_l$, $p^{(i)} \wedge e^{(i)} \Downarrow_{\underline{\sigma}} \top$ and therefore $\underline{\sigma}''(t^{(i)}) = \top$. Additionally, since $\underline{\sigma} \preccurlyeq^{\underline{s}} \underline{\sigma}''$ we have $\sigma_i \in_i \underline{\sigma}''$ by Lemma A.1.6. For all $i \in I_l$, $\underline{\sigma}''(t^{(i)}) = \bot$. By Lemma A.1.7.2, we have $\langle \underline{s_l}, \underline{\sigma}'' \rangle \rightarrow^{l_2} \langle \texttt{skip}, \underline{\sigma}''' \rangle$ and can assume the induction hypothesis for this and A_l (IH1), from which we get that $\underline{\sigma}'' \precsim_i^{\underline{s_l}} \underline{\sigma}'''$ for all $i \in A_l$ and $\underline{\sigma}'' \preccurlyeq_i^{\underline{s_l}} \underline{\sigma}'''$ for all I_l. As a result, for all $i \in A$, $\underline{\sigma}'''(p^{(i)}) = \top$, and for all $i \in I$, $\underline{\sigma}'''(p^{(i)}) = \bot$. Subsequently, we can assume the induction hypothesis for $\langle \underline{s}, \underline{\sigma}''' \rangle \rightarrow^{l_3} \langle \texttt{skip}, \underline{\sigma}' \rangle$ and A (IH2).
  We first show (b). For all $i \in I$ we have that $p^{(i)} \wedge e^{(i)} \Downarrow_{\underline{\sigma}} \bot$ and therefore $\underline{\sigma}''(t^{(i)}) = \bot$. By (IH1), we get that $\underline{\sigma}'' \preccurlyeq_i^{\underline{s_l}} \underline{\sigma}'''$ and $\underline{\sigma}'''(p^{(i)}) = \bot$. By (IH2) we get that $\underline{\sigma}''' \preccurlyeq_i^{\underline{s}} \underline{\sigma}'$. Since *freshvars*($\underline{s_l}$) $\subseteq$ *freshvars*($\underline{s}$), by Prop. A.1.3, $\underline{\sigma} \preccurlyeq_i^{\underline{s}} \underline{\sigma}'$.
  We now show (a) for all $i \in A_l$. For all $i \in A_l$ we have $e^{(i)} \Downarrow_{\underline{\sigma}} \top$. By Lemma 3.4.1, $e \Downarrow_{\sigma_i} \top$, and therefore by Whl1, $\langle s, \sigma_i \rangle \rightarrow \langle s_l; s, \sigma_i \rangle$. Since $p^{(i)} \wedge e^{(i)} \Downarrow_{\underline{\sigma}} \top$ and therefore $\underline{\sigma}''(t^{(i)}) = \top$, we get from (IH1) that $\langle s_l, \sigma_i \rangle \rightarrow^* \langle \texttt{skip}, \sigma_i''' \rangle$ for some σ_i''' s.t. $\sigma_i''' \in_i \underline{\sigma}'''$, and that $\underline{\sigma}'' \precsim_i^{\underline{s_l}} \underline{\sigma}'''$ and therefore $\underline{\sigma}'''(p^{(i)}) = \top$. Then by (IH2) we get $\langle s, \sigma_i''' \rangle \rightarrow^* \langle \texttt{skip}, \sigma_i' \rangle$ s.t. $\sigma_i' \in_i \underline{\sigma}'$. We can construct $\langle s, \sigma_i \rangle \rightarrow \langle s_l; s, \sigma_i \rangle \rightarrow^* \langle \texttt{skip}; s, \sigma_i''' \rangle \rightarrow \langle s, \sigma_i''' \rangle \rightarrow^* \langle \texttt{skip}, \sigma_i' \rangle$. Additionally, we have $\underline{\sigma} \preccurlyeq_i^{\underline{s}} \underline{\sigma}''$ and $\underline{\sigma}'' \precsim_i^{\underline{s_l}} \underline{\sigma}'''$ and $\underline{\sigma}''' \precsim_i^{\underline{s}} \underline{\sigma}'$, which by Prop. A.1.3 gives us $\underline{\sigma} \precsim_i^{\underline{s}} \underline{\sigma}'$.
  We now show (a) for all $i \in A \setminus A_l$. For all $i \in A \setminus A_l$ we

have $\underline{\sigma}(p^{(i)}) = \top$ and $e^{(i)} \Downarrow_{\underline{\sigma}} \bot$, and therefore $\underline{\sigma}''(t^{(i)}) = \bot$. Therefore by (IH1) we have $\underline{\sigma}'' \preccurlyeq_i^{s_l} \underline{\sigma}'''$, which implies $\underline{\sigma}'''(p^{(i)}) = \top$. By Lemma A.1.6, since $\sigma_i \in_i \underline{\sigma}''$, we get that $\sigma_i \in_i \underline{\sigma}'''$. Then by (IH2), $\langle s, \sigma_i \rangle \longrightarrow^* \langle \texttt{skip}, \sigma_i' \rangle$. Additionally, since $\underline{\sigma} \preccurlyeq_i^{s} \underline{\sigma}''$ and $\underline{\sigma}'' \preccurlyeq_i^{s_l} \underline{\sigma}'''$ and $\underline{\sigma}''' \precsim_i^{s} \underline{\sigma}'$, by Prop. A.1.3 we get $\underline{\sigma} \precsim_i^{s} \underline{\sigma}'$.

► $s \equiv x_1, \ldots, x_m := m(e_1, \ldots, e_n)$: Then $\underline{s}$ has the form

$$\begin{array}{l}
\texttt{if}\quad (\bigvee_{i=1}^{k} p^{(i)})\ \{\\
\qquad \bigodot_{i=1}^{k} \texttt{if}\,(p^{(i)})\ \{\bigodot_{j=1}^{n}(a_j{}^{(i)}{:=}e_j{}^{(i)})\};\\
\qquad ts{:=}\ m(p^{(1)}, \ldots, p^{(k)}, as);\\
\qquad \bigodot_{i=1}^{k} \texttt{if}\,(p^{(i)})\ \{\bigodot_{j=1}^{m}(x_j{}^{(i)}{:=}t_j{}^{(i)})\}\\
\}
\end{array}$$

where $\mathring{a}_j$ and $\mathring{t}_j$ are in *freshvars*($\underline{s}$) and $ts = [t_1{}^{(1)}, \ldots, t_1{}^{(k)}, \ldots, t_m{}^{(1)}, \ldots, t_m{}^{(k)}]$ and $as = [a_1{}^{(1)}, \ldots, a_1{}^{(k)}, \ldots, a_n{}^{(1)}, \ldots, a_n{}^{(k)}]$.
$\underline{s}$ can proceed either by Cond1 or by Cond2.

- If it proceeds by Cond2, we have that $\bigvee_{i=1}^{k} p^{(i)} \Downarrow_{\underline{\sigma}} \bot$ and $\langle \underline{s}, \underline{\sigma} \rangle \longrightarrow \langle \texttt{skip}, \underline{\sigma} \rangle$, and therefore $\underline{\sigma}' = \underline{\sigma}$. By the rules of expression evaluation, $\bigvee_{i=1}^{k} p^{(i)} \Downarrow_{\underline{\sigma}} \bot$ implies that for all $i \in \{1, \ldots, k\}$ we have $p^{(i)} \Downarrow_{\underline{\sigma}} \bot$, which implies that $A = \emptyset$. (a) is therefore trivially true and we must only show (b), which we get by reflexivity of $\preccurlyeq_i^{V}$.
- If the product proceeds by Cond1 then $\bigvee_{i=1}^{k} p^{(i)} \Downarrow_{\underline{\sigma}} \top$ and we must have the following by repeated use of Lemma A.1.7.1:

$$\begin{array}{l}
\langle \underline{s}, \underline{\sigma} \rangle\\
\longrightarrow^{1} \left\langle \begin{array}{l} \bigodot_{i=1}^{k} \texttt{if}\,(p^{(i)})\ \{\bigodot_{j=1}^{n}(a_j{}^{(i)}{:=}e_j{}^{(i)})\};\\ ts{:=}\ m(p^{(1)}, \ldots, p^{(k)}, as);\\ \bigodot_{i=1}^{k} \texttt{if}\,(p^{(i)})\ \{\bigodot_{j=1}^{m}(x_j{}^{(i)}{:=}t_j{}^{(i)})\} \end{array}, \underline{\sigma} \right\rangle\\
\longrightarrow^{l_1} \left\langle \begin{array}{l} \texttt{skip};\\ ts{:=}\ m(p^{(1)}, \ldots, p^{(k)}, as);\\ \bigodot_{i=1}^{k} \texttt{if}\,(p^{(i)})\ \{\bigodot_{j=1}^{m}(x_j{}^{(i)}{:=}t_j{}^{(i)})\} \end{array}, \underline{\sigma}'' \right\rangle\\
\hfill \text{(for some } l_1, \underline{\sigma}''\text{)}\\
\longrightarrow^{1} \left\langle \begin{array}{l} ts{:=}\ m(p^{(1)}, \ldots, p^{(k)}, as);\\ \bigodot_{i=1}^{k} \texttt{if}\,(p^{(i)})\ \{\bigodot_{j=1}^{m}(x_j{}^{(i)}{:=}t_j{}^{(i)})\} \end{array}, \underline{\sigma}'' \right\rangle\\
\longrightarrow^{l_2} \left\langle \begin{array}{l} \texttt{skip};\\ \bigodot_{i=1}^{k} \texttt{if}\,(p^{(i)})\ \{\bigodot_{j=1}^{m}(x_j{}^{(i)}{:=}t_j{}^{(i)})\} \end{array}, \underline{\sigma}''' \right\rangle\\
\hfill \text{(for some } l_2, \underline{\sigma}'''\text{)}\\
\longrightarrow^{1} \langle \bigodot_{i=1}^{k} \texttt{if}\,(p^{(i)})\ \{\bigodot_{j=1}^{m}(x_j{}^{(i)}{:=}t_j{}^{(i)})\}, \underline{\sigma}''' \rangle\\
\longrightarrow^{l_3} \langle \texttt{skip}, \underline{\sigma}' \rangle \hfill \text{(for some } l_3\text{)}
\end{array}$$

where $l = 3 + l_1 + l_2 + l_3$, and all $l_i \geq 0$.
By Lemma A.1.7.2 and Lemma A.1.13 we have that for all $i \in A$ and $1 \leq j \leq n$, $\underline{\sigma}''(a_j{}^{(i)}) = v_{i,j}$ for some $v_{i,j}$ s.t. $e_j{}^{(i)} \Downarrow_{\underline{\sigma}} v_{i,j}$. Moreover, since $a_1, \ldots, a_n$ are in *freshvars*($\underline{s}$), we have $\underline{\sigma} \preccurlyeq^{\underline{s}} \underline{\sigma}''$ by Prop. A.1.5.

Let $\Phi(m) = ([q_1, \ldots, q_n], [r_1, \ldots, r_m], s_m)$. Then because of $\mathit{match}(\Phi, \underline{\Phi})$, we know $\underline{\Phi}(m) = ([p'^{(1)}, \ldots, p'^{(k)}, {q_1}^{(1)}, \ldots, {q_1}^{(k)}, \ldots, {q_n}^{(1)}, \ldots, {q_n}^{(k)}], rs, \underline{s_m})$ where $\underline{s_m} = [\![s_m]\!]_k^{\mathring{p}'}$ and $rs = [{r_1}^{(1)}, \ldots, {r_1}^{(k)}, \ldots, {r_m}^{(1)}, \ldots, {r_m}^{(k)}]$.
By Lemma A.1.7.2 and Call we must have that

$$\begin{aligned}
&\langle ts\!:= m(p^{(1)}, \ldots, p^{(k)}, as), \underline{\sigma}''\rangle \\
\rightarrow&\langle ts\!:=\!\texttt{frame}_{rs}(\underline{s_m}, \underline{\sigma}_f), \underline{\sigma}''\rangle \\
\rightarrow^{l_2-1}&\langle \texttt{skip}, \underline{\sigma}''\rangle
\end{aligned}$$

where

$$\begin{aligned}
\underline{\sigma}_f = [&p'^{(1)} \mapsto \underline{\sigma}''(p^{(1)}), \ldots, p'^{(k)} \mapsto \underline{\sigma}''(p^{(k)}), \\
&{q_1}^{(1)} \mapsto v_{1,1}, \ldots, {q_n}^{(1)} \mapsto v_{1,n}, \ldots, \\
&{q_1}^{(k)} \mapsto v_{k,1}, \ldots, {q_n}^{(k)} \mapsto v_{k,n}]
\end{aligned}$$

and ${a_j}^{(i)} \Downarrow_{\underline{\sigma}''} v_{i,j}$.
For all $i \in A$, since $\underline{\sigma}({a_j}^{(i)}) = v_{i,j}$ and therefore ${e_j}^{(i)} \Downarrow_{\underline{\sigma}} v_{i,j}$, we have by Lemma 3.4.1 that $e_j \Downarrow_{\sigma_i} v_{i,j}$. By Call, we have $\langle s, \sigma_i\rangle \rightarrow \langle x_1, \ldots, x_m\!:=\!\texttt{frame}_{r_1,\ldots,r_m}(s_m, \sigma_{(i,f)}), \sigma_i\rangle$, where $\sigma_{(i,f)} = [q_1 \mapsto v_{i,1}, \ldots, q_n \mapsto v_{i,n}]$, for all $i \in A$.
We now show that for all $i \in A$ we have $\sigma_{(i,f)} \in_i \underline{\sigma}_f$; we denote the empty store as [] and abbreviate $\underline{\sigma}_{p'} = [p'^{(1)} \mapsto \_, \ldots, p'^{(k)} \mapsto \_]$:

$$\begin{aligned}
&[] \in_i [] \wedge [] \preccurlyeq_i^{s_m} [] && \text{(by definitions of } \in_i \text{ and } \preccurlyeq_i^V\text{)} \\
\Rightarrow&[] \in_i [] \wedge [] \preccurlyeq_i^{s_m} [][p'^{(1)} \mapsto \underline{\sigma}''(p^{(1)}), \ldots, p'^{(k)} \mapsto \underline{\sigma}''(p^{(k)})] \\
& && \text{(Prop. A.1.5)} \\
\Rightarrow&[] \in_i \underline{\sigma}_{p'} && \text{(Lemma A.1.6)} \\
\Rightarrow&[] \in_i \underline{\sigma}_{p'}[{q_1}^{(1)} \mapsto v_{1,1}, \ldots, {q_n}^{(1)} \mapsto v_{1,n}, \ldots, \\
&\qquad {q_1}^{(i-1)} \mapsto v_{i-1,1}, \ldots, {q_n}^{(i-1)} \mapsto v_{i-1,n}] \\
& && \text{(Prop. A.1.2)} \\
\Rightarrow&\sigma_{(i,f)} \in_i \underline{\sigma}_{p'}[{q_1}^{(1)} \mapsto v_{1,1}, \ldots, {q_n}^{(1)} \mapsto v_{1,n}, \ldots, \\
&\qquad {q_1}^{(i)} \mapsto v_{i,1}, \ldots, {q_n}^{(i)} \mapsto v_{i,n}] && \text{(Prop. A.1.1)} \\
\Rightarrow&\sigma_{(i,f)} \in_i \underline{\sigma}_{p'}[{q_1}^{(1)} \mapsto v_{1,1}, \ldots, {q_n}^{(1)} \mapsto v_{1,n}, \ldots, \\
&\qquad {q_1}^{(k)} \mapsto v_{k,1}, \ldots, {q_n}^{(k)} \mapsto v_{k,n}] && \text{(Prop. A.1.2)} \\
\Rightarrow&\sigma_{(i,f)} \in_i \underline{\sigma}_f
\end{aligned}$$

By Lemma A.1.8.1 we know that $\langle \underline{s_m}, \underline{\sigma}_f\rangle \rightarrow^{l_2-2} \langle \texttt{skip}, \underline{\sigma}'_f\rangle$ for some $\underline{\sigma}'_f$ s.t. $\underline{\sigma}'''({t_j}^{(i)}) = \underline{\sigma}'_f({r_j}^{(i)})$ for all $1 \leq j \leq m$ and $i \in A \cup I$. We may apply the induction hypothesis to this trace and A (IH). By Lemma A.1.13 we get that for all $i \in A$ and $1 \leq j \leq m$, $\underline{\sigma}'({x_j}^{(i)}) = \underline{\sigma}'''({t_j}^{(i)})$.
We now prove (a). For all $i \in A$, since $\underline{\sigma}_f(p'^{(i)}) = \top$, we know by (IH) that $\langle s_m, \sigma_{(i,f)}\rangle \rightarrow^* \langle \texttt{skip}, \sigma'_{(i,f)}\rangle$ for some $\sigma'_{(i,f)}$ s.t.

$\sigma'_{(i,f)} \in_i \underline{\sigma}'_f$. We can therefore construct

$$\begin{aligned}&\langle x_1, \dots, x_m{:=}\mathtt{frame}_{r_1,\dots,r_m}(s_m, \sigma_{(i,f)}), \sigma_i\rangle\\ \rightarrow^* &\langle x_1, \dots, x_m{:=}\mathtt{frame}_{r_1,\dots,r_m}(\mathtt{skip}, \sigma'_{(i,f)}), \sigma_i\rangle\end{aligned}$$

using Frame1 and $\langle x_1, \dots, x_m{:=}\mathtt{frame}_{r_1,\dots,r_m}(\mathtt{skip}, \sigma'_{(i,f)}), \sigma_i\rangle \rightarrow \langle \mathtt{skip}, \sigma'_i\rangle$ by Frame2, where $\sigma'_i = \sigma_i[x_1 \mapsto \sigma'_{(i,f)}(r_1)] \dots [x_m \mapsto \sigma'_{(i,f)}(r_m)]$ and therefore $\sigma'_i \in_i \underline{\sigma}'$ by Prop. A.1.2 and Prop. A.1.1. By Prop. A.1.5, we also get $\underline{\sigma}'' \precsim_i^{\underline{s}} \underline{\sigma}'''$ since the assigned variables ts are in *freshvars*($\underline{s}$), and $\underline{\sigma}''' \precsim_i^{\underline{s}} \underline{\sigma}'$ since the assigned variables $\mathring{p}_j$ are in RPVar. Because of this and $\underline{\sigma} \preccurlyeq^{\underline{s}} \underline{\sigma}''$ we get by Prop. A.1.3 that $\underline{\sigma} \precsim_i^{\underline{s}} \underline{\sigma}'$.

We now prove (b). For all $i \in I$, we also have $\underline{\sigma} \preccurlyeq_i^{\underline{s}} \underline{\sigma}'$ by Prop. A.1.3 since $\underline{\sigma} \preccurlyeq^{\underline{s}} \underline{\sigma}''$, $\underline{\sigma}'' \preccurlyeq_i^{\underline{s}} \underline{\sigma}'''$ by Prop. A.1.5 and $\underline{\sigma}''' \preccurlyeq_i^{\underline{s}} \underline{\sigma}'$ by Prop. A.1.4.

- $s = s_1; s_2$: Then $\underline{s}$ must have the form $\underline{s_1}; \underline{s_2}$, where $\underline{s_1} = [\![s_1]\!]_k^{\mathring{p}}$ and $\underline{s_2} = [\![s_2]\!]_k^{\mathring{p}}$. By Lemma A.1.7.1, we get that

$$\begin{aligned}&\langle \underline{s}, \underline{\sigma}\rangle\\ \rightarrow^{l_1} &\langle \mathtt{skip}; \underline{s_2}, \underline{\sigma}'' && \text{(for some } l_1, \underline{\sigma}''\text{)}\\ \rightarrow^{1} &\langle \underline{s_2}, \underline{\sigma}''\rangle\\ \rightarrow^{l_2} &\langle \mathtt{skip}, \underline{\sigma}'\rangle && \text{(for some } l_2\text{)}\end{aligned}$$

where $l = 1 + l_1 + l_2$ and all $l_i \geq 0$.
Then by Lemma A.1.7.2, we have $\langle \underline{s_1}, \underline{\sigma}\rangle \rightarrow^{l_1} \langle \mathtt{skip}, \underline{\sigma}''\rangle$, and we can apply the induction hypothesis to this and A (IH1). We get that $\underline{\sigma} \precsim_i^{\underline{s_1}} \underline{\sigma}''$ for all $i \in A$ and $\underline{\sigma} \preccurlyeq_i^{\underline{s_1}} \underline{\sigma}''$ for all $i \in I$. Therefore $\underline{\sigma}''(p^{(i)}) = \underline{\sigma}(p^{(i)})$ for all $i \in \{1, \dots, k\}$. Subsequently we can apply the induction hypothesis to $\langle \underline{s_2}, \underline{\sigma}''\rangle \rightarrow^{l_2} \langle \mathtt{skip}, \underline{\sigma}'\rangle$ and A (IH2).
We first prove (a): For all $i \in A$, we have $\underline{\sigma}(p^{(i)}) = \top$ and $\sigma_i \in_i \underline{\sigma}$. By (IH1) we get $\langle s_1, \sigma_i\rangle \rightarrow^* \langle \mathtt{skip}, \sigma''_i\rangle$ for some σ''_i s.t. $\sigma''_i \in_i \underline{\sigma}''$, and $\underline{\sigma} \precsim_i^{\underline{s_1}} \underline{\sigma}''$. Then since $\underline{\sigma}''(p^{(i)}) = \top$, by (IH2) we get $\langle s_2, \sigma''_i\rangle \rightarrow^* \langle \mathtt{skip}, \sigma'_i\rangle$ s.t. $\sigma'_i \in_i \underline{\sigma}'$ and we can construct $\langle s, \sigma_i\rangle \rightarrow^* \langle \mathtt{skip}; s_2, \sigma''_i\rangle \rightarrow \langle s_2, \sigma''_i\rangle \rightarrow^* \langle \mathtt{skip}, \sigma'_i\rangle$. Additionally, we get $\underline{\sigma}'' \precsim_i^{\underline{s_2}} \underline{\sigma}'$ by (IH2), and by Prop. A.1.3, we get $\underline{\sigma} \precsim_i^{\underline{s}} \underline{\sigma}'$, and we are done.
We now prove (b): For all $i \in I$ we get that $\underline{\sigma} \preccurlyeq_i^{\underline{s_1}} \underline{\sigma}''$ from (IH1) and $\underline{\sigma}'' \preccurlyeq_i^{\underline{s_2}} \underline{\sigma}'$ from (IH2), and by Prop. A.1.3, we get $\underline{\sigma} \preccurlyeq_i^{\underline{s}} \underline{\sigma}'$ and are done.

□

A.1.3. Proof of Lemma 3.4.4

Lemma A.1.20 *Assume that match*$(\Phi, \underline{\Phi})$ *and that for some sets* $A \subseteq \{1, \dots, k\}$ *and* $I = \{1, \dots, k\} \setminus A$ *we have that* $\forall i \in A.\, p^{(i)} \Downarrow_{\underline{\sigma}} \top \wedge \sigma_i \in_i \underline{\sigma}$ *and* $\forall i \in I.\, p^{(i)} \Downarrow_{\underline{\sigma}} \bot$. *If* $\langle \underline{s}, \underline{\sigma}\rangle \rightarrow^l \langle \dot{s}, \underline{\sigma}'\rangle$, *where* $\underline{s} = [\![s]\!]_k^{\mathring{p}}$ *and* $\dot{s} \neq \mathtt{skip}$, *under $\underline{\Phi}$, then for at least one $i_e \in A$, $\langle s, \sigma_{i_e} \rangle \longrightarrow^* \langle \dot{s}, \sigma'_{i_e} \rangle$ under Φ for some σ'_{i_e}.*

Proof. We prove the lemma separately for the two possible cases $\dot{s} = \texttt{error}$ and $\dot{s} = \texttt{magic}$. Here we show only the case for $\dot{s} = \texttt{error}$, since the other case is completely analogous, with the cases for asserts and assumes being swapped.

The proof goes by strong induction on l. We perform a case split on the structure of s.

- $s = x{:=}e$: This case is impossible by Lemma A.1.13.
- $s = \texttt{havoc}\ x$: This case is impossible by Lemma A.1.15.
- $s = \texttt{assert}\ e$: Then by Lemma A.1.17 we have some non-empty set $A_e \subseteq A$ s.t. for all $i \in A_e$, $e^{(i)} \Downarrow_{\underline{\sigma}} \bot$. By Lemma 3.4.1, this implies that $e \Downarrow_{\sigma_i} \bot$, and by Assert2 we therefore have $\langle s, \sigma_i \rangle \longrightarrow \langle \texttt{error}, \sigma_i \rangle$ for all $i \in A_e$.
- $s = \texttt{assume}\ e$: This case is impossible by Lemma A.1.18 and Lemma A.1.19.
- $s = \texttt{if}\ (e)\ \{s_1\}\ \texttt{else}\ \{s_2\}$:
  Then $\underline{s}$ has the form $\bigodot_{i=1}^{k}(t^{(i)}{:=}p^{(i)} \wedge e^{(i)}); \bigodot_{i=1}^{k}(f^{(i)}{:=}p^{(i)} \wedge \neg e^{(i)}); \underline{s_1}; \underline{s_2}$, where $\underline{s_1} = \llbracket s_1 \rrbracket_k^{\mathring{t}}$ and $\underline{s_2} = \llbracket s_2 \rrbracket_k^{\mathring{f}}$ and $\mathring{t}$ and $\mathring{f}$ are fresh variable names.
  By Lemma A.1.7.3 and Lemma A.1.11 we must have that $\langle \underline{s}, \underline{\sigma} \rangle \longrightarrow^* \langle \underline{s_1}; \underline{s_2}, \underline{\sigma}'' \rangle$, where $\underline{\sigma}'' = \underline{\sigma}[t^{(1)} \mapsto v_1] \ldots [t^{(k)} \mapsto v_k][f^{(1)} \mapsto v'_1] \ldots [f^{(k)} \mapsto v'_l]$, s.t. $p^{(i)} \wedge e^{(i)} \Downarrow_{\underline{\sigma}} v_i$ and $p^{(i)} \wedge \neg e^{(i)} \Downarrow_{\underline{\sigma}} v'_i$. Then by repeated use of Seq1 and Seq2 we have $\langle \underline{s}, \underline{\sigma} \rangle \longrightarrow^* \langle \underline{s_1}; \underline{s_2}, \underline{\sigma}'' \rangle$. For some $A_1 \subseteq A$ and $A_2 = A \setminus A_1$ we have that for all $i \in A_1$, $e^{(i)} \Downarrow_{\underline{\sigma}} \top$ and therefore $\underline{\sigma}''(t^{(i)}) = \top$, and $\underline{\sigma}''(t^{(i)}) = \bot$ for all $i \in A_2$. We must then have that $\langle \underline{s_1}; \underline{s_2}, \underline{\sigma}'' \rangle \longrightarrow^{l'} \langle \texttt{error}, \underline{\sigma}' \rangle$ for some $l' < l$. By Lemma A.1.7.3 this means that either $\langle \underline{s_1}, \underline{\sigma}'' \rangle \longrightarrow^{l'-1} \langle \texttt{error}, \underline{\sigma}' \rangle$ or $\langle \underline{s_1}, \underline{\sigma}'' \rangle \longrightarrow^{l_1} \langle \texttt{skip}, \underline{\sigma}''' \rangle \wedge \langle \underline{s_2}, \underline{\sigma}''' \rangle \longrightarrow^{l_2} \langle \texttt{error}, \underline{\sigma}' \rangle$ for some $\underline{\sigma}''', l_1, l_2$ s.t. $l' = 1 + l_1 + l_2$.

  - In the former case, we get by the induction hypothesis that for some $i \in A_1$, $\langle s_1, \sigma_i \rangle \longrightarrow^* \langle \texttt{error}, \sigma'_i \rangle$ for some σ'_i. Since we also have that $e^{(i)} \Downarrow_{\underline{\sigma}} \top$ and therefore by Lemma 3.4.1 $e \Downarrow_{\sigma_i} \top$, we get by Cond1 and Seq1 and Seq3 that $\langle s, \sigma_i \rangle \longrightarrow^* \langle \texttt{error}; s_2, \sigma'_i \rangle \longrightarrow \langle \texttt{error}, \sigma'_i \rangle$, and we are done.
  - In the latter case, we get by Thm. 3.4.2 that for all $i \in A_1$, $\underline{\sigma}'''(f^{(i)}) = \bot$ and $\langle s_1, \sigma_i \rangle \longrightarrow^* \langle \texttt{skip}, \sigma'''_i \rangle$ for some σ'''_i s.t. $\sigma'''_i \in_i \underline{\sigma}'''$, and for all $i \in A_2$, $\underline{\sigma}'''(f^{(i)}) = \top$ and $\underline{\sigma}'' \preccurlyeq_i^{s_1} \underline{\sigma}'''$, which implies $\sigma_i \in_i \underline{\sigma}''$ by Lemma A.1.6, and for all $i \in I$, $\underline{\sigma}'''(f^{(i)}) = \bot$. By the induction hypothesis, we get that for some $i \in A_2$, $\langle s_2, \sigma_i \rangle \longrightarrow^* \langle \texttt{error}, \sigma'_i \rangle$ for some σ'_i. Since $e^{(i)} \Downarrow_{\underline{\sigma}} \bot$, we know that $e \Downarrow_{\sigma_i} \bot$ by Lemma 3.4.1, and we can use Cond2 to construct $\langle s, \sigma_i \rangle \longrightarrow \langle s_2, \sigma_i \rangle \longrightarrow^* \langle \texttt{error}, \sigma'_i \rangle$.

- $s \equiv \texttt{while}\ (e)\ \{s_l\}$:
  Then $\underline{s}$ must have the form $\texttt{while}\ (\bigvee_{i=1}^{k}(p^{(i)} \wedge e^{(i)}))\ \{\bigodot_{i=1}^{k}(t^{(i)}{:=}p^{(i)} \wedge e^{(i)}); \underline{s_l}\}$ where $\underline{s_l} = \llbracket s_l \rrbracket_k^{\mathring{t}}$ and *freshvars*$(\underline{s}) = \{\mathring{t}\} \cup$ *freshvars*$(\underline{s_l})$.
  $\underline{s}$ must progress via Whl1, since if it progressed by Whl2, we would have $\langle \underline{s}, \underline{\sigma} \rangle \longrightarrow \langle \texttt{skip}, \underline{\sigma} \rangle$. Therefore we have $\langle \underline{s}, \underline{\sigma} \rangle \longrightarrow \langle \bigodot_{i=1}^{k}(t^{(i)}{:=}p^{(i)} \wedge e^{(i)}); \underline{s_l}; \underline{s}, \underline{\sigma} \rangle \longrightarrow^{l-1} \langle \texttt{error}, \underline{\sigma}' \rangle$.

Then by Lemma A.1.7.3, Lemma A.1.7.2 and Lemma A.1.11 we have $\langle \bigodot_{i=1}^{k}(t^{(i)}{:=}p^{(i)} \wedge e^{(i)}), \underline{\sigma}\rangle \rightarrow^{*} \langle \mathtt{skip}; \underline{s_l}; \underline{s}, \underline{\sigma}''\rangle$ for $\underline{\sigma}'' = \underline{\sigma}[t^{(1)} \mapsto v_1]\ldots[t^{(k)} \mapsto v_k]$ s.t. $p^{(i)} \wedge e^{(i)} \Downarrow_{\underline{\sigma}} v_i$, and $\langle \underline{s_l}; \underline{s}, \underline{\sigma}''\rangle \rightarrow^{l'} \langle \mathtt{error}, \underline{\sigma}'\rangle$ for some $l' < l$. We define $A_t = \{i | i \in A \wedge e^{(i)} \Downarrow_{\underline{\sigma}} \top\}$ and $I_t = \{1, \ldots, k\} \setminus A_t$. We have that for all $i \in A_t$, $\underline{\sigma}''(t^{(i)}) = \top$, and for all $i \in I_t$, $\underline{\sigma}''(t^{(i)}) = \bot$. Since $\mathring{t}$ are in *freshvars*($\underline{s}$), we have that $\underline{\sigma} \preccurlyeq^{\underline{s}} \underline{\sigma}''$ by Prop. A.1.5. Therefore by Lemma A.1.6 we have that $\sigma_i \in_i \underline{\sigma}''$ for all $i \in A$.
Then by Lemma A.1.7.3 we have that either $\langle \underline{s_l}, \underline{\sigma}''\rangle \rightarrow^{l'-1} \langle \mathtt{error}, \underline{\sigma}'\rangle$ or $\langle \underline{s_l}, \underline{\sigma}''\rangle \rightarrow^{l_1} \langle \mathtt{skip}, \underline{\sigma}'''\rangle \wedge \langle \underline{s}, \underline{\sigma}'''\rangle \rightarrow^{l_2} \langle \mathtt{error}, \underline{\sigma}'\rangle$ for some $\underline{\sigma}''', l_1, l_2$ s.t. $l' = 1 + l_1 + l_2$.

- In the first case, we can apply the induction hypothesis to $\langle \underline{s_l}, \underline{\sigma}''\rangle \rightarrow^{l'-1} \langle \mathtt{error}, \underline{\sigma}'\rangle$ and A and get that for some $i \in A_t$, $\langle s_l, \sigma_i\rangle \rightarrow^{*} \langle \mathtt{error}, \sigma_i'\rangle$ for some σ_i'. Since we also have $e \Downarrow_{\sigma_i} \top$ by Lemma 3.4.1, we can use Whl1 and Seq1 and Seq3 to construct $\langle s, \sigma_i\rangle \rightarrow \langle s_l; s, \sigma_i\rangle \rightarrow^{*} \langle \mathtt{error}; s, \sigma_i'\rangle \rightarrow \langle \mathtt{error}, \sigma_i'\rangle$, and we are done.
- In the second case, by Thm. 3.4.2, we have that for all $i \in A_t$, $\underline{\sigma}'' \precsim_i^{s_l} \underline{\sigma}'''$ and $\langle s_l, \sigma_i\rangle \rightarrow^{*} \langle \mathtt{skip}, \sigma_i''\rangle$ for some σ_i'' s.t. $\sigma_i'' \in_i \underline{\sigma}'''$, and for all $i \in I_t$, $\underline{\sigma}'' \preccurlyeq_i^{s_l} \underline{\sigma}'''$. Therefore for all $i \in A$ we have $\underline{\sigma}'''(p^{(i)}) = \top$ and for all $i \in I$ $\underline{\sigma}'''(p^{(i)}) = \bot$. Additionally, by Lemma A.1.6 we have for all $i \in A \setminus A_t$ that $\sigma_i \in_i \underline{\sigma}'''$. Therefore by applying the induction hypothesis to $\langle \underline{s}, \underline{\sigma}'''\rangle \rightarrow^{l_2} \langle \mathtt{error}, \underline{\sigma}'\rangle$ and A, we get that for some $i \in A$, $\langle s, \sigma_i'''\rangle \rightarrow^{*} \langle \mathtt{error}, \sigma_i'\rangle$ for some σ_i' and $\sigma_i''' = \sigma_i''$ if $i \in A_t$ and $\sigma_i''' = \sigma_i$ otherwise.
  - ∗ If $i \in A_t$ then we can construct $\langle s, \sigma_i\rangle \rightarrow \langle s_l; s, \sigma_i\rangle \rightarrow^{*} \langle \mathtt{skip}; s, \sigma_i''\rangle \rightarrow \langle s, \sigma_i''\rangle \rightarrow^{*} \langle \mathtt{error}, \sigma_i'\rangle$ by Seq1 and Seq2, and we are done.
  - ∗ Otherwise we already have $\langle s, \sigma_i\rangle \rightarrow^{*} \langle \mathtt{error}, \sigma_i'\rangle$ and we are also done.

► $s \equiv x_1, \ldots, x_m{:=}\ m(e_1, \ldots, e_n)$: Then $\underline{s}$ has the form

```
if   $(\bigvee_{i=1}^{k} p^{(i)})$ {
        $\bigodot_{i=1}^{k}$ if $(p^{(i)})$ {$\bigodot_{j=1}^{n}(a_j{}^{(i)}{:=}e_j{}^{(i)})$};
        $ts{:=}\ m(p^{(1)}, \ldots, p^{(k)}, as)$;
        $\bigodot_{i=1}^{k}$ if $(p^{(i)})$ {$\bigodot_{j=1}^{m}(x_j{}^{(i)}{:=}t_j{}^{(i)})$}
}
```

where $\mathring{a}_j$ and $\mathring{t}_j$ are in *freshvars*($\underline{s}$) and $ts = [t_1{}^{(1)}, \ldots, t_1{}^{(k)}, \ldots, t_m{}^{(1)}, \ldots, t_m{}^{(k)}]$ and $as = [a_1{}^{(1)}, \ldots, a_1{}^{(k)}, \ldots, a_n{}^{(1)}, \ldots, a_n{}^{(k)}]$.
Then by Lemma A.1.7.3, Lemma A.1.7.2 and Lemma A.1.13, we must have that

$$\langle \underline{s}, \underline{\sigma}\rangle \rightarrow^{*} \langle ts{:=}\ m(p^{(1)}, \ldots, p^{(k)}, as); \bigodot_{i=1}^{k} \mathtt{if}\ (p^{(i)})\ \{\bigodot_{j=1}^{m}(x_j{}^{(i)}{:=}t_j{}^{(i)})\}, \underline{\sigma}''\rangle$$

for some $\underline{\sigma}''$, and $\langle ts{:=}\ m(p^{(1)}, \ldots, p^{(k)}, as), \underline{\sigma}''\rangle \rightarrow^{l'} \langle \mathtt{error}, \underline{\sigma}'\rangle$ for some $l' \leq l$. Assuming the same form of Φ and $\underline{\Phi}$ as in the previous proof, we must then have that by Call, $\langle ts{:=}\ m(p^{(1)}, \ldots, p^{(k)}, as), \underline{\sigma}''\rangle \rightarrow \langle ts{:=}\mathtt{frame}_{rs}(\underline{s_m}, \underline{\sigma}_f), \underline{\sigma}''\rangle \rightarrow^{l'-1}$

$\langle \texttt{error}, \underline{\sigma}' \rangle$ for some $\underline{\sigma}_f$. Then by Lemma A.1.8.2 we get that $\langle \underline{s_m}, \underline{\sigma}_f \rangle \rightarrow^{l'-2} \langle \texttt{error}, \underline{\sigma}'_f \rangle$ for some $\underline{\sigma}'_f$.
For all $i \in A$, we get by Call that $\langle s, \sigma_i \rangle \rightarrow \langle x_1, \dots, x_m{:=}\texttt{frame}_{r_1,\dots,r_m}(s_m, \sigma_{(i,f)}), \sigma_i \rangle$ for some $\sigma_{(i,f)}$, and by the same argument as in the proof for Thm. 3.4.2, we have that $\sigma_{(i,f)} \in_i \underline{\sigma}_f$ and $\underline{\sigma}_f(p'^{(i)}) = \top$ for all $i \in A$, and $\underline{\sigma}_f(p'^{(i)}) = \bot$ for all $i \in I$. Therefore by applying the induction hypothesis to $\langle \underline{s_m}, \underline{\sigma}_f \rangle \rightarrow^{l'-2} \langle \texttt{error}, \underline{\sigma}'_f \rangle$ and A we get for some $i \in A$ that $\langle s_m, \sigma_{(i,f)} \rangle \rightarrow^* \langle \texttt{error}, \sigma'_{(i,f)} \rangle$ for some $\sigma'_{(i,f)}$.
Then we can construct $\langle s, \sigma_i \rangle \rightarrow \langle x_1, \dots, x_m{:=}\texttt{frame}_{r_1,\dots,r_m}(s_m, \sigma_{(i,f)}), \sigma_i \rangle$ $\rightarrow^* \langle x_1, \dots, x_m{:=}\texttt{frame}_{r_1,\dots,r_m}(\texttt{error}, \sigma'_{(i,f)}), \sigma_i \rangle \rightarrow \langle \texttt{error}, \sigma_i \rangle$ using Frame1 and Frame3 and we are done.

- ► $s \equiv s_1; s_2$: Then $\underline{s} = \underline{s_1}; \underline{s_2}$, where $\underline{s_1} = [\![s_1]\!]^{\mathring{p}}_k$ and $\underline{s_2} = [\![s_2]\!]^{\mathring{p}}_k$. By Lemma A.1.7.3 we must have that $\langle \underline{s_1}, \underline{\sigma} \rangle \rightarrow^{l-1} \langle \texttt{error}, \underline{\sigma}' \rangle$ or that $\langle \underline{s_1}, \underline{\sigma} \rangle \rightarrow^{l_1} \langle \texttt{skip}, \underline{\sigma}'' \rangle \wedge \langle \underline{s_2}, \underline{\sigma}'' \rangle \rightarrow^* \langle \texttt{error}, \underline{\sigma}' \rangle l_2$ for some $\underline{\sigma}'', l_1, l_2$ s.t. $l = 1 + l_1 + l_2$.

  - In the former case, we get by the induction hypothesis that $\langle s_1, \sigma_i \rangle \rightarrow^* \langle \texttt{error}, \sigma'_i \rangle$ for some $i \in A$ and some σ'_i. We can construct $\langle s_1; s_2, \sigma_i \rangle \rightarrow^* \langle \texttt{error}; s_2, \sigma'_i \rangle \rightarrow \langle \texttt{error}, \sigma'_i \rangle$ using Seq1 and Seq3 and we are done.
  - In the latter case, we get by Thm. 3.4.2 that for all $i \in A$, $\underline{\sigma}''(p^{(i)}) = \top$ and $\langle s_1, \sigma_i \rangle \rightarrow^* \langle \texttt{skip}, \sigma''_i \rangle$ for some σ''_i s.t. $\sigma''_i \in_i \underline{\sigma}''$, and that for all $i \in I$, $\underline{\sigma}''(p^{(i)}) = \bot$. We then get by the induction hypothesis that $\langle s_2, \sigma''_i \rangle \rightarrow^* \langle \texttt{error}, \sigma'_i \rangle$ for some $i \in A$ and some σ'_i. We can construct $\langle s, \sigma_i \rangle \rightarrow^* \langle \texttt{skip}; s_2, \sigma''_i \rangle \rightarrow \langle s_2, \sigma''_i \rangle \rightarrow^* \langle \texttt{error}, \sigma'_i \rangle$ by Seq1, Seq2, and Seq3, and we are done.

□

A.1.4. Proof of Thm. 3.4.5

Theorem 3.4.5 *Assume that match*$(\Phi, \underline{\Phi})$ *and that for a set of indices* $A \subseteq \{1, \dots, k\}$ *there is a derivation* $d_i = \langle s, \sigma_i \rangle \rightarrow^{l_i} \langle \texttt{skip}, \sigma'_i \rangle$ *under* Φ *for each* $i \in A$. *Assume also that* $\sigma_i \in_i \underline{\sigma}$ *and* $\underline{\sigma}(p^{(i)}) = \top$ *for all* $i \in A$, *and* $\underline{\sigma}(p^{(i)}) = \bot$ *for any* $i \in I$, *where* $I = \{1, \dots, k\} \setminus A$.

Then $\langle \underline{s}, \underline{\sigma} \rangle \rightarrow^* \langle \texttt{skip}, \underline{\sigma}' \rangle$, *where* $\underline{s} = [\![s]\!]^{\mathring{p}}_k$, *under* $\underline{\Phi}$ *for some* $\underline{\sigma}'$ *s.t. for all* $i \in A$, $\sigma'_i \in_i \underline{\sigma}'$ *and* $\underline{\sigma} \precsim^s_i \underline{\sigma}'$, *and for all* $i \in I$, $\underline{\sigma} \preccurlyeq^s_i \underline{\sigma}'$.

Proof. The proof goes by strong induction on the sum of the lengths of traces $l = \sum_{i \in A} l_i$.

- ► Case $l = 0$: Then either $|A| = 0$ or $|A| > 0$.
  If $|A| = 0$ then we need to show that for all $i \in \{1, \dots, k\}$ we have $\langle \underline{s}, \underline{\sigma} \rangle \rightarrow^* \langle \texttt{skip}, \underline{\sigma}' \rangle$ for some $\underline{\sigma}'$ s.t. $\underline{\sigma} \preccurlyeq^s_i \underline{\sigma}'$, which we get by Lemma 3.4.3.
  If $|A| > 0$ we must have that $s = \texttt{skip}$, since all other statements require a trace of length at least 1 to step to $\texttt{skip}$ and the sum of j such lengths would therefore be greater than zero. Therefore $\sigma_i = \sigma'_i$ for all $i \in A$. Since $\underline{s} = \texttt{skip}$, we know that $\underline{\sigma}' = \underline{\sigma}$ and

therefore for all $i \in A$ $\sigma'_i \in_i \underline{\sigma}'$ and $\underline{\sigma} \precsim_i^{s} \underline{\sigma}'$ by reflexivity of $\preccurlyeq_i^{V}$. For all $i \in I$, we get that $\underline{\sigma} \preccurlyeq_i^{s} \underline{\sigma}'$ by reflexivity of $\preccurlyeq_i^{V}$.

- Case $l > 0$: We make a case distinction based on the structure of s.
  - $s \equiv x{:=}e$: Then $\underline{s}$ has the form $\bigodot_{i=1}^{k} \texttt{if}\,(p^{(i)})\ \{x^{(i)}{:=}e^{(i)}\}$. By Assign, for each $i \in A$ we have $d_i = \langle s, \sigma_i \rangle \rightarrow \langle \texttt{skip}, \sigma_i[x \mapsto v_i] \rangle$ s.t. $e \Downarrow_{\sigma_i} v_i$, and therefore $\sigma'_i = \sigma_i[x \mapsto v_i]$. By Lemma 3.4.1, since $\mathit{fv}(e) \in \mathrm{PVar}$, we get that $e^{(i)} \Downarrow_{\underline{\sigma}} v_i$ for each $i \in A$. By Lemma A.1.12 and Lemma A.1.13 we have that $\langle \underline{s}, \underline{\sigma} \rangle \rightarrow^{*} \langle \texttt{skip}, \underline{\sigma}' \rangle$ where $\underline{\sigma}' = \underline{\sigma}[x^{(j_1)} \mapsto v_{j_1}] \ldots [x^{(j_n)} \mapsto v_{j_n}]$ for $\{j_1, \ldots, j_n\} = A$ s.t. $e^{(j_n)} \Downarrow_{\underline{\sigma}} v_{j_n}$. Like in the corresponding case in the proof for Thm. 3.4.2, we get that $\sigma'_i \in_i \underline{\sigma}'$ for all $i \in A$ by repeated use of Prop. A.1.2 and Prop. A.1.1, and since all $x^{(i)} \in \mathrm{RPVar}$, we also have that $\underline{\sigma} \precsim_i^{s} \underline{\sigma}'$ by Prop. A.1.5. Similarly, we have $\underline{\sigma} \preccurlyeq_i^{s} \underline{\sigma}'$ for all $i \in I$ by Prop. A.1.4.
  - $s \equiv \texttt{havoc}\ x$: Then $\underline{s}$ has the form $\bigodot_{i=1}^{k} \texttt{if}\,(p^{(i)})\ \{\texttt{havoc}\ x^{(i)}\}$. By Havoc, for each $i \in A$ we have $d_i = \langle s, \sigma_i \rangle \rightarrow \langle \texttt{skip}, \sigma_i[x \mapsto v_i] \rangle$ for some v_i, and therefore $\sigma'_i = \sigma_i[x \mapsto v_i]$. By Lemma A.1.14 and Lemma A.1.15 we get that $\langle \underline{s}, \underline{\sigma} \rangle \rightarrow^{*} \langle \texttt{skip}, \underline{\sigma}' \rangle$ where $\underline{\sigma}' = \underline{\sigma}[x^{(i_1)} \mapsto v_{i_1}] \ldots [x^{(i_n)} \mapsto v_{i_n}]$ for $\{i_1, \ldots, i_n\} = A$. Like in the corresponding case in the proof for Thm. 3.4.2, we get that $\sigma'_i \in_i \underline{\sigma}'$ for all $i \in A$ by repeated use of Prop. A.1.2 and Prop. A.1.1, and since all $x^{(i)} \in \mathrm{RPVar}$, we also have that $\underline{\sigma} \precsim_i^{s} \underline{\sigma}'$ by Prop. A.1.5. Similarly, we have $\underline{\sigma} \preccurlyeq_i^{s} \underline{\sigma}'$ for all $i \in I$ by Prop. A.1.4.
  - $s \equiv \texttt{skip}$: This is impossible, since any trace for $s = \texttt{skip}$ has length zero, and therefore the sum of the lengths of such traces is zero as well.
  - $s \equiv \texttt{assert}\ e$: Then $\underline{s}$ must have the form $\bigodot_{i=1}^{k} \texttt{if}\,(p^{(i)})\ \{\texttt{assert}\ e^{(i)}\}$. We must have that for all $i \in A$, d_i proceeds with Assert1 and therefore $e \Downarrow_{\sigma_i} \top$, and by Lemma 3.4.1, $e^{(i)} \Downarrow_{\underline{\sigma}} \top$. Since for all $i \in I$, $\underline{\sigma}(p^{(i)}) = \bot$, by Lemma A.1.16, we have that $\langle \underline{s}, \underline{\sigma} \rangle \rightarrow^{*} \langle \texttt{skip}, \underline{\sigma} \rangle$ and therefore $\underline{\sigma}' = \underline{\sigma}$, which trivially implies $\sigma'_i \in_i \underline{\sigma}'$ for all $i \in A$, as well as $\underline{\sigma} \precsim_i^{s} \underline{\sigma}'$ because of reflexivity of $\preccurlyeq_i^{V}$. Similarly, for all $i \in I$, we have $\underline{\sigma} \preccurlyeq_i^{s} \underline{\sigma}'$ because of reflexivity.
  - $s \equiv \texttt{assume}\ e$: This case is analogous to the previous one.
  - $s \equiv \texttt{if}\,(e)\ \{s_1\}\ \texttt{else}\ \{s_2\}$:

    Then $\underline{s}$ has the form $\bigodot_{i=1}^{k}(t^{(i)}{:=}p^{(i)} \wedge e^{(i)}); \bigodot_{i=1}^{k}(f^{(i)}{:=}p^{(i)} \wedge \neg e^{(i)}); \underline{s_1}; \underline{s_2}$, where $\underline{s_1} = [\![s_1]\!]_k^{\mathring{t}}$ and $\underline{s_2} = [\![s_2]\!]_k^{\mathring{f}}$ and $\mathring{t}$ and $\mathring{f}$ are fresh variable names.

    We first identify the sets of subtraces to which we can apply the induction hypothesis. We define $A_1 = \{i | i \in A \wedge e \Downarrow_{\sigma_i} \top\}$ and $A_2 = \{i | i \in A \wedge e \Downarrow_{\sigma_i} \bot\}$. Note that $A_1 \cup A_2 = A$.

    We must then have that for all $i \in A_1$, d_i progresses by Cond1 and therefore has the form $\langle s, \sigma_i \rangle \rightarrow \langle s_1, \sigma_i \rangle$ followed by a subtrace $d'_i = \langle s_1, \sigma_i \rangle \rightarrow^{l_i - 1} \langle \texttt{skip}, \sigma'_i \rangle$. We apply the induction hypothesis to A_1 and the subtraces d'_i (IH1). Conversely, for all $i \in A_2$, d_i progresses by Cond2 and therefore has the form $\langle s, \sigma_i \rangle \rightarrow \langle s_2, \sigma_i \rangle$ followed by $d''_i = \langle s_2, \sigma_i \rangle \rightarrow^{l_i - 1} \langle \texttt{skip}, \sigma'_i \rangle$. We again apply the induction hypothesis to A_2 and the traces d''_i (IH2).

    We now construct a corresponding product execution.

By Lemma A.1.10 and Lemma A.1.11 we know that $\langle \bigodot_{i=1}^{k}(t^{(i)}{:=}p^{(i)} \land e^{(i)}), \underline{\sigma}\rangle \rightarrow^{*} \langle \texttt{skip}, \underline{\sigma}[t^{(1)} \mapsto v_1]\ldots[t^{(i)} \mapsto v_k]\rangle$ s.t. $p^{(i)} \land e^{(i)} \Downarrow_{\underline{\sigma}} v_i$. By repeated use of Seq1 and Seq2, we get that $\langle \underline{s}, \underline{\sigma}\rangle \rightarrow^{*} \langle \bigodot_{i=1}^{k}(f^{(i)}{:=}p^{(i)} \land \neg e^{(i)}); \underline{s_1}; \underline{s_2}, \underline{\sigma}[t^{(1)} \mapsto v_1]\ldots[t^{(i)} \mapsto v_k]\rangle$. By the same argument, we get $\langle \bigodot_{i=1}^{k}(f^{(i)}{:=}p^{(i)} \land \neg e^{(i)}); \underline{s_1}; \underline{s_2}, \underline{\sigma}[t^{(1)} \mapsto v_1]\ldots[t^{(i)} \mapsto v_k]\rangle \rightarrow^{*} \langle \underline{s_1}; \underline{s_2}, \underline{\sigma}''\rangle$ where $\underline{\sigma}'' = \underline{\sigma}[t^{(1)} \mapsto v_1]\ldots[t^{(i)} \mapsto v_k][f^{(1)} \mapsto v'_1]\ldots[f^{(k)} \mapsto v'_k]$ s.t. $p^{(i)} \land \neg e^{(i)} \Downarrow_{\underline{\sigma}} v'_i$.

Now we have for all $i \in A_1$ that $e \Downarrow_{\sigma_i} \top$ and by Lemma 3.4.1 that $e^{(i)} \Downarrow_{\underline{\sigma}} \top$. Since $\underline{\sigma}(p^{(i)}) = \top$, we also have that $\underline{\sigma}''(t^{(i)}) = \top$ and $\underline{\sigma}''(f^{(i)}) = \bot$ by the rules of expression evaluation. On the other hand, we have for all $i \in A_2$ that $\underline{\sigma}''(t^{(i)}) = \bot$ and $\underline{\sigma}''(f^{(i)}) = \top$, and for all $i \in I$ we have $\underline{\sigma}''(t^{(i)}) = \bot$ and $\underline{\sigma}''(f^{(i)}) = \bot$.

Then by (IH1) we have that $\langle \underline{s_1}, \underline{\sigma}''\rangle \rightarrow^{*} \langle \texttt{skip}, \underline{\sigma}'''\rangle$ for some $\underline{\sigma}'''$ s.t. for all $i \in A_1$ $\sigma'_i \in_i \underline{\sigma}'''$ and $\underline{\sigma}'' \precsim_i^{s_1} \underline{\sigma}'''$, and for all $i \in \{1, \ldots, k\} \setminus A_1$, $\underline{\sigma}'' \preccurlyeq_i^{s_1} \underline{\sigma}'''$ and therefore by Lemma A.1.6, $\sigma_i \in_i \underline{\sigma}'''$.

By (IH2), we have that $\langle \underline{s_2}, \underline{\sigma}'''\rangle \rightarrow^{*} \langle \texttt{skip}, \underline{\sigma}'\rangle$ s.t. for all $i \in A_2$, $\sigma'_i \in_i \underline{\sigma}'$ and $\underline{\sigma}''' \precsim_i^{s_2} \underline{\sigma}'$. Additionally, for all $i \in \{1, \ldots, k\} \setminus A_2$, $\underline{\sigma}''' \preccurlyeq_i^{s_2} \underline{\sigma}'$ and therefore by Lemma A.1.6 we have for all $i \in A_1$ that $\sigma'_i \in_i \underline{\sigma}'$. Since $A = A_1 \cup A_2$, we therefore have $\sigma'_i \in_i \underline{\sigma}'$ for all $i \in A$.

We still need to show that $\underline{\sigma} \precsim_i^{s} \underline{\sigma}'$ for all $i \in A$ and $\underline{\sigma} \preccurlyeq_i^{s} \underline{\sigma}'$ for all $i \in I$:

By Prop. A.1.5 we have $\underline{\sigma} \preccurlyeq_i^{s} \underline{\sigma}''$ for all $i \in \{1, \ldots, k\}$. Additionally, we have $\underline{\sigma}'' \precsim_i^{s_1} \underline{\sigma}'''$ for all $i \in A_1$ and $\underline{\sigma}'' \preccurlyeq_i^{s_1} \underline{\sigma}'''$ for all $i \in \{1, \ldots, k\} \setminus A_1$. Similarly, we have $\underline{\sigma}''' \precsim_i^{s_2} \underline{\sigma}'$ for all $i \in A_2$ and $\underline{\sigma}''' \preccurlyeq_i^{s_2} \underline{\sigma}'$ for all $i \in \{1, \ldots, k\} \setminus A_2$. By repeated use of Prop. A.1.3 we therefore have $\underline{\sigma} \precsim_i^{s} \underline{\sigma}'$.

Since $I \subseteq \{1, \ldots, k\}$ and $I \subseteq (\{1, \ldots, k\} \setminus A_1)$ and $I \subseteq (\{1, \ldots, k\} \setminus A_2)$, we also have $\underline{\sigma} \preccurlyeq_i^{s} \underline{\sigma}'$ for all $i \in I$ by Prop. A.1.3.

- $s \equiv \texttt{while}\,(e)\,\{s_l\}$:

  Then $\underline{s}$ must have the form $\texttt{while}\,(\bigvee_{i=1}^{k}(p^{(i)} \land e^{(i)}))\,\{\bigodot_{i=1}^{k}(t^{(i)}{:=}p^{(i)} \land e^{(i)}); \underline{s_l}\}$ where $\underline{s_l} = [\![s_l]\!]_k^{\mathring{t}}$ and $\mathit{freshvars}(\underline{s}) = \{\mathring{t}\} \cup \mathit{freshvars}(\underline{s_l})$.

  We define $A_l = \{i | i \in A \land e \Downarrow_{\sigma_i} \top\}$ and $I_l = A \setminus A_l$. Then we have that for all $i \in A_l$, $\langle s, \sigma_i\rangle \rightarrow \langle s_l; s, \sigma_i\rangle$ by Whl1 and subsequently $\langle s_l; s, \sigma_i\rangle \rightarrow^{l_i - 1} \langle \texttt{skip}, \sigma'_i\rangle$, and for all $i \in I_l$, $\langle s, \sigma_i\rangle \rightarrow \langle \texttt{skip}, \sigma_i\rangle$ (and therefore $\sigma'_i = \sigma_i$) by Whl2.

  * If $A_l = \emptyset$, then for all $i \in A$, we have $e \Downarrow_{\sigma_i} \bot$. By Lemma 3.4.1 we get that $e^{(i)} \Downarrow_{\underline{\sigma}} \bot$ and since we also have for all $i \in I$ that $p^{(i)} \Downarrow_{\underline{\sigma}} \bot$, we have that $\bigvee_{i=1}^{k}(p^{(i)} \land e^{(i)}) \Downarrow_{\underline{\sigma}} \bot$. Therefore by Whl2 we get that $\langle \underline{s}, \underline{\sigma}\rangle \rightarrow \langle \texttt{skip}, \underline{\sigma}\rangle$, and therefore $\underline{\sigma}' = \underline{\sigma}$. We therefore trivially have $\sigma'_i \in_i \underline{\sigma}'$ and $\underline{\sigma} \precsim_i^{s} \underline{\sigma}'$ for all $i \in A$, and $\underline{\sigma} \preccurlyeq_i^{s} \underline{\sigma}'$ for all $i \in I$.
  * If $A_l \neq \emptyset$, then for at least one $i \in A$, we have $e \Downarrow_{\sigma_i} \top$. By Lemma 3.4.1 this means that $e^{(i)} \Downarrow_{\underline{\sigma}} \top$, and since we

also have $p^{(i)} \Downarrow_{\underline{\sigma}} \top$, we get that $\bigvee_{i=1}^{k}(p^{(i)} \wedge e^{(i)}) \Downarrow_{\underline{\sigma}} \top$. We therefore have that $\underline{s}$ progresses by Whl1 s.t. $\langle \underline{s}, \underline{\sigma} \rangle \rightarrow \langle \bigodot_{i=1}^{k}(t^{(i)}{:=}p^{(i)} \wedge e^{(i)}); \underline{s_l}; \underline{s}, \underline{\sigma} \rangle$. By Lemma A.1.10 and Lemma A.1.11, we have that $\langle \bigodot_{i=1}^{k}(t^{(i)}{:=}p^{(i)} \wedge e^{(i)}), \underline{\sigma} \rangle \rightarrow^{*} \langle \mathtt{skip}, \underline{\sigma}'' \rangle$, where $\underline{\sigma}'' = \underline{\sigma}[t^{(1)} \mapsto v_1] \ldots [t^{(i)} \mapsto v_k]$ s.t. $p^{(i)} \wedge e^{(i)} \Downarrow_{\underline{\sigma}} v_i$. Therefore by Seq1 and Seq2 we get $\langle \bigodot_{i=1}^{k}(t^{(i)}{:=}p^{(i)} \wedge e^{(i)}); \underline{s_l}; \underline{s}, \underline{\sigma} \rangle \rightarrow^{*} \langle \mathtt{skip}; \underline{s_l}; \underline{s}, \underline{\sigma}'' \rangle \rightarrow \langle \underline{s_l}; \underline{s}, \underline{\sigma}'' \rangle$. Note that $v_i = \top \Leftrightarrow i \in A_l$.

By repeated use of Lemma A.1.7.1 we also have that $\langle s, \sigma_i \rangle \rightarrow \langle s_l; s, \sigma_i \rangle \rightarrow^{l_{(i,1)}} \langle \mathtt{skip}; s, \sigma_i''' \rangle \rightarrow \langle s, \sigma_i''' \rangle \rightarrow^{l_{(i,2)}} \langle \mathtt{skip}, \sigma_i' \rangle$, for some $l_{(i,1)}, l_{(i,2)}, \sigma_i'''$ s.t. $l_i = 2 + l_{(i,1)} + l_{(i,2)}$, for every $i \in A_l$. By Lemma A.1.7.2 this implies that $\langle s_l, \sigma_i \rangle \rightarrow^{l_{(i,1)}} \langle \mathtt{skip}, \sigma_i''' \rangle$. We can apply the induction hypothesis to A_l and these subderivations (IH1). We get that $\langle \underline{s_l}, \underline{\sigma}'' \rangle \rightarrow^{*} \langle \mathtt{skip}, \underline{\sigma}''' \rangle$ for some $\underline{\sigma}'''$ s.t. for each $i \in A_l$, $\sigma_i''' \in_i \underline{\sigma}'''$ and $\underline{\sigma}'' \precsim_i^{\underline{s_l}} \underline{\sigma}'''$, which implies that $\underline{\sigma}'''(p^{(i)}) = \top$. For each $i \in A \setminus A_l$, since $i \in I_l$, we have $\underline{\sigma}'' \preccurlyeq_i^{\underline{s_l}} \underline{\sigma}'''$ and therefore $\sigma_i \in_i \underline{\sigma}'''$ by Lemma A.1.6 and $\underline{\sigma}'''(p^{(i)}) = \top$. Furthermore, for each $i \in I$, we have $\underline{\sigma}'' \preccurlyeq_i^{\underline{s_l}} \underline{\sigma}'''$ and therefore $\underline{\sigma}'''(p^{(i)}) = \bot$. We can therefore apply the induction hypothesis again to A and the subderivations $\langle s, \sigma_i''' \rangle \rightarrow^{l_{(i,2)}} \langle \mathtt{skip}, \sigma_i' \rangle$ for all $i \in A_l$ and $\langle s, \sigma_i \rangle \rightarrow^{l_i} \langle \mathtt{skip}, \sigma_i' \rangle$ for all $i \in A \setminus A_l$ (IH2). Since A_l is non-empty, the sum of trace lengths for the resulting set is lower than l.

We know by (IH1) that $\langle \underline{s_l}, \underline{\sigma}'' \rangle \rightarrow^{*} \langle \mathtt{skip}, \underline{\sigma}''' \rangle$ for some $\underline{\sigma}'''$ s.t. $\sigma_i''' \in_i \underline{\sigma}'''$ and $\underline{\sigma}'' \precsim_i^{\underline{s_l}} \underline{\sigma}'''$ for all $i \in A_l$ and $\underline{\sigma}'' \preccurlyeq_i^{\underline{s_l}} \underline{\sigma}'''$ for all $i \in I_l$.

By (IH2) we get that $\langle \underline{s}, \underline{\sigma}''' \rangle \rightarrow^{*} \langle \mathtt{skip}, \underline{\sigma}' \rangle$ s.t. $\sigma_i' \in_i \underline{\sigma}'$ and $\underline{\sigma}''' \precsim_i^{\underline{s}} \underline{\sigma}'$ for all $i \in A$, and $\underline{\sigma}''' \preccurlyeq_i^{\underline{s}} \underline{\sigma}'$ for all $i \in I$. By repeated use of Seq1 and Seq2 we can construct a trace $\langle \underline{s}, \underline{\sigma} \rangle \rightarrow \langle \bigodot_{i=1}^{k}(t^{(i)}{:=}p^{(i)} \wedge e^{(i)}); \underline{s_l}; \underline{s}, \underline{\sigma} \rangle \rightarrow^{*} \langle \mathtt{skip}; \underline{s_l}; \underline{s}, \underline{\sigma}'' \rangle \rightarrow \langle \underline{s_l}; \underline{s}, \underline{\sigma}'' \rangle \rightarrow^{*} \langle \mathtt{skip}; \underline{s}, \underline{\sigma}''' \rangle \rightarrow \langle \underline{s}, \underline{\sigma}''' \rangle \rightarrow^{*} \langle \mathtt{skip}, \underline{\sigma}' \rangle$. We already have that $\sigma_i' \in_i \underline{\sigma}'$ for all $i \in A$.

We still need to show that $\underline{\sigma} \precsim_i^{\underline{s}} \underline{\sigma}'$ for all $i \in A$ and $\underline{\sigma} \preccurlyeq_i^{\underline{s}} \underline{\sigma}'$ for all $i \in I$. We have that $\underline{\sigma} \preccurlyeq_i^{\underline{s}} \underline{\sigma}''$ for all $i \in \{1, \ldots, k\}$. Then for all $i \in A_l$, we have that $\underline{\sigma}'' \precsim_i^{\underline{s_l}} \underline{\sigma}'''$ and $\underline{\sigma}''' \precsim_i^{\underline{s}} \underline{\sigma}'$, so we get that $\underline{\sigma} \precsim_i^{\underline{s}} \underline{\sigma}'$ by Prop. A.1.3. For all $i \in A \setminus A_l$, we have that $\underline{\sigma}'' \preccurlyeq_i^{\underline{s_l}} \underline{\sigma}'''$ and $\underline{\sigma}''' \precsim_i^{\underline{s}} \underline{\sigma}'$, so we get that $\underline{\sigma} \precsim_i^{\underline{s}} \underline{\sigma}'$ by Prop. A.1.3. For all $i \in I$, we have that $\underline{\sigma}'' \preccurlyeq_i^{\underline{s_l}} \underline{\sigma}'''$ and $\underline{\sigma}''' \preccurlyeq_i^{\underline{s}} \underline{\sigma}'$, so we get that $\underline{\sigma} \preccurlyeq_i^{\underline{s}} \underline{\sigma}'$ by Prop. A.1.3.

- $s \equiv x_1, \ldots, x_m{:=}\ m(e_1, \ldots, e_n)$: Then $\underline{s}$ has the form

```
if   $(\bigvee_{i=1}^{k} p^{(i)})$ {
        $\bigodot_{i=1}^{k}$ if $(p^{(i)})$ {$\bigodot_{j=1}^{n}(a_j{}^{(i)}{:=}e_j{}^{(i)})$};
        $ts{:=}\ m(p^{(1)}, \ldots, p^{(k)}, as)$;
        $\bigodot_{i=1}^{k}$ if $(p^{(i)})$ {$\bigodot_{j=1}^{m}(x_j{}^{(i)}{:=}t_j{}^{(i)})$}
}
```

where $\mathring{a}_j$ and $\mathring{t}_j$ are in *freshvars*$(\underline{s})$ and ts =

$[t_1{}^{(1)}, \ldots, t_1{}^{(k)}, \ldots, t_m{}^{(1)}, \ldots, t_m{}^{(k)}]$ and $as = [a_1{}^{(1)}, \ldots, a_1{}^{(k)}, \ldots, a_n{}^{(1)}, \ldots, a_n{}^{(k)}]$. In the following text, we will abbreviate $ass_{as} = \bigodot_{i=1}^{k} \texttt{if}\,(p^{(i)})\,\{\bigodot_{j=1}^{n}(a_j{}^{(i)}{:=}e_j{}^{(i)})\}$ and $c = ts{:=}\,m(p^{(1)}, \ldots, p^{(k)}, as)$ and $ass_{xs} = \bigodot_{i=1}^{k} \texttt{if}\,(p^{(i)})\,\{\bigodot_{j=1}^{m}(x_j{}^{(i)}{:=}t_j{}^{(i)})\}$.

Let $\Phi(m) = ([q_1, \ldots, q_n], [r_1, \ldots, r_m], s_m)$. Then because of $\mathit{match}(\Phi, \underline{\Phi})$, we know $\underline{\Phi}(m) = ([p'^{(1)}, \ldots, p'^{(k)}, q_1{}^{(1)}, \ldots, q_1{}^{(k)}, \ldots, q_n{}^{(1)}, \ldots, q_n{}^{(k)}], rs, \underline{s_m})$ where $\underline{s_m} = [\![s_m]\!]_k^{\mathring{p}'}$ and $rs = [r_1{}^{(1)}, \ldots, r_1{}^{(k)}, \ldots, r_m{}^{(1)}, \ldots, r_m{}^{(k)}]$.

All $i \in A$ must progress via Call, so we must have

$$\langle s, \sigma_i \rangle \longrightarrow \langle x_1, \ldots, x_m{:=}\texttt{frame}_{r_1,\ldots,r_m}(s_m, \sigma_{(i,f)}), \sigma_i \rangle \longrightarrow^{l_i - 1} \langle \texttt{skip}, \sigma_i' \rangle,$$

where $\sigma_{(i,f)} = [q_1 \mapsto v_{(i,1)}] \ldots [q_n \mapsto v_{(i,n)}]$ and $e_j \Downarrow_{\sigma_i} v_{(i,j)}$.

By Lemma A.1.8.1 we get that $\langle s_m, \sigma_{(i,f)} \rangle \longrightarrow^{l_i - 2} \langle \texttt{skip}, \sigma'_{(i,f)} \rangle$ for some $\sigma'_{(i,f)}$ and $\sigma_i' = \sigma_i[x_1 \mapsto \sigma'_{(i,f)}(r_1)] \ldots [x_m \mapsto \sigma'_{(i,f)}(r_m)]$. We apply the induction hypothesis to these derivations and A (IH).

Since $A \neq \emptyset$, we have $p^{(i)} \Downarrow_{\underline{\sigma}} \top$ for some $i \in A$, and therefore $\langle \underline{s}, \underline{\sigma} \rangle \longrightarrow \langle ass_{as}; c; ass_{xs}, \underline{\sigma} \rangle$ by Cond1.

By Lemma 3.4.1 we get that $e_j{}^{(i)} \Downarrow_{\underline{\sigma}} v_{(i,j)}$ for all $i \in A$. By Lemma A.1.12 and Lemma A.1.13 we get that $\langle \bigodot_{i=1}^{k} \texttt{if}\,(p^{(i)})\,\{\bigodot_{j=1}^{n}(a_j{}^{(i)}{:=}e_j{}^{(i)})\}, \underline{\sigma} \rangle \longrightarrow^* \langle \texttt{skip}, \underline{\sigma}'' \rangle$ s.t. $\underline{\sigma}'' = \underline{\sigma}[a_1{}^{(i_1)} \mapsto v_{(i_1,1)}] \ldots [a_n{}^{(i_1)} \mapsto v_{(i_1,n)}] \ldots [a_1{}^{(i_j)} \mapsto v_{(i_j,1)}] \ldots [a_n{}^{(i_j)} \mapsto v_{(i_j,n)}]$ for $\{i_1, \ldots, i_j\} = A$. By repeated use of Seq1 and Seq2 we therefore have $\langle ass_{as}; c; ass_{xs}, \underline{\sigma} \rangle \longrightarrow^* \langle \texttt{skip}; c; ass_{xs}, \underline{\sigma}'' \rangle \longrightarrow \langle c; ass_{xs}, \underline{\sigma}'' \rangle$. Then by Call and Seq1 we get that $\langle c; ass_{xs}, \underline{\sigma}'' \rangle \longrightarrow \langle ts{:=}\texttt{frame}_{rs}(\underline{s_m}, \underline{\sigma}_f); ass_{xs}, \underline{\sigma}'' \rangle$ where

$$\begin{aligned}\underline{\sigma}_f = [&p'^{(1)} \mapsto \underline{\sigma}''(p^{(1)}), \ldots, p'^{(k)} \mapsto \underline{\sigma}''(p^{(k)}),\\ &q_1{}^{(1)} \mapsto v_{(1,1)}, \ldots, q_n{}^{(1)} \mapsto v_{(1,n)}, \ldots,\\ &q_1{}^{(k)} \mapsto v_{(k,1)}, \ldots, q_n{}^{(k)} \mapsto v_{(k,n)}]\end{aligned}$$

and $rs = [r_1{}^{(1)}, \ldots, r_1{}^{(k)}, \ldots, r_m{}^{(1)}, \ldots, r_m{}^{(k)}]$. Then for all $i \in A$ we have that that $\sigma_{(i,f)} \in_i \underline{\sigma}_f$ by the same argument as in the corresponding case of the proof for Thm. 3.4.2.

By (IH) we therefore get that $\langle \underline{s_m}, \underline{\sigma}_f \rangle \longrightarrow^* \langle \texttt{skip}, \underline{\sigma}'_f \rangle$ for some $\underline{\sigma}_f$ s.t. for all $i \in A$ we have $\sigma'_{(i,f)} \in_i \underline{\sigma}'_f$. By repeated use of Frame1 we get that $\langle ts{:=}\texttt{frame}_{rs}(\underline{s_m}, \underline{\sigma}_f), \underline{\sigma}'' \rangle \longrightarrow^* \langle ts{:=}\texttt{frame}_{rs}(\texttt{skip}, \underline{\sigma}'_f), \underline{\sigma}'' \rangle$, and by Frame2 that $\langle ts{:=}\texttt{frame}_{rs}(\texttt{skip}, \underline{\sigma}'_f), \underline{\sigma}'' \rangle \longrightarrow \langle \texttt{skip}, \underline{\sigma}''' \rangle$ where $\underline{\sigma}''' = \underline{\sigma}''[t_1{}^{(1)} \mapsto \underline{\sigma}'_f(r_1{}^{(1)})] \ldots [r_1{}^{(k)} \mapsto \underline{\sigma}'_f(r_1{}^{(k)})] \ldots [t_m{}^{(1)} \mapsto \underline{\sigma}'_f(r_m{}^{(1)})] \ldots [t_m{}^{(k)} \mapsto \underline{\sigma}'_f(r_m{}^{(k)})]$. We get that $\underline{\sigma} \preccurlyeq^{\underline{s}} \underline{\sigma}'''$ by Prop. A.1.5.

By repeated use of Seq1 and Seq2 we get that $\langle ts{:=}\texttt{frame}_{rs}(\underline{s_m}, \underline{\sigma}_f); ass_{xs}, \underline{\sigma}'' \rangle \longrightarrow^* \langle \texttt{skip}; ass_{xs}, \underline{\sigma}''' \rangle$. Then by Lemma A.1.12 and Lemma A.1.13 we get that $\langle ass_{xs}, \underline{\sigma}'' \rangle \longrightarrow^* \langle \texttt{skip}, \underline{\sigma}' \rangle$ for some $\underline{\sigma}'$ s.t. $\underline{\sigma}' =$

$\underline{\sigma}'''[{x_1}^{(i_1)} \mapsto \underline{\sigma}'_f({r_1}^{(i_1)})] \dots [{x_1}^{(i_j)} \mapsto \underline{\sigma}'_f({r_1}^{(i_j)})] \dots [{x_m}^{(i_1)} \mapsto \underline{\sigma}'_f({r_m}^{(i_1)})] \dots [{x_m}^{(i_j)} \mapsto \underline{\sigma}'_f({r_m}^{(i_j)})]$ where $\{i_1, \dots, i_j\} = A$, and we have $\sigma'_i \in_i \underline{\sigma}'$ for all $i \in A$ by repeated use of Prop. A.1.1 and Prop. A.1.2. Since we have that $\underline{\sigma}' = \underline{\sigma}[{a_1}^{(i_1)} \mapsto \_] \dots [{a_n}^{(i_1)} \mapsto \_] \dots [{a_1}^{(i_j)} \mapsto \_] \dots [{a_n}^{(i_j)} \mapsto \_][{x_1}^{(i_1)} \mapsto \_] \dots [{x_1}^{(i_j)} \mapsto \_] \dots [{x_m}^{(i_1)} \mapsto \_] \dots [{x_m}^{(i_j)} \mapsto \_]$, and all ${a_j}^{(i)} \in \mathit{freshvars}(\underline{s})$ and all ${x_j}^{(i)} \in$ RPVar we also have that $\underline{\sigma} \lesssim_i^{\underline{s}} \underline{\sigma}'$ for all $i \in A$ by Prop. A.1.5. Similarly, for all $i' \in I$, we have that $\underline{\sigma} \preccurlyeq_{i'}^{\underline{s}} \underline{\sigma}'$ by Prop. A.1.5 and Prop. A.1.4 since all ${a_j}^{(i)} \in \mathit{freshvars}(\underline{s})$ and there is no $i \in A$ s.t. $i = i'$.

- $s \equiv s_1; s_2$: Then $\underline{s}$ has the form $\underline{s_1}; \underline{s_2}$, where $\underline{s_1} = [\![s_1]\!]_k^{\mathring{p}}$ and $\underline{s_2} = [\![s_2]\!]_k^{\mathring{p}}$. By Lemma A.1.7.1 we get that for all $i \in A$, $\langle s, \sigma_i \rangle \rightarrow^{l_{(i,1)}} \langle \texttt{skip}; s_2, \sigma''_i \rangle \rightarrow \langle s_2, \sigma''_i \rangle \rightarrow^{l_{(i,2)}} \langle \texttt{skip}, \sigma'_i \rangle$ for some $l_{(i,1)}, l_{(i,2)}, \sigma''_i$ s.t. $l_i = 1 + l_{(i,1)} + l_{(i,2)}$. By Lemma A.1.7.2 we get that $\langle s_1, \sigma_i \rangle \rightarrow^{l_{(i,1)}} \langle \texttt{skip}, \sigma''_i \rangle$.
  We apply the induction hypothesis to A and the derivations $\langle s_1, \sigma_i \rangle \rightarrow^{l_{(i,1)}} \langle \texttt{skip}, \sigma''_i \rangle$ (IH1). We get that $\langle \underline{s_1}, \underline{\sigma} \rangle \rightarrow^* \langle \texttt{skip}, \underline{\sigma}'' \rangle$ for some $\underline{\sigma}''$ s.t. $\sigma''_i \in_i \underline{\sigma}''$ and $\underline{\sigma} \lesssim_i^{\underline{s_1}} \underline{\sigma}''$ for all $i \in A$, and $\underline{\sigma} \preccurlyeq_i^{\underline{s_1}} \underline{\sigma}''$ for all $i \in I$. Therefore $\underline{\sigma}''(p^{(i)}) = \underline{\sigma}(p^{(i)})$ for all $i \in \{1, \dots, k\}$. By repeated use of Seq1 and a single use of Seq2 we get that $\langle \underline{s}, \underline{\sigma} \rangle \rightarrow^* \langle \texttt{skip}; \underline{s_2}, \underline{\sigma}'' \rangle \rightarrow \langle \underline{s_2}, \underline{\sigma}'' \rangle$.
  Subsequently, we apply the induction hypothesis to A and the derivations $\langle s_2, \sigma''_i \rangle \rightarrow^{l_{(i,2)}} \langle \texttt{skip}, \sigma'_i \rangle$. We get that $\langle \underline{s_2}, \underline{\sigma}'' \rangle \rightarrow^* \langle \texttt{skip}, \underline{\sigma}' \rangle$ for some $\underline{\sigma}'$ s.t. $\sigma'_i \in_i \underline{\sigma}'$ and $\underline{\sigma}'' \lesssim_i^{\underline{s_2}} \underline{\sigma}'$ for all $i \in A$, and $\underline{\sigma}'' \preccurlyeq_i^{\underline{s_2}} \underline{\sigma}'$ for all $i \in I$. We use Prop. A.1.3 to get $\underline{\sigma} \lesssim_i^{\underline{s}} \underline{\sigma}'$ for all $i \in A$. We also use Prop. A.1.3 to get $\underline{\sigma} \preccurlyeq_i^{\underline{s}} \underline{\sigma}'$ for all $i \in I$.

□

A.1.5. Proof of Lemma 3.4.6

Lemma A.1.20 *Assume that for a set of indices A s.t. $|A| = j$ and $1 \leq j \leq k$ and $A \subseteq \{1, \dots, k\}$ there is a derivation $d_i = \langle s, \sigma_i \rangle \rightarrow^{l_i} \langle \dot{s}_i, \sigma'_i \rangle$, where $\dot{s}_i \in \{\texttt{skip}, \texttt{error}, \texttt{magic}\}$, under Φ for each $i \in A$. Assume given $A_s \subset A$ and $A_e = A \setminus A_s$ s.t. for all $i \in A_e$, $\dot{s}_i \neq \texttt{skip}$, and for all $i \in A_s$, $\dot{s}_i = \texttt{skip}$. Assume also that $\sigma_i \in_i \underline{\sigma}$ and $\underline{\sigma}(p^{(i)}) = \top$ for all $i \in A$, and $\underline{\sigma}(p^{(i)}) = \bot$ for any $i \in I$, where $I = \{1, \dots, k\} \setminus A$.*

Then $\langle \underline{s}, \underline{\sigma} \rangle \rightarrow^* \langle \dot{s}, \underline{\sigma}' \rangle$ under $\underline{\Phi}$, where $\underline{s} = [\![s]\!]_k^{\mathring{p}}$, for some $\underline{\sigma}'$ and $\dot{s}$ s.t. $\exists i \in A_e. \dot{s}_i = \dot{s}$

Proof. The proof goes by strong induction on the sum of the lengths of traces $l = \sum_{i \in A} l_i$.

- Case $l = 0$: This case is impossible. If $j = 0$ then $A = \emptyset$ and we cannot have that $A_s \subset A$. If $j > 0$ we must have that $s = \texttt{skip}$ and therefore all $\dot{s}_i = \texttt{skip}$, and we again cannot have that $A_s \subset A$.
- Case $l > 0$: We perform a case split on the structure of s.
  - $s \equiv x{:=}e$: This case is impossible, since all d_i must progress by Assign, and then $\dot{s}_i = \texttt{skip}$ for all $i \in A$.

- $s \equiv \texttt{havoc}\ x$: This case is impossible, since all d_i must progress by Havoc, and then $\dot{s}_i = \texttt{skip}$ for all $i \in A$.
- $s \equiv \texttt{assert}\ e$: Then all $i \in A_s$ must progress by Assert1 and we must therefore have $e \Downarrow_{\sigma_i} \top$ and by Lemma 3.4.1 $e^{(i)} \Downarrow_{\underline{\sigma}} \top$. Conversely, all $i \in A_e$ must progress by Assert2 and we must therefore have $e \Downarrow_{\sigma_i} \bot$, and by Lemma 3.4.1 $e^{(i)} \Downarrow_{\underline{\sigma}} \bot$.
  $\underline{s}$ must have the form $\bigodot_{i=1}^{k} \texttt{if}\,(p^{(i)})\,\{\texttt{assert}\ e^{(i)}\}$, and by Lemma A.1.17, since $A_e \neq \emptyset$, we have that $\langle \underline{s}, \underline{\sigma} \rangle \rightarrow^* \langle \texttt{error}, \underline{\sigma} \rangle$.
- $s \equiv \texttt{assume}\ e$: Then all $i \in A_s$ must progress by Assume1 and we must therefore have $e \Downarrow_{\sigma_i} \top$ and by Lemma 3.4.1 $e^{(i)} \Downarrow_{\underline{\sigma}} \top$. Conversely, all $i \in A_e$ must progress by Assume2 and we must therefore have $e \Downarrow_{\sigma_i} \bot$, and by Lemma 3.4.1 $e^{(i)} \Downarrow_{\underline{\sigma}} \bot$.
  $\underline{s}$ must have the form $\bigodot_{i=1}^{k} \texttt{if}\,(p^{(i)})\,\{\texttt{assume}\ e^{(i)}\}$, and by Lemma A.1.19, since $A_e \neq \emptyset$, we have that $\langle \underline{s}, \underline{\sigma} \rangle \rightarrow^* \langle \texttt{magic}, \underline{\sigma} \rangle$.
- $s \equiv \texttt{if}\,(e)\,\{s_1\}\,\texttt{else}\,\{s_2\}$:
  Then $\underline{s}$ has the form $\bigodot_{i=1}^{k}(t^{(i)}{:=}p^{(i)} \wedge e^{(i)}); \bigodot_{i=1}^{k}(f^{(i)}{:=}p^{(i)} \wedge \neg e^{(i)}); \underline{s_1}; \underline{s_2}$, where $\underline{s_1} = [\![s_1]\!]_k^{\mathring{t}}$ and $\underline{s_2} = [\![s_2]\!]_k^{\mathring{f}}$ and $\mathring{t}$ and $\mathring{f}$ are fresh variable names.
  Let $A_1 \subseteq A$ and $A_2 = A \setminus A_1$ s.t. all $i \in A_1$ progress by Cond1 and we therefore have $e \Downarrow_{\sigma_i} \top$ and by Lemma 3.4.1 $e^{(i)} \Downarrow_{\underline{\sigma}} \top$, and all $i \in A_2$ progress by Cond2 and we therefore have $e \Downarrow_{\sigma_i} \bot$ and by Lemma 3.4.1 $e^{(i)} \Downarrow_{\underline{\sigma}} \bot$.
  By Lemma A.1.10 and Lemma A.1.11 and repeated use of Seq1 and Seq2 we must have that $\langle \underline{s}, \underline{\sigma} \rangle \rightarrow^* \langle \underline{s_1}; \underline{s_2}, \underline{\sigma}'' \rangle$, where $\underline{\sigma}'' = \underline{\sigma}[t^{(1)} \mapsto v_1] \ldots [t^{(k)} \mapsto v_k]$ s.t. $p^{(i)} \wedge e^{(i)} \Downarrow_{\underline{\sigma}} v_i$. In particular, we have that for all $i \in I$, $\underline{\sigma}''(t^{(i)}) = \bot$ and $\underline{\sigma}''(f^{(i)}) = \bot$; for all $i \in A_1$, we have $\underline{\sigma}''(t^{(i)}) = \top$ and $\underline{\sigma}''(f^{(i)}) = \bot$, and for all $i \in A_2$, we have $\underline{\sigma}''(t^{(i)}) = \bot$ and $\underline{\sigma}''(f^{(i)}) = \top$.
  * If for some non-empty $A_1' \subseteq A_1$ we have for all $i \in A_1'$, $\langle s_1, \sigma_i \rangle \rightarrow^* \langle \dot{s}_i, \sigma_i' \rangle$ where $\dot{s}_i \neq \texttt{skip}$, then by the induction hypothesis we get that $\langle \underline{s_1}, \underline{\sigma}'' \rangle \rightarrow^* \langle \dot{s}_i, \underline{\sigma}' \rangle$ for some $\underline{\sigma}'$, i and we are done by Seq3 or Seq4.
  * If there is no such $i \in A_2$, we must have for some non-empty $A_2' \in A_2$ that for all $i \in A_2'$, $\langle s_2, \sigma_i \rangle \rightarrow^* \langle \dot{s}_i, \sigma_i' \rangle$ where $\dot{s}_i \neq \texttt{skip}$. In this case, by Thm. 3.4.5 we get $\langle \underline{s_1}, \underline{\sigma}'' \rangle \rightarrow^* \langle \texttt{skip}, \underline{\sigma}''' \rangle$ for some $\underline{\sigma}'''$ s.t. $\underline{\sigma}'''(f^{(i)}) = \top$ and $\sigma_i \in_i \underline{\sigma}'''$ for all $i \in A_2$, and $\underline{\sigma}'''(f^{(i)}) = \bot$ for all $i \in \{1, \ldots, l\} \setminus A_2$. By repeated use of Seq1 and Seq2 this gives us $\langle \underline{s_1}; \underline{s_2}, \underline{\sigma}'' \rangle \rightarrow^* \langle \texttt{skip}; \underline{s_2}, \underline{\sigma}''' \rangle \rightarrow \langle \underline{s_2}, \underline{\sigma}''' \rangle$. Subsequently, by the induction hypothesis, we get $\langle \underline{s_2}, \underline{\sigma}''' \rangle \rightarrow^* \langle \dot{s}_i, \underline{\sigma}' \rangle$, for some i and we are done again.
- $s \equiv \texttt{while}\,(e)\,\{s_l\}$:
  Then $\underline{s}$ must have the form $\texttt{while}\,(\bigvee_{i=1}^{k}(p^{(i)} \wedge e^{(i)}))\,\{\bigodot_{i=1}^{k}(t^{(i)}{:=}p^{(i)} \wedge e^{(i)}); \underline{s_l}\}$ where $\underline{s_l} = [\![s_l]\!]_k^{\mathring{t}}$ and $\mathit{freshvars}(\underline{s}) = \{\mathring{t}\} \cup \mathit{freshvars}(\underline{s_l})$.
  Then we must have that for some non-empty $A_t \subseteq A$, all $i \in A_t$ progress by Whl1, since otherwise for all $i \in A$ we have $\dot{s}_i = \texttt{skip}$ by Whl2. We therefore have $\langle s, \sigma_i \rangle \rightarrow \langle s_l; s, \sigma_i \rangle \rightarrow^{l_i - 1} \langle \dot{s}_i, \sigma_i' \rangle$ and $e \Downarrow_{\sigma_i} \top$ for all $i \in A_t$, and $\langle s, \sigma_i \rangle \rightarrow \langle \texttt{skip}, \sigma_i \rangle$

and $e \Downarrow_{\sigma_i} \bot$ for all $i \in A \setminus A_t$.
By Lemma 3.4.1 we have $e^{(i)} \Downarrow_{\underline{\sigma}} \top$ for all $i \in A_t$. Since we also have $p^{(i)} \Downarrow_{\underline{\sigma}} \top$ for all $i \in A$, we have that $\bigvee_{i=1}^{k}(p^{(i)} \wedge e^{(i)}) \Downarrow_{\underline{\sigma}} \top$ and therefore $\langle \underline{s}, \underline{\sigma} \rangle \rightarrow \langle \bigvee_{i=1}^{k}(p^{(i)} \wedge e^{(i)}); \underline{s_l}; \underline{s}, \underline{\sigma} \rangle$ by Whl1.
By Lemma A.1.7.3 we get that for each $i \in A_t \cap A_e$, either $\langle s_l, \sigma_i \rangle \rightarrow^{*} \langle \dot{s}, \sigma'_i \rangle$ or $\langle s_l, \sigma_i \rangle \rightarrow^{*} \langle \texttt{skip}, \sigma''_i \rangle \wedge \langle s, \sigma''_i \rangle \rightarrow^{*} \langle \dot{s}_i, \sigma'_i \rangle$ for some σ''_i and some $\dot{s}_i \neq \texttt{skip}$. Let $A_{t_1} \subseteq A_t \cap A_e$ be the set of executions for which the former is true.

* If A_{t_1} is non-empty, then by the induction hypothesis, $\langle \underline{s_l}, \underline{\sigma}'' \rangle \rightarrow^{*} \langle \dot{s}_i, \underline{\sigma}' \rangle$ for some $\underline{\sigma}', i$, and we are done by Seq3 or Seq4.
* If A_{t_1} is empty, then by Thm. 3.4.5 we get $\langle \underline{s_l}, \underline{\sigma}'' \rangle \rightarrow^{*} \langle \texttt{skip}, \underline{\sigma}''' \rangle$ for some $\underline{\sigma}'''$ s.t. $\underline{\sigma}'''(p^{(i)}) = \bot$ for all $i \in I$, $\underline{\sigma}'''(p^{(i)}) = \top$ and $\sigma''_i \in_i \underline{\sigma}'''$ for all $i \in A_t$, and $\underline{\sigma}'''(p^{(i)}) = \top$ and $\sigma_i \in_i \underline{\sigma}'''$ for all $i \in A \setminus A_t$. We must then have $\langle s, \sigma''_i \rangle \rightarrow^{*} \langle \dot{s}_i, \sigma'_i \rangle$ for some $\dot{s}_i \neq \texttt{skip}$ for each i in some non-empty $A_{t_2} \subseteq A_t \cap A_e$ and $\langle s, \sigma''_i \rangle \rightarrow^{*} \langle \texttt{skip}, \sigma'_i \rangle$ for each $i \in A_t \setminus A_{t_2}$. By the induction hypothesis we get that $\langle \underline{s}, \underline{\sigma}''' \rangle \rightarrow^{*} \langle \dot{s}_i, \underline{\sigma}' \rangle$ for some $\underline{\sigma}', i$, and we are done.

- $s \equiv x_1, \ldots, x_m := m(e_1, \ldots, e_n)$: Then $\underline{s}$ has the form

```
if  $(\bigvee_{i=1}^{k} p^{(i)})$ {
        $\bigodot_{i=1}^{k}$ if $(p^{(i)})$ {$\bigodot_{j=1}^{n}(\mathring{a}_j^{(i)} := \mathring{e}_j^{(i)})$};
        $ts := m(p^{(1)}, \ldots, p^{(k)}, as)$;
        $\bigodot_{i=1}^{k}$ if $(p^{(i)})$ {$\bigodot_{j=1}^{m}(x_j^{(i)} := \mathring{t}_j^{(i)})$}
}
```

where $\mathring{a}_j$ and $\mathring{t}_j$ are in *freshvars*$(\underline{s})$ and $ts = [t_1^{(1)}, \ldots, t_1^{(k)}, \ldots, t_m^{(1)}, \ldots, t_m^{(k)}]$ and $as = [a_1^{(1)}, \ldots, a_1^{(k)}, \ldots, a_n^{(1)}, \ldots, a_n^{(k)}]$.
Each d_i must progress by Call s.t.

$$d_i = \langle s, \sigma_i \rangle \rightarrow \langle x_1, \ldots, x_m{:=}\texttt{frame}_{r_1,\ldots,r_m}(s_m, \sigma_{(i,f)}), \sigma_i \rangle \rightarrow^{l_i - 1} \langle \dot{s}_i, \sigma'_i \rangle.$$

By Lemma A.1.8.2 and Lemma A.1.8.1 we know that $\langle s_m, \sigma_{(i,f)} \rangle \rightarrow^{l_i-2} \langle \dot{s}_i, \sigma'_{(i,f)} \rangle$ for some $\sigma'_{(i,f)}$ and some $\dot{s}_i \neq \texttt{skip}$ for all $i \in A_e$, and $\langle s_m, \sigma_{(i,f)} \rangle \rightarrow^{l_i-2} \langle \texttt{skip}, \sigma'_{(i,f)} \rangle$ for some $\sigma'_{(i,f)}$ for all $i \in A_s$. By the induction hypothesis we get that for all $\underline{\sigma}_f$ s.t. $\sigma_{(i,f)} \in_i \underline{\sigma}_f$ and $\underline{\sigma}_f(p'^{(i)}) = \top$ for all $i \in A$ and $\underline{\sigma}_f(p'^{(i)}) = \bot$ for all $i \in I$, we have $\langle \underline{s_m}, \underline{\sigma}_f \rangle \rightarrow^{*} \langle \dot{s}_i, \underline{\sigma}'_f \rangle$ for some $\underline{\sigma}'_f$ and some i.
By the same argument as in the proof for Thm. 3.4.5, we get for some $xs, rs, \underline{\sigma}''$ and some $\underline{\sigma}_f$ that fulfills these conditions that $\langle \underline{s}, \underline{\sigma} \rangle \rightarrow^{*} \langle xs{:=}\texttt{frame}_{rs}(\underline{s_m}, \underline{\sigma}_f), \underline{\sigma}'' \rangle$. Therefore by repeated use of Frame1 and Frame3 or Frame4 we are done.
- $s \equiv s_1; s_2$: For all $i \in A_s$, we have by Lemma A.1.7.1 that $\langle s, \sigma_i \rangle \rightarrow^{l_{i,1}} \langle \texttt{skip}; s_2, \sigma''_i \rangle \rightarrow \langle s_2, \sigma''_i \rangle \rightarrow^{l_{i,2}} \langle \texttt{skip}, \sigma'_i \rangle$ for some σ''_i, and therefore by Lemma A.1.7.2 that $\langle s_1, \sigma_i \rangle \rightarrow^{l_{i,1}} \langle \texttt{skip}, \sigma''_i \rangle$. On the other hand, for all $i \in A_e$ we have by Lemma A.1.7.3 that either $\langle s_1, \sigma_i \rangle \rightarrow^{*} \langle \dot{s}_i, \sigma'_i \rangle$ or $\langle s_1, \sigma_i \rangle \rightarrow^{*} \langle \texttt{skip}, \sigma''_i \rangle \wedge \langle s_2, \sigma''_i \rangle \rightarrow^{*} \langle \dot{s}_i, \sigma'_i \rangle$, for some σ''_i and some $\dot{s}_i \neq \texttt{skip}$.

- If the former is the case for at least one $i \in A_e$, then by the induction hypothesis, we get that $\langle \underline{s_1}, \underline{\sigma} \rangle \rightarrow^* \langle \dot{s}_i, \underline{\sigma}' \rangle$ for some i and by SEQ3 or SEQ4 we are done.
- Otherwise, by Thm. 3.4.5 we get that $\langle \underline{s_1}, \underline{\sigma} \rangle \rightarrow^* \langle \mathtt{skip}, \underline{\sigma}'' \rangle$ for some $\underline{\sigma}''$ s.t. $\sigma''_i \in_i \underline{\sigma}''$ and $\underline{\sigma}''(p^{(i)}) = \top$ for all $i \in A$, and $\underline{\sigma}''(p^{(i)}) = \bot$ for all $i \in \{1, \ldots, k\} \setminus A$. Then by the induction hypothesis we get $\langle \underline{s_2}, \underline{\sigma}'' \rangle \rightarrow^* \langle \dot{s}_i, \underline{\sigma}' \rangle$ for some i and we are done.

□

A.1.6. Proof of Thm. 3.5.2

In this section, we consider programs written in our ordinary language but with all `while` loops being annotated with termination annotations of the form `while` (e) `terminates`(e_c, e_r) $\{s\}$ (but unchanged operational semantics). Since we prove our technique correct with respect to non-termination resulting from loops, we consider only programs that do not contain (mutually) recursive method calls, since those could result in non-termination resulting from infinite recursion.

We first define some terminology:

- We say that s *always terminates* from σ if there exists some n s.t. if $\langle s, \sigma \rangle \rightarrow^k \langle s', \sigma' \rangle$ for some s', σ' and k then $k \leq n$.
- We say that s *never terminates* from σ if for any trace $\langle s, \sigma \rangle \rightarrow^k \langle \hat{s}, \sigma' \rangle$ we have that $\hat{s}$ is not final.
- We say that s *does not fail from* σ if for all $\dot{s}, \sigma'$ s.t. $\langle s, \sigma \rangle \rightarrow^* \langle \dot{s}, \sigma' \rangle$ we have $\dot{s} \neq \mathtt{error}$.
- We say that the termination of s *coincides* from stores σ_1, σ_2 if s either always terminates from both σ_1 and σ_2 or never terminates from both σ_1 and σ_2.

As noted in Section 3.5.5, our definition of the termination instrumentation slightly abuses the notation in that it uses `assert` statements with information flow assertions $\tilde{P}$ instead of expressions e for convenience. Since the intention is that in the product, said assertions assert the encoded version of the information flow assertion, we define

$$[\![\mathtt{assert}\ \tilde{P}]\!]_2^{\mathring{p}} = \mathtt{assert}\ \lceil \tilde{P} \rceil^{\mathring{p}}$$

where we use the following encoding of information flow assertions $\tilde{P}$ into expressions e:

$$\begin{array}{lcl}
\lceil e \rceil^{\mathring{p}} & = & (\neg p^{(1)} \lor e^{(1)}) \land (\neg p^{(2)} \land e^{(2)}) \\
\lceil \mathit{low}(e) \rceil^{\mathring{p}} & = & \neg(p^{(1)} \land p^{(2)}) \lor e^{(1)} = e^{(2)} \\
\lceil \mathit{lowEvent} \rceil^{\mathring{p}} & = & p^{(1)} = p^{(2)} \\
\lceil \tilde{P}_1 \land \tilde{P}_2 \rceil^{\mathring{p}} & = & \lceil \tilde{P}_1 \rceil^{\mathring{p}} \land \lceil \tilde{P}_2 \rceil^{\mathring{p}} \\
\lceil e_1 \Rightarrow e_2 \rceil^{\mathring{p}} & = & (\neg(p^{(1)} \land {e_1}^{(1)}) \lor {e_2}^{(1)}) \land (\neg(p^{(2)} \land {e_1}^{(2)}) \lor {e_2}^{(2)}) \\
\lceil e_1 \Rightarrow \mathit{low}(e_2) \rceil^{\mathring{p}} & = & \neg(p^{(1)} \land p^{(2)} \land {e_1}^{(1)} \land {e_1}^{(2)}) \lor {e_2}^{(1)} = {e_2}^{(2)} \\
\lceil e \Rightarrow \mathit{lowEvent} \rceil^{\mathring{p}} & = & \neg((p^{(1)} \land e^{(2)}) \lor (p^{(2)} \land e^{(2)})) \lor p^{(1)} = p^{(2)} \\
\lceil \mathit{low}(e_1) \Rightarrow \mathit{low}(e_2) \rceil^{\mathring{p}} & = & \neg(p^{(1)} \land p^{(2)} \land {e_1}^{(1)} = {e_1}^{(2)}) \lor {e_2}^{(1)} = {e_2}^{(2)})
\end{array}$$

Note that this encoding is equivalent to the one presented in Fig. 3.9 (except that quantification, which is not used in the termination instrumentation, is not supported) in the sense that if $\underline{\sigma} \models \lceil \tilde{P} \rceil^{\mathring{p}}$ according to Fig. 3.9, then $\lceil \tilde{P} \rceil^{\mathring{p}} \Downarrow_{\underline{\sigma}} \top$ according to this encoding.

As a result, the originally presented loop instrumentation

```
$term(w, c)$  =  $cond := e_c;$
                 assert $\neg cond \Rightarrow lowEvent;$
                 assert $low(cond);$
                 assert $cond \Rightarrow e_r \geq 0;$
                 assert $c \Rightarrow cond;$
                 assert $\neg cond \Rightarrow e;$
                 while $(e)$
                  do {
                    if $(cond)$ $\{rank := e_r\};$
                    $term(s, cond);$
                    assert $cond \Rightarrow 0 \leq e_r \wedge e_r < rank$
                    assert $\neg cond \Rightarrow e;$
                 }
```

will be encoded as follows in the product:

```
$[\![w]\!]_2^{\mathring{p}} \equiv$  if $(p^{(1)})$ $\{cond^{(1)} := e_c{}^{(1)}\}$
        if $(p^{(2)})$ $\{cond^{(2)} := e_c{}^{(2)}\}$
        assert $\neg((p^{(1)} \wedge \neg cond^{(1)}) \vee (p^{(2)} \wedge \neg cond^{(2)})) \vee p^{(1)} = p^{(2)}$
        assert $\neg(p^{(1)} \wedge p^{(2)}) \vee cond^{(1)} = cond^{(2)}$
        assert $(\neg(p^{(1)} \wedge cond^{(1)}) \vee (e_r{}^{(1)} \geq 0)) \wedge$
               $(\neg(p^{(2)} \wedge cond^{(2)}) \vee (e_r{}^{(2)} \geq 0))$
        assert $(\neg(p^{(1)} \wedge c^{(1)}) \vee cond^{(1)}) \wedge (\neg(p^{(2)} \wedge c^{(2)}) \vee cond^{(2)})$
        assert $(\neg(p^{(1)} \wedge \neg cond^{(1)}) \vee e^{(1)}) \wedge (\neg(p^{(2)} \wedge \neg cond^{(2)}) \vee e^{(2)})$
         while  $(p^{(1)} \wedge e^{(1)} \vee p^{(2)} \wedge e^{(2)})$
          do {
               $p_l{}^{(1)} := p^{(1)} \wedge e^{(1)}$
               $p_l{}^{(2)} := p^{(2)} \wedge e^{(2)}$
               $p_c{}^{(1)} := p_l{}^{(1)} \wedge cond^{(1)}$
               $p_c{}^{(2)} := p_l{}^{(2)} \wedge cond^{(2)}$
               if $(p_c{}^{(1)})$ $\{rank^{(1)} := e_r{}^{(1)}\}$
               if $(p_c{}^{(2)})$ $\{rank^{(2)} := e_r{}^{(2)}\}$
               $[\![term(s, cond)]\!]_2^{\mathring{p}_l}$
               assert $(\neg(p^{(1)} \wedge cond^{(1)}) \vee (0 \leq e_r{}^{(1)} \wedge e_r{}^{(1)} < rank^{(1)})) \wedge$
                      $(\neg(p^{(2)} \wedge cond^{(2)}) \vee (0 \leq e_r{}^{(2)} \wedge e_r{}^{(2)} < rank^{(2)}))$
               assert $(\neg(p^{(1)} \wedge \neg cond^{(1)}) \vee (e^{(1)})) \wedge$
                      $(\neg(p^{(2)} \wedge \neg cond^{(2)}) \vee (e^{(2)}))$
        }
```

We start by proving that if the ordinary product of a statement terminates from some store, then the product of the instrumentation of this statement either terminates from the same store in an equivalent state (which is identical modulo added fresh variables), or it fails (due to one of the assertions added in the instrumentation):

Lemma A.1.20 *If* $\underline{s} = [\![s]\!]_2^{\mathring{p}}$ *and* $\underline{s'} = [\![term(s, cond)]\!]_2^{\mathring{p}}$ *and* $\langle \underline{s}, \underline{\sigma} \rangle \to^k \langle \dot{s}, \underline{\sigma'} \rangle$ *then* $\langle \underline{s'}, \underline{\sigma} \rangle \to^* \langle \dot{s}', \underline{\sigma''} \rangle$ *for some* $\underline{\sigma''}, \dot{s}'$ *s.t. either* $\dot{s}' = \dot{s} \wedge \underline{\sigma''} \lesssim^{\underline{s'}} \underline{\sigma'}$ *or* $\dot{s}' = \texttt{error}$.

Proof. The proof goes by strong induction on k. After a case split on the structure of s, the cases for assignments, havocs, asserts and assumes are trivial because $\underline{s} \equiv \underline{s'}$. The cases for sequential composition and conditionals follow directly from applying the induction hypothesis on the traces of the substatements. For the loop case, in any single iteration either an added assertion in $\underline{s'}$ fails, in which case $\underline{s'}$ ends in an error state, or the added assertions succeed, in which case the conclusion follows from applying the induction hypothesis to the traces of the statement in the loop body and the next iteration. The additional variables assigned to in $\underline{s'}$ are fresh and therefore preserve that $\underline{\sigma''} \lesssim^{\underline{s'}} \underline{\sigma'}$. □

We first prove that in the conditions ensured by the initial assertions of our loop encoding, if the termination condition is true, executions of the actual program must always terminate:

Lemma A.1.21 *Assume that*

```
$w$  =  while ($e$)
         do {
           if ($cond$) {$rank$:=$e_r$};
           $term(s, cond)$;
           assert $cond \Rightarrow 0 \leq e_r \wedge e_r < rank$
           assert $\neg cond \Rightarrow e$
        }
```

and $\underline{w} = [\![w]\!]_2^{\mathring{p}}$, $\underline{w}$ *never fails from* $\underline{\sigma}$, *either* $p^{(1)} \Downarrow_{\underline{\sigma}} \bot$ *or* $\sigma_1 \in_1 \underline{\sigma} \wedge cond^{(1)} \Downarrow_{\underline{\sigma}} \top$, *either* $p^{(2)} \Downarrow_{\underline{\sigma}} \bot$ *or* $\sigma_2 \in_2 \underline{\sigma} \wedge cond^{(2)} \Downarrow_{\underline{\sigma}} \top$, s *never terminates with* `magic` *from* σ_1 *and* σ_2, $term(s, cond)$ *does not assign to cond or rank, and for all* $i \in \{1, 2\}$, $\underline{\sigma''}, \sigma_i''$, *if* $cond^{(i)} \Downarrow_{\underline{\sigma''}} \top$ *and* $p_l{}^{(i)} \Downarrow_{\underline{\sigma''}} \top$ *and* $\sigma_i'' \in_i \underline{\sigma''}$ *and* $[\![term(s, cond)]\!]_2^{\mathring{p}_l}$ *does not fail from* $\underline{\sigma''}$, *then* s *always terminates from* σ_i''.

Then

1. *if* $p^{(1)} \Downarrow_{\underline{\sigma}} \top$ *and* $e_r{}^{(1)} \Downarrow_{\underline{\sigma}} v_1$ *for some* $v_1 \geq 0$ *then* `while` $(e)\ \{s\}$ *always terminates from* σ_1
2. *if* $p^{(2)} \Downarrow_{\underline{\sigma}} \top$ *and* $e_r{}^{(2)} \Downarrow_{\underline{\sigma}} v_2$ *for some* $v_2 \geq 0$ *then* `while` $(e)\ \{s\}$ *always terminates from* σ_2

Proof. We show the proof for (1), the proof for (2) is completely analogous. We have $p^{(1)} \Downarrow_{\underline{\sigma}} \top$ and therefore $\sigma_1 \in_1 \underline{\sigma}$. We show that there is some n s.t. all traces from σ_1 terminate within at most n steps. The proof goes by strong induction on v_1.

$\underline{w}$ must have the form

```
while  $(p^{(1)} \land e^{(1)} \lor p^{(2)} \land e^{(2)})$
 do {
        ${p_l}^{(1)}{:=}p^{(1)} \land e^{(1)}$
        ${p_l}^{(2)}{:=}p^{(2)} \land e^{(2)}$
        ${p_c}^{(1)}{:=}{p_l}^{(1)} \land \mathit{cond}^{(1)}$
        ${p_c}^{(2)}{:=}{p_l}^{(2)} \land \mathit{cond}^{(2)}$
        if $({p_c}^{(1)})$ $\{\mathit{rank}^{(1)}{:=}{e_r}^{(1)}\}$
        if $({p_c}^{(2)})$ $\{\mathit{rank}^{(2)}{:=}{e_r}^{(2)}\}$
        $[\![\mathit{term}(s, \mathit{cond})]\!]_2^{\mathring{p}_l}$
        assert $(\neg(p^{(1)} \land \mathit{cond}^{(1)}) \lor (0 \leq {e_r}^{(1)} \land {e_r}^{(1)} < \mathit{rank}^{(1)}))\land$
                $(\neg(p^{(2)} \land \mathit{cond}^{(2)}) \lor (0 \leq {e_r}^{(2)} \land {e_r}^{(2)} < \mathit{rank}^{(2)}))$
        assert $(\neg(p^{(1)} \land \neg\mathit{cond}^{(1)}) \lor (e^{(1)})) \land (\neg(p^{(2)} \land \neg\mathit{cond}^{(2)}) \lor (e^{(2)}))$
}
```

We perform a case split on the value of e in σ_1:

- If $e \Downarrow_{\sigma_1} \bot$ then all traces from $\langle \texttt{while}\,(e)\ \{s\}, \sigma\rangle$ must progress to `skip` by Whl2 in one step, and therefore $n = 1$ and we are done.
- If $e \Downarrow_{\sigma_1} \top$ then we must have $\langle \texttt{while}\,(e)\ \{s\}, \sigma_1\rangle \longrightarrow \langle s; \texttt{while}\,(e)\ \{s\}, \sigma_1\rangle$ by Whl1. Additionally, we have $e^{(1)} \Downarrow_{\underline{\sigma}} \top$ by Lemma 3.4.1 and therefore $p^{(1)} \land e^{(1)} \lor p^{(2)} \land e^{(2)} \Downarrow_{\underline{\sigma}} \top$. Therefore all traces of the product must progress by Whl1. We perform another case split:
  - If $p^{(2)} \Downarrow_{\underline{\sigma}} \top \land e^{(2)} \Downarrow_{\underline{\sigma}} \top$ any trace of the product must continue by executing the following assignments of p_l and p_c as well as cond to create a new store

    $$\begin{aligned}\underline{\sigma}'' = \underline{\sigma}[{p_l}^{(1)} \mapsto \top][{p_l}^{(2)} \mapsto \top][{p_c}^{(1)} \mapsto \top]\\ [{p_c}^{(2)} \mapsto \top][\mathit{rank}^{(1)} \mapsto v_1][\mathit{rank}^{(2)} \mapsto v_2]\end{aligned}$$

    for some v_2 s.t. ${e_r}^{(2)} \Downarrow_{\underline{\sigma}} v_2$. By Lemma A.1.6 we still have $\sigma_1 \in_1 \underline{\sigma}''$ and $\sigma_2 \in_2 \underline{\sigma}''$. Since we still have that ${p_l}^{(1)} \Downarrow_{\underline{\sigma}''} \top$ and ${p_l}^{(2)} \Downarrow_{\underline{\sigma}''} \top$ and $\mathit{cond}^{(1)} \Downarrow_{\underline{\sigma}''} \top$ and $\mathit{cond}^{(2)} \Downarrow_{\underline{\sigma}''} \top$, we get that s always terminates from σ_1 and σ_2, and in particular, that there is some number n' s.t. s terminats from σ_1 in at most n' steps.

    By Thm. 3.4.5 and Lemma 3.4.6 and Lemma A.1.20 we get that for each pair of traces ending in some stores σ_1'' and σ_2'' of the original program, we get a terminating trace $\langle [\![\mathit{term}(s, \mathit{cond})]\!]_2^{\mathring{p}_l}, \underline{\sigma}''\rangle \longrightarrow^* \langle \dot{s}', \underline{\sigma}'''\rangle$ for some $\underline{\sigma}'''$. We cannot have that $\dot{s}' = \texttt{error}$ since $\underline{w}$ does not fail from $\underline{\sigma}$, can we have that $\dot{s}' = \texttt{magic}$ by Lemma 3.4.4 since s does not terminate with `magic` from either σ_1 or σ_2. We must therefore have that $\dot{s}' = \texttt{skip}$, and that $\sigma_1'' \in_1 \underline{\sigma}'''$ and $\sigma_2'' \in_2 \underline{\sigma}'''$ by Thm. 3.4.5. Since $\mathit{term}(s, \mathit{cond})$ does not assign to rank, we must still have that $\mathit{rank}^{(1)} \Downarrow_{\underline{\sigma}'''} v_1$. Since the following assertions do not fail from $\underline{\sigma}'''$ and $\mathit{cond}^{(1)}$ is also not modified by $\mathit{term}(s, \mathit{cond})$, we must have that ${e_r}^{(1)} \Downarrow_{\underline{\sigma}'''} v_1'$ for some v_1' s.t. $0 \leq v_1' < v_1$. The product must then step to $\langle \underline{w}, \underline{\sigma}'''\rangle$, and by the induction hypothesis, we get that there is some number n'' s.t. w always terminates from σ_1'' in at most n'' steps. Then we can conclude that w terminates from σ_1 in at most $2 + n' + n''$ steps (one step using Whl2 and one using Seq2).

- Otherwise the trace must step to a state with a store

$$\begin{aligned}\underline{\sigma}'' = \underline{\sigma}[{p_l}^{(1)} \mapsto \top][{p_l}^{(2)} \mapsto \bot]\\ [{p_c}^{(1)} \mapsto \top][{p_c}^{(2)} \mapsto \bot][\mathit{rank}^{(1)} \mapsto v_1]\end{aligned}$$

The argument from here is the same as before except that the second execution is inactive inside the loop.

□

We can conclude from this that if the product of the termination instrumentation of any statement is guaranteed not to fail from a product state with only one execution being active, the statement is guaranteed to terminate from a store that mirrors that of the active execution:

Lemma A.1.22 *Assume that* $\underline{s} = [\![\mathit{term}(s,c)]\!]_2^{\mathring{p}}$*,* s *is not and does not syntactically contain a method call,* $\underline{s}$ *never fails from* $\underline{\sigma}$*,* $p^{(a)} \Downarrow_{\underline{\sigma}} \top$ *and* $p^{(b)} \Downarrow_{\underline{\sigma}} \bot$*, where* $a, b \in \{1,2\}$*,* $\sigma_a \in_a \underline{\sigma}$*, and* s *never terminates with* `magic` *from* σ_a*. Then* s *always terminates from* σ_a*.*

Proof. By induction on the structure of s.

- Assignments, havocs, assumes and asserts always terminate in one step, so these cases are trivial.
- The case for method calls is impossible because s is not a method call.
- Case $s \equiv s_1; s_2$: Then we must have that $\underline{s} \equiv [\![\mathit{term}(s_1,c)]\!]_2^{\mathring{p}}; [\![\mathit{term}(s_2,c)]\!]_2^{\mathring{p}}$. By the induction hypothesis, s_1 always terminates from σ_a in at most n_1 steps (for some n_1). For any such trace $\langle s_1, \sigma_a\rangle \rightarrow^* \langle \dot{s}_a'', \sigma_a''\rangle$, we get by Thm. 3.4.5 and Lemma 3.4.6 and Lemma A.1.20 that $\dot{s}_a'' = \texttt{skip}$ and there is a trace $\langle [\![\mathit{term}(s_1,c)]\!]_2^{\mathring{p}}, \underline{\sigma}\rangle \rightarrow^* \langle \texttt{skip}, \underline{\sigma}''\rangle$ for some $\underline{\sigma}''$ s.t. $p^{(1)}$ and $p^{(2)}$ are unchanged and $\sigma_a'' \in_a \underline{\sigma}''$. By the same argument, s_2 always terminates from σ_a'', in some maximal number n_2 of steps. Therefore s terminates from σ_a in at most $1 + n_1 + n_2$ steps (one step resulting from Seq2).
- Case $s \equiv \texttt{if}\,(e)\,\{s_1\}\,\texttt{else}\,\{s_2\}$:
  Then we must have $\underline{s} \equiv {p_t}^{(1)}{:=}p^{(1)} \wedge e^{(1)}; {p_t}^{(2)}{:=}p^{(2)} \wedge e^{(2)}; {p_f}^{(1)}{:=}p^{(1)} \wedge \neg e^{(1)}; {p_f}^{(2)}{:=}p^{(2)} \wedge \neg e^{(2)}; [\![\mathit{term}(s_1,c)]\!]_2^{\mathring{p}_t}; [\![\mathit{term}(s_2,c)]\!]_2^{\mathring{p}_f}$.
  The product must then execute the assignments of the new activation variables, leading to a trace $\langle \underline{s}, \underline{\sigma}\rangle \rightarrow^* \langle [\![\mathit{term}(s_1,c)]\!]_2^{\mathring{p}_t}; [\![\mathit{term}(s_2,c)]\!]_2^{\mathring{p}_f}, \underline{\sigma}''\rangle$ for some $\underline{\sigma}''$.
  - If $e \Downarrow_{\sigma_a} \top$ then by Lemma 3.4.1 we have that ${p_t}^{(1)} \Downarrow_{\underline{\sigma}''} \top$ and ${p_t}^{(2)} \Downarrow_{\underline{\sigma}''} \bot$. We then get by the induction hypothesis that s_1 terminates from σ_a in some maximal number n_1 of steps, and s therefore terminates in at most $n_1 + 1$ steps (one step using Cond1).

- Otherwise we have that ${p_t}^{(1)} \Downarrow_{\underline{\sigma}''} \bot$ and ${p_t}^{(2)} \Downarrow_{\underline{\sigma}''} \bot$ and ${p_f}^{(1)} \Downarrow_{\underline{\sigma}''} \top$ and ${p_f}^{(2)} \Downarrow_{\underline{\sigma}''} \bot$ and therefore get from Lemma 3.4.3 that $\langle [\![term(s_1, c)]\!]_2^{\mathring{p}_1}, \underline{\sigma}'' \rangle \rightarrow^* \langle \texttt{skip}, \underline{\sigma}''' \rangle$ for some store $\underline{\sigma}'''$, s.t. we still have $\sigma_a \in_a \underline{\sigma}'''$. Since ${p_f}^{(1)}$ and ${p_f}^{(2)}$ are unchanged in $\underline{\sigma}'''$, we have the same situation as before; s_2 therefore always terminates from σ_a in some number n_2 of steps by the induction hypothesis, and therefore s terminates in at most $n_2 + 1$ steps from σ_a (using one step for Cond2).

▸ Case $s \equiv \texttt{while}\,(e)\ \texttt{terminates}(e_c, e_r)\ \{s_l\}$: Then we must have

$$
\begin{array}{ll}
\underline{s} \equiv & \texttt{if}\,(p^{(1)})\ \{cond^{(1)}{:=}{e_c}^{(1)}\} \\
& \texttt{if}\,(p^{(2)})\ \{cond^{(2)}{:=}{e_c}^{(2)}\} \\
& \texttt{assert}\ \neg((p^{(1)} \land \neg cond^{(1)}) \lor (p^{(2)} \land \neg cond^{(2)})) \lor p^{(1)} = p^{(2)} \\
& \texttt{assert}\ \neg(p^{(1)} \land p^{(2)}) \lor cond^{(1)} = cond^{(2)} \\
& \texttt{assert}\ (\neg(p^{(1)} \land cond^{(1)}) \lor ({e_r}^{(1)} \geq 0)) \land \\
& \qquad\quad (\neg(p^{(2)} \land cond^{(2)}) \lor ({e_r}^{(2)} \geq 0)) \\
& \texttt{assert}\ (\neg(p^{(1)} \land c^{(1)}) \lor cond^{(1)}) \land (\neg(p^{(2)} \land c^{(2)}) \lor cond^{(2)}) \\
& \texttt{assert}\ (\neg(p^{(1)} \land \neg cond^{(1)}) \lor e^{(1)}) \land (\neg(p^{(2)} \land \neg cond^{(2)}) \lor e^{(2)}) \\
& \underline{w}
\end{array}
$$

where $\underline{w}$ has the same form as in the proof of Lemma A.1.21.

By the induction hypothesis, we have that if $[\![term(s_l, cond)]\!]_2^{\mathring{p}_l}$ does not fail from some store $\underline{\sigma}'''$ in which only one execution a is active, s_l always terminates from a store σ_a''' s.t. $\sigma_a''' \in_a \underline{\sigma}'''$. After executing the assignments, $\underline{s}$ executes assertions that fail if ${e_r}^{(a)}$ is negative or if ${e_r}^{(a)}$ is not true in $\underline{\sigma}$. Since they cannot fail from $\underline{\sigma}$, we have the conditions required to apply Lemma A.1.21 and we are done.

□

Similarly, we can conclude that if the product of the termination instrumentation $term(s, c)$ of a statement s is guaranteed not to fail from a product state in which c is true for both executions, the statement is guaranteed to terminate from any pair of stores that mirror that of the product:

Lemma A.1.23 *Assume that $\underline{s} = [\![term(s, c)]\!]_2^{\mathring{p}}$, s is not and does not syntactically contain a method call, $\underline{s}$ never fails from $\underline{\sigma}$, $p^{(2)} \Downarrow_{\underline{\sigma}} \top$ and $p^{(2)} \Downarrow_{\underline{\sigma}} \top$, $c^{(2)} \Downarrow_{\underline{\sigma}} \top$ and $c^{(2)} \Downarrow_{\underline{\sigma}} \top$, $\sigma_1 \in_1 \underline{\sigma}$ and $\sigma_2 \in_2 \underline{\sigma}$, and s never terminates with* `magic` *from σ_1 or σ_2. Then s always terminates from σ_1 and σ_2*

Proof. By induction on the structure of s. The cases for basic statements are again trivial and the case for method calls is impossible. The case for sequential composition follows directly from using the induction hypothesis on the substatements. For conditionals, we use Lemma A.1.22 in case the executions take different branches, and otherwise use the induction hypothesis on the branch that is executed and Lemma 3.4.3. For loops, similar to the previous proof, the assertion at the beginning of the product must fail if its termination e_c is not true for both executions, which cannot happen. We therefore have the conditions to apply Lemma A.1.21 and we are done. □

Next, we prove that in the conditions ensured by the initial assertions of our loop encoding, if the termination condition is false, executions of the actual program never terminate:

Lemma A.1.24 *Assume that*

```
w  =  while (e)
        do {
          if (cond) {rank:=e_r};
          term(s, cond);
          assert cond ⇒ 0 ≤ e_r ∧ e_r < rank
          assert ¬cond ⇒ e
        }
```

and $\underline{w} = [\![w]\!]_2^{\mathring{p}}$, $\underline{w}$ *never fails from* $\underline{\sigma}$, $p^{(1)} \Downarrow_{\underline{\sigma}} \top$ *and* $\sigma_1 \in_1 \underline{\sigma}$ *and* $p^{(2)} \Downarrow_{\underline{\sigma}} \top$ *and* $\sigma_2 \in_2 \underline{\sigma}$, $cond^{(1)} \Downarrow_{\underline{\sigma}} \bot$ *and* $cond^{(2)} \Downarrow_{\underline{\sigma}} \bot$, $e^{(1)} \Downarrow_{\underline{\sigma}} \top$ *and* $e^{(2)} \Downarrow_{\underline{\sigma}} \top$, s *never terminates with* `magic` *from* σ_1 *and* σ_2, $term(s, cond)$ s *does not assign to cond, and for all* $\underline{\sigma}'', \sigma_1'', \sigma_2''$, *if* $p^{(1)} \Downarrow_{\underline{\sigma}''} \top$ *and* $p^{(2)} \Downarrow_{\underline{\sigma}''} \top$ *and* $\sigma_1'' \in_1 \underline{\sigma}''$ *and* $\sigma_2'' \in_2 \underline{\sigma}''$ *and* $[\![term(s, cond)]\!]_2^{\mathring{p}}$ *does not fail from* $\underline{\sigma}''$, *then the termination of* s *from* σ_1'' *and* σ_2'' *coincides.*

Then

1. *if* $\langle \texttt{while}\,(e)\,\{s\}, \sigma_1\rangle \rightarrow^{k_1} \langle \hat{s}_1', \sigma_1'\rangle$ *for some* $\hat{s}_1', \sigma_1'$ *then* $\hat{s}_1'$ *is not final.*
2. *if* $\langle \texttt{while}\,(e)\,\{s\}, \sigma_2\rangle \rightarrow^{k_2} \langle \hat{s}_2', \sigma_2'\rangle$ *for some* $\hat{s}_2', \sigma_2'$ *then* $\hat{s}_2'$ *is not final.*

Proof. We prove both conclusions separately. We show the proof for (1), the one for (2) is analogous. We take an arbitrary trace $\langle \texttt{while}\,(e)\,\{s\}, \sigma_1\rangle \rightarrow^{k_1} \langle s_1', \sigma_1'\rangle$. The proof goes by strong induction on k_1. $\underline{w}$ must have the same form as described in the proof for Lemma A.1.21. If $k_1 = 0$ we are done. Since $e^{(1)} \Downarrow_{\underline{\sigma}} \top$ and $e^{(2)} \Downarrow_{\underline{\sigma}} \top$ we have that $e \Downarrow_{\sigma_1} \top$ and $e \Downarrow_{\sigma_2} \top$ by Lemma 3.4.1. Therefore by Whl1 we have that $\langle \texttt{while}\,(e)\,\{s\}, \sigma_1\rangle \rightarrow \langle s;\texttt{while}\,(e)\,\{s\}, \sigma_1\rangle$ and $\langle \texttt{while}\,(e)\,\{s\}, \sigma_2\rangle \rightarrow \langle s;\texttt{while}\,(e)\,\{s\}, \sigma_2\rangle$ and we are done if $k_1 = 1$. Now either s terminates from both σ_1 and σ_2 or from none of them. If it does not, we are done, since no final state will be reached from here. We consider the case where both terminate with some new stores σ_1'', σ_2'' and assume the first execution terminates from σ_1 in k_1' steps. If $k_1 < k_1' + 1$ we are again done because none of the intermediate configurations are final. After executing the assignments of new activation variables as well as the *rank* variable the product must reach a configuration with a new store $\underline{\sigma}''$ as in the proof for Lemma A.1.21. By Thm. 3.4.5 and Lemma 3.4.6 we get that $[\![s]\!]_2^{\mathring{p}_l}$ terminates from $\underline{\sigma}$ in some $\underline{\sigma}''$. By Lemma A.1.20 we get that the instrumented product terminates with some $\underline{\sigma}'''$ s.t. $\underline{\sigma}''' \precsim^{\underline{w}} \underline{\sigma}''$. Since *cond* is unchanged in $\underline{\sigma}'''$, the product must subsequently assert that $e^{(1)} \Downarrow_{\underline{\sigma}'''} \top$ and $e^{(2)} \Downarrow_{\underline{\sigma}'''} \top$. Since the assertion cannot fail and both activation variables are still active, we must have that $p^{(1)} \wedge e^{(1)} \wedge p^{(2)} \wedge e^{(2)} \Downarrow_{\underline{\sigma}'''} \top$. The product must then step to a configuration $\langle \underline{w}, \underline{\sigma}'''\rangle$, and similarly the original executions step to $\langle w, \sigma_1''\rangle$ and $\langle w, \sigma_2''\rangle$, respectively, by Seq2. By the induction hypothesis, s never terminates from σ_1'', and we are done. □

Finally, we can prove our main theorem:

Theorem 3.5.2 *If $\underline{s} = \llbracket term(s, false) \rrbracket_2^{\mathring{p}}$ and $\vDash \{\lfloor \check{P} \rfloor_2^{\mathring{p}}\}\underline{s}\{true\}$ and $(\sigma_1, \sigma_2) \vDash \check{P}$ and s does not terminate with `magic` from σ_1 and σ_2 and s does not contain calls to (mutually) recursive methods then the termination of s from σ_1 and σ_2 coincides.*

Proof. Since s does not contain calls to (mutually) recursive methods, any calls in it can be inlined without changing the program semantics. We assume that this has happened, and s therefore does not contain any method calls whatsoever. We pick $\underline{\sigma}$ s.t. $p^{(1)} \Downarrow_{\underline{\sigma}} \top$ and $p^{(2)} \Downarrow_{\underline{\sigma}} \top$ and $\sigma_1 \in_1 \underline{\sigma}$ and $\sigma_2 \in_2 \underline{\sigma}$. Because $\vDash \{\lfloor \check{P} \rfloor_2^{\mathring{p}}\}\underline{s}\{true\}$ by Lemma 3.4.8 we know that $\underline{s}$ does not fail from $\underline{\sigma}$.

We have to prove that for both combinations of i, j s.t. $\{i, j\} = \{1, 2\}$, if $\langle s, \sigma_i \rangle \longrightarrow^* \langle \dot{s}, \sigma_i' \rangle$ for some $\dot{s}, \sigma_i'$ then s also always terminates from σ_j. We now prove the following statement by induction on the structure of s:

If $s' = term(s, c)$, $\underline{s} = \llbracket s' \rrbracket_2^{\mathring{p}}$, $\underline{s}$ never fails from $\underline{\sigma}$, $p^{(1)} \Downarrow_{\underline{\sigma}} \top$ and $p^{(2)} \Downarrow_{\underline{\sigma}} \top$, $\sigma_1 \in_1 \underline{\sigma}$, $\sigma_2 \in_2 \underline{\sigma}$, s never terminates with `magic` from σ_1 and σ_2, and s does not contain method calls.

Then if $\langle s, \sigma_i \rangle \longrightarrow^* \langle \dot{s}_i, \sigma_i' \rangle$ for some $\dot{s}_i, \sigma_i'$ then s always terminates from σ_j.

We assume that $\langle s, \sigma_i \rangle \longrightarrow^* \langle \dot{s}_i, \sigma_i' \rangle$ (otherwise we are done) and need to show that s always terminates from σ_j.

- The cases for `skip`, assignments, `havoc`, `assume`, and `assert` are trivial since those statements always terminate.
- The case for method calls is impossible.
- Case $s \equiv s_1; s_2$: Then $\underline{s} \equiv \llbracket term(s_1, c) \rrbracket_2^{\mathring{p}}; \llbracket term(s_2, c) \rrbracket_2^{\mathring{p}}$. By Lemma A.1.7.2 and Lemma A.1.7.3 we get that $\langle s_1, \sigma_i \rangle \longrightarrow^* \langle \dot{s}_i, \sigma_i'' \rangle$ for some σ_i'' and $\dot{s}_i \neq$ `magic`. By Lemma A.1.7.3 we also get that $\llbracket term(s_1, c) \rrbracket_2^{\mathring{p}}$ does not fail from $\underline{\sigma}$. By the induction hypothesis, we then get that s_1 always terminates from σ_j in some configuration $\langle \dot{s}_j, \sigma_j'' \rangle$. By Lemma 3.4.6 and Lemma A.1.20 we then get that $\dot{s}_i =$ `skip`, and by Thm. 3.4.5 we have that for any such σ_i'' and σ_j'' there is some $\underline{\sigma}''$ s.t. $\sigma_i'' \in_i \underline{\sigma}''$ and $\sigma_j'' \in_j \underline{\sigma}''$ and the activation variables are still true. We can then apply the induction hypothesis again to s_2 and we are done.
- Case $s \equiv$ `if` $(e)\,\{s_1\}$ `else` $\{s_2\}$: Then $\underline{s} \equiv {p_t}^{(1)} := p^{(1)} \wedge e^{(1)}; {p_t}^{(2)} := p^{(2)} \wedge e^{(2)}; {p_f}^{(1)} := p^{(1)} \wedge \neg e^{(1)}; {p_f}^{(2)} := p^{(2)} \wedge \neg e^{(2)}; \llbracket term(s_1, c) \rrbracket_2^{\mathring{p}_t}; \llbracket term(s_2, c) \rrbracket_2^{\mathring{p}_f}$. The product begins by assigning the new activation variables, resulting in a store $\underline{\sigma}''$ modified in the obvious way. We perform a case split on the value of the condition expression in both executions:
  - If $e^{(i)} \Downarrow_{\underline{\sigma}} \top$ and $e^{(j)} \Downarrow_{\underline{\sigma}} \top$ then also $e \Downarrow_{\sigma_i} \top$ and $e \Downarrow_{\sigma_j} \top$ by Lemma 3.4.1 and both executions must proceed by Cond1. We then have that $\langle s_1, \sigma_i \rangle \longrightarrow^* \langle \dot{s}_i, \sigma_i' \rangle$ for some σ_i' and that $\llbracket term(s_1, c) \rrbracket_2^{\mathring{p}_t}$ does not fail from $\underline{\sigma}''$. Since ${p_t}^{(i)} \Downarrow_{\underline{\sigma}''} \top$ and ${p_t}^{(j)} \Downarrow_{\underline{\sigma}''} \top$, we get that s_1 always terminates from σ_j by the induction hypothesis.

- If $e^{(i)} \Downarrow_{\underline{\sigma}} \bot$ and $e^{(j)} \Downarrow_{\underline{\sigma}} \bot$ we have that ${p_t}^{(i)} \Downarrow_{\underline{\sigma}''} \bot$ and ${p_t}^{(i)} \Downarrow_{\underline{\sigma}''} \bot$. We use Lemma 3.4.3 to get that $\langle [\![\mathit{term}(s_1, c)]\!]_2^{\mathring{p}_t}, \underline{\sigma}'' \rangle \rightarrow^* \langle \texttt{skip}, \underline{\sigma}''' \rangle$ for some store $\underline{\sigma}'''$ that still mirrors the stores of both single executions. Then the argument goes as in the previous case.
- If $e^{(i)} \Downarrow_{\underline{\sigma}} \top$ and $e^{(j)} \Downarrow_{\underline{\sigma}} \bot$ we have that ${p_t}^{(i)} \Downarrow_{\underline{\sigma}''} \top$ and ${p_t}^{(j)} \Downarrow_{\underline{\sigma}''} \bot$ and ${p_f}^{(i)} \Downarrow_{\underline{\sigma}''} \bot$ and ${p_f}^{(j)} \Downarrow_{\underline{\sigma}''} \top$. By Lemma A.1.22 we get that s_1 always terminates from σ_i. By Thm. 3.4.5 and Lemma 3.4.6 and Lemma A.1.20 we get for any trace of s_1 from σ_i that $[\![\mathit{term}(s_1, c)]\!]_2^{\mathring{p}_t}$ terminates from $\underline{\sigma}''$ in some $\underline{\sigma}'''$ with ${p_f}^{(1)}$ and ${p_f}^{(2)}$ unchanged and $\sigma_j \in_j \underline{\sigma}'''$. Then we can apply Lemma A.1.22 again to get that s_2 always terminates from σ_j and we are done.
- If $e^{(i)} \Downarrow_{\underline{\sigma}} \bot$ and $e^{(j)} \Downarrow_{\underline{\sigma}} \top$ the argument goes as in the previous case.

► Case $s \equiv \texttt{while}\ (e)\ \texttt{terminates}(e_c, e_r)\ \{s_l\}$: Then

$$\begin{aligned}
\underline{s} \equiv\ \ & \texttt{if}\ (p^{(1)})\ \{\mathit{cond}^{(1)}{:=}{e_c}^{(1)}\} \\
& \texttt{if}\ (p^{(2)})\ \{\mathit{cond}^{(2)}{:=}{e_c}^{(2)}\} \\
& \texttt{assert}\ \neg((p^{(1)} \wedge \neg \mathit{cond}^{(1)}) \vee (p^{(2)} \wedge \neg \mathit{cond}^{(2)})) \vee p^{(1)} = p^{(2)} \\
& \texttt{assert}\ \neg(p^{(1)} \wedge p^{(2)}) \vee \mathit{cond}^{(1)} = \mathit{cond}^{(2)} \\
& \texttt{assert}\ (\neg(p^{(1)} \wedge \mathit{cond}^{(1)}) \vee ({e_r}^{(1)} \geq 0)) \wedge (\neg(p^{(2)} \wedge \mathit{cond}^{(2)}) \vee ({e_r}^{(2)} \geq 0)) \\
& \texttt{assert}\ (\neg(p^{(1)} \wedge c^{(1)}) \vee \mathit{cond}^{(1)}) \wedge (\neg(p^{(2)} \wedge c^{(2)}) \vee \mathit{cond}^{(2)}) \\
& \texttt{assert}\ (\neg(p^{(1)} \wedge \neg \mathit{cond}^{(1)}) \vee e^{(1)}) \wedge (\neg(p^{(2)} \wedge \neg \mathit{cond}^{(2)}) \vee e^{(2)}) \\
& \underline{w}
\end{aligned}$$

where $\underline{w}$ has the same form as in the proof of Lemma A.1.21. We case split on the value of e_c for both executions:

- If ${e_c}^{(i)} \Downarrow_{\underline{\sigma}} \top$ and ${e_c}^{(j)} \Downarrow_{\underline{\sigma}} \top$ then $\mathit{cond}^{(i)}$ and $\mathit{cond}^{(j)}$ both become true in the product store. By Lemma A.1.23 s_l always terminates from any stores σ_i'', σ_j'' if there is a product store $\underline{\sigma}''$ s.t. $\sigma_i'' \in_i \underline{\sigma}''$ and $\sigma_j'' \in_j \underline{\sigma}''$ and $\mathit{cond}^{(1)} \Downarrow_{\underline{\sigma}''} \top$ and $\mathit{cond}^{(2)} \Downarrow_{\underline{\sigma}''} \top$ and from which $[\![\mathit{term}(s_l, \mathit{cond})]\!]_2^{\mathring{p}_l}$ never fails. We can therefore use Lemma A.1.21 and we are done.
- If ${e_c}^{(i)} \Downarrow_{\underline{\sigma}} \bot$ and ${e_c}^{(j)} \Downarrow_{\underline{\sigma}} \bot$ then $\mathit{cond}^{(i)}$ and $\mathit{cond}^{(j)}$ both become false in the product store. We must therefore have that $e^{(1)} \Downarrow_{\underline{\sigma}} \top$ and $e^{(2)} \Downarrow_{\underline{\sigma}} \top$, since otherwise the last assertion will fail. By the induction hypothesis, the termination of s_l coincides from any stores σ_i'', σ_j'' if there is a product store $\underline{\sigma}''$ s.t. $\sigma_i'' \in_i \underline{\sigma}''$ and $\sigma_j'' \in_j \underline{\sigma}''$ and $\mathit{cond}^{(1)} \Downarrow_{\underline{\sigma}''} \bot$ and $\mathit{cond}^{(2)} \Downarrow_{\underline{\sigma}''} \bot$ and from which $[\![\mathit{term}(s_l, \mathit{cond})]\!]_2^{\mathring{p}_l}$ never fails. Therefore by Lemma A.1.24 s never terminates from σ_1 and this case is impossible.
- If ${e_c}^{(i)} \Downarrow_{\underline{\sigma}} \top$ and ${e_c}^{(j)} \Downarrow_{\underline{\sigma}} \bot$ we must get $\mathit{cond}^{(i)} \Downarrow_{\underline{\sigma}''} \top$ and $\mathit{cond}^{(j)} \Downarrow_{\underline{\sigma}''} \bot$ in the product store after the initial assignments. Then the assertion $\neg(p^{(1)} \wedge p^{(2)}) \vee \mathit{cond}^{(1)} = \mathit{cond}^{(2)}$ must fail during the product execution, which cannot happen, so this case is impossible.
- If ${e_c}^{(i)} \Downarrow_{\underline{\sigma}} \bot$ and ${e_c}^{(j)} \Downarrow_{\underline{\sigma}} \bot$ then the same assertion will fail, so this case is also impossible.

□

Additional Benchmarks for Chapter 4 B.

B.1. Benchmarks from Functional Test Suite

Name	T	T_{SIF}	Factor
empty.py	3.42	3.50	1.02
examples/cav_example.py	7.64	90.46	11.83
examples/iap_bst.py	6.49	83.72	12.90
examples/parkinson_recell.py	3.74	4.05	1.09
examples/rosetta_qsort.py	6.94	120.87	17.41
examples/test_student_enroll_preds.py	5.74	34.40	5.99
issues/00001.py	3.49	3.50	1.00
issues/00003.py	3.74	4.69	1.25
issues/00005.py	3.92	3.67	0.94
issues/00007.py	3.67	3.80	1.04
issues/00008.py	3.67	3.50	0.95
issues/00010.py	3.60	3.67	1.02
issues/00015.py	3.82	4.00	1.05
issues/00016.py	3.42	3.47	1.01
issues/00017.py	4.92	5.27	1.07
issues/00019.py	4.35	4.62	1.06
issues/00021.py	3.50	3.54	1.01
issues/00022.py	3.55	3.54	1.00
issues/00023_2.py	3.92	3.89	0.99
issues/00023.py	3.67	3.80	1.04
issues/00027.py	3.52	3.69	1.05
issues/00031.py	4.42	4.94	1.12
issues/00032.py	3.45	3.60	1.04
issues/00033.py	3.57	3.65	1.02
issues/00041.py	3.57	3.52	0.99
issues/00043.py	3.49	3.47	1.00
issues/00044.py	3.60	3.64	1.01
issues/00045.py	3.65	3.60	0.99
issues/00046.py	4.35	4.27	0.98
issues/00047.py	4.49	4.61	1.03
issues/00048.py	4.44	7.41	1.67
issues/00049.py	3.79	4.56	1.20
issues/00053.py	3.69	3.77	1.02
issues/00054.py	3.59	3.60	1.00
issues/00055.py	3.60	3.54	0.98
issues/00056.py	4.29	4.32	1.01
issues/00057.py	4.39	4.27	0.97
issues/00059.py	6.02	57.77	9.59
issues/00071.py	3.42	3.59	1.05
issues/00112.py	3.60	3.59	1.00
issues/00113.py	3.87	3.85	1.00
issues/00115.py	4.45	5.89	1.32
issues/00118.py	3.89	4.47	1.15
issues/00120.py	4.10	4.50	1.10
issues/00141.py	4.02	5.87	1.46

test_adt_1.py	4.56	5.46	1.20
test_adt_2.py	3.92	3.92	1.00
test_adt_3.py	4.29	5.01	1.17
test_assign.py	6.19	11.38	1.84
test_behavioural_subtyping_classmethod.py	3.82	4.37	1.14
test_behavioural_subtyping_predicates.py	3.95	4.02	1.02
test_behavioural_subtyping_static.py	3.82	3.92	1.03
test_behavioural_subtyping.py	5.59	11.15	1.99
test_boolop.py	4.64	5.77	1.24
test_boxing.py	4.89	6.63	1.36
test_builtin_functions.py	3.89	4.10	1.06
test_builtin_globals_1.py	3.40	3.55	1.04
test_builtin_globals_2.py	3.55	3.89	1.09
test_bytes.py	4.35	4.82	1.11
test_cast.py	3.82	4.34	1.14
test_classmethod.py	4.02	4.34	1.08
test_constructor.py	3.60	4.05	1.13
test_conversion.py	4.96	6.42	1.30
test_default_return.py	5.12	22.01	4.30
test_definedness.py	3.64	3.72	1.02
test_dict_pred.py	5.09	7.26	1.43
test_dicts.py	4.44	6.17	1.39
test_dynamic_field_creation.py	4.05	4.49	1.11
test_enumerate.py	5.44	13.94	2.56
test_exception_loop.py	4.02	6.54	1.63
test_exception.py	5.46	8.33	1.53
test_exists.py	4.54	4.69	1.03
test_field_deletion.py	3.80	3.87	1.02
test_fields.py	4.02	5.04	1.25
test_float.py	3.92	3.97	1.01
test_forall.py	5.32	6.98	1.31
test_funcs_and_methods.py	4.59	5.21	1.13
test_function_basics.py	4.20	4.29	1.02
test_generic_classes.py	4.72	6.49	1.37
test_generic_methods.py	4.35	5.27	1.21
test_global_definedness_1.py	3.70	3.72	1.00
test_global_definedness_10.py	3.50	3.59	1.02
test_global_definedness_11.py	3.70	3.67	0.99
test_global_definedness_2.py	3.59	3.65	1.02
test_global_definedness_3.py	3.47	3.65	1.05
test_global_definedness_4.py	3.74	4.24	1.13
test_global_definedness_5.py	3.64	3.79	1.04
test_global_definedness_6.py	3.84	4.45	1.16
test_global_definedness_7.py	3.65	3.65	1.00
test_global_definedness_8.py	3.49	3.57	1.02
test_global_definedness_9.py	3.54	3.64	1.03
test_global_mutable_1.py	3.70	4.09	1.10
test_global_mutable_2.py	3.64	3.84	1.06
test_global_program.py	4.16	5.79	1.39
test_global_scopes.py	4.07	4.24	1.04
test_global_stateful.py	3.84	4.12	1.07
test_global_vars.py	3.75	3.79	1.01
test_havoced_types.py	5.14	18.33	3.57

test_identity.py	4.09	4.14	1.01
test_import_execution.py	4.97	8.73	1.76
test_imports_2.py	3.55	3.79	1.07
test_imports_cyclic.py	3.77	4.02	1.07
test_imports.py	3.82	4.12	1.08
test_isinstance.py	3.39	3.87	1.14
test_iterator_dict.py	6.57	26.99	4.11
test_iterator_list.py	8.76	65.56	7.48
test_iterator_set.py	6.39	30.10	4.71
test_let.py	3.89	4.04	1.04
test_list_comprehension.py	4.91	7.28	1.48
test_list_pred.py	4.45	5.39	1.21
test_lists.py	5.39	6.74	1.25
test_loop_break.py	5.32	26.51	4.98
test_loop_continue.py	4.30	7.86	1.83
test_magic_methods.py	3.95	4.07	1.03
test_method_calls.py	4.59	5.47	1.19
test_named_args.py	4.09	4.77	1.17
test_namespace.py	4.27	5.31	1.24
test_operator_overloading.py	4.05	4.55	1.12
test_operators.py	4.09	6.22	1.52
test_optional_types.py	4.37	5.14	1.18
test_pmultiset.py	4.00	4.24	1.06
test_predicate_families.py	4.07	4.61	1.13
test_predicate.py	3.72	4.09	1.10
test_property.py	4.07	4.51	1.11
test_pseq.py	4.37	5.16	1.18
test_pset.py	3.99	3.92	0.98
test_raised_exception.py	3.74	4.72	1.26
test_range.py	5.29	10.45	1.98
test_set_pred.py	4.50	4.96	1.10
test_set.py	3.97	4.64	1.17
test_slicing.py	5.41	21.05	3.89
test_starred.py	3.99	4.72	1.18
test_static_calls.py	4.39	6.47	1.48
test_static_class_members.py	4.07	4.40	1.08
test_string.py	3.59	3.65	1.02
test_super.py	3.69	3.90	1.06
test_tuples.py	4.09	4.64	1.13
test_type_aliases.py	3.79	4.17	1.10
test_union_contracts.py	5.24	10.42	1.99
test_union_types.py	4.12	4.57	1.11
test_varargs.py	4.46	5.37	1.21
test_while.py	4.05	5.66	1.40
test_wildcard_permissions.py	4.00	4.64	1.16
test_with.py	4.67	5.86	1.25

Table B.1.: Evaluated tests from Nagini's functional test suite. For all tests, the same errors are reported with and without enabling the information flow option. T and T_{SIF} represent the average verification time without and with the information flow, respectively, in seconds. Factor describes the slowdown factor introduced by enabling the information flow option.