DISS. ETH NO. 22868

Partial Verification Results

A thesis submitted to attain the degree of

Doctor of Sciences of ETH Zurich

(Dr. sc. ETH Zurich)

presented by

Valentin Tobias Wüstholz

MSc ETH CS, ETH Zurich

born on 22.09.1985

citizen of

Germany

accepted on the recommendation of

Prof. Dr. Peter Müller, examiner
Prof. Dr. Işil Dillig, co-examiner
Dr. Francesco Logozzo, co-examiner
Prof. Dr. Bertrand Meyer, co-examiner

2015

Abstract

In recent years, program analysis tools have been increasingly applied to real-world software to prevent defects as early as possible. Examples of such tools include both static analyzers and automatic test case generation tools. While the latter traditionally under-approximate the possible program executions to find errors, the former traditionally consider additional program executions that are not actually possible in the analyzed program. This makes it possible to efficiently analyze programs with a large or infinite number of program executions and to prove the correctness of the analyzed program in case the analysis *over-approximates* the possible program executions (i.e., is sound). As a consequence, static analyzers may report *spurious* errors that do not reveal real defects in the analyzed program.

In practice, many static analyzer neither under- nor over-approximate the program executions of the analyzed program. Their *designers* trade soundness for other qualities—such as precision, performance, and automation—by deliberately ignoring certain checks (e.g., that a method respects its write effect specification) or by deliberately making assumptions that do not hold for all program executions (e.g., that no arithmetic overflow occurs).

These characteristics of static analyzers motivate why verification results are often *partial* in practice: (1) some assertions of a program have neither been verified nor have been shown to lead to a defect (i.e., may be spurious errors) and (2) some program executions—including ones that may result in defects—have been ignored due to sources of deliberate unsoundness in the analysis.

To express and share such results with the user or other program analysis tools, we have developed a technique for annotating programs with partial verification results using two new language constructs. For instance, these allow us to express if a property could not definitely be verified and if a property has only been verified under assumptions that might not always hold. We describe several novel use cases for expressing partial verification results from diverse areas of program analysis—such as test case generation, specification inference, counterexample-based error reporting, and static analysis.

In particular, we present an architecture for combining static analyzers with test case generation tools by exchanging programs that have been annotated with partial verification results between tools. By soundly expressing what has already been verified, tools can benefit from the results of other tools and reduce their verification effort.

To evaluate deliberate unsoundness in a practical static analyzer, we identified and documented all sources of deliberate unsoundness in the .NET static analyzer Clousot. Based on this, we developed a wrapper for Clousot that uses our technique for annotating a program using partial verification results. By expressing most sources of unsoundness explicitly and developing a suitable runtime instrumentation, we evaluated whether Clousot's unsound assumptions are violated in practice and whether such violations cause Clousot to miss bugs. Such findings can guide users of static analyzers in using them fruitfully, and help their designers in striking a good balance between soundness and other qualities of an analyzer, such as precision, performance, and automation.

Nowadays, partial verification results are often shown to users within an integrated development environment (IDE). We present the IDE for Dafny—a programming language, verifier, and proof assistant—that addresses two issues present in most state-of-the-art IDEs for program verifiers: low responsiveness and lack of support for understanding non-obvious verification failures. To this end, we present both new techniques and integrate existing technique to improve the user experience. This allows the IDE to provide verification feedback as the user types and to present more helpful information about the program or failed verification attempts in a demand-driven and unobtrusive way. As a result, the user is able to quickly gain insights about the program and the cause of partial verification results.

To increase the responsiveness of the program verifier during such interactions with the user, we designed a system for fine-grained caching of verification results. The caching system uses the program's call graph and control-flow graph to focus the verification effort on just the parts of the program that were affected by the user's most recent modifications. The novelty lies in how the original program is instrumented with partial verification results from the cache to avoid unnecessary work for the verifier. By using our technique for expressing partial verification results, we are able to reuse some cached verification results even if assumptions in the program (e.g., due to modular reasoning about calls by means of the callee's postcondition) are affected by a change.

Zusammenfassung

In den letzten Jahren wurden Werkzeuge zur Programmanalyse mehr und mehr dazu eingesetzt, um Defekte in praxisnaher Software so früh wie möglich zu verhindern. Beispiele für solche Werkzeuge sind Werkzeuge zur statischen Analyse und zur automatischen Testfall-Generierung. Während Letztere traditionell die möglichen Programmausführungen unterapproximieren um Fehler zu finden, betrachten Erstere weitere Programmausführungen, die nicht tatsächlich im analysierten Programm auftreten. Dies ermöglicht es, Programme mit einer grossen oder unendlichen Zahl von Programmausführungen effizient zu analysieren und die Korrektheit des analysierten Programms zu beweisen, falls die Analyse die möglichen Programmausführungen *überapproximiert* (d.h. sound ist). Folglich kann es dazu kommen, dass Werkzeuge zur statischen Analyse *unechte* Fehler melden, die gar nicht zu echten Defekten im analysierten Programm führen.

In der Praxis betrachten viele Werkzeuge zur statischen Analyse weder eine Unter- noch eine Überapproximation der möglichen Programmausführungen. Ihre *Designer* wägen Soundness gegen andere Qualitäten—wie etwa Präzision, Effizienz, und Automatisierung—ab, indem sie absichtlich gewisse Überprüfungen auslassen (zum Beispiel, dass eine Methode ihre Write-Effect-Spezifikation erfüllt) oder absichtlich Annahmen treffen, die nicht in allen Programmausführungen zutreffen (zum Beispiel, dass kein arithmetischer Overflow eintritt).

Diese Charakteristika von Werkzeugen zur statischen Analyse begründen, weshalb Verifikationsresulte in der Praxis oft *partiell* sind: (1) einige Assertions des Programms wurden weder verifiziert noch wurde gezeigt, dass sie zu einem Defekt führen (d.h. sie könnten unechte Fehler sein) und (2) einige Programmausführungen—inklusive solcher, die zu Defekten führen könnten—wurden ignoriert aufgrund von absichtlicher Unsoundness in der Analyse.

Um solche Resultat auszudrücken und mit dem Benutzer oder anderen Werkzeugen zur Programmanalyse zu teilen, haben wir eine Technik entwickelt, die es erlaubt, mittels zweier neuer Sprachkonstrukte das Programm mit partiellen Verifikationsresulten zu annotieren. Beispielsweise lässt sich damit ausdrücken, dass eine Eigenschaft nicht mit Sicherheit verifiziert werden konnte und dass eine Eigenschaft lediglich unter Annahmen verifiziert wurde, die nicht zwangsläufig zutreffen müssen. Wir beschreiben mehrere Fälle aus verschiedenen Gebieten der Programmanalyse—wie zum Beispiel Testfall-Generierung, Spezifikationsinferenz, Gegenbeispiel-basierte Fehlermeldeverfahren und statische Analyse—in denen sich partielle Verifikationsresulte ausdrücken lassen.

Im Speziellen stellen wir eine Architektur vor, um Werkzeuge zur statischen Analyse mit solchen zur Testfall-Generierung zu kombinieren, indem Programme ausgetauscht werden, die mit partiellen Verifikationsresulten annotiert wurden. Indem sound ausgedrückt wird, was bereits verifiziert wurde, können Werkzeuge von den Resultaten anderer Werkzeuge profitieren und ihren Verifikationsaufwand senken.

Um absichtliche Unsoundness in einem praktischen Werkzeug für statische Analyse auszuwerten, haben wir alle Quellen von absichtlicher Unsoundness in Clousot, einem Werkzeug zur statische Analyse für .NET, identifiziert und dokumentiert. Basierend darauf haben wir einen Wrapper für Clousot entwickelt, der unsere Technik zur Annotation von Programmen mit partiellen Verifikationsresulten einsetzt. Indem die Mehrzahl aller Quellen von absichtlicher Unsoundness explizit gemacht werden und durch die Entwicklung einer geeigneten Laufzeitinstrumentierung, haben wir untersucht, ob Clousots unsounde Annahmen in der Praxis verletzt werden und ob solche Verletzungen dazu führen, dass Clousot Fehler übersieht. Solche Erkenntnisse können Designern von Werkzeugen zur statischen Analyse sowohl dabei helfen, diese nutzbringend einzusetzen, als auch eine gute Balance zwischen Soundness und anderen Qualitäten solcher Werkzeuge, wie beispielsweise Präzision, Effizienz und Automatisierung, zu finden.

Heutzutage werden partielle Verifikationsresulte dem Benutzer oft in einer integrierten Entwicklungsumgebung (IDE) angezeigt. Wir stellen die IDE für Dafny—eine Programmiersprache, ein Verifikationswerkzeug, und ein Beweisassistent—vor, welche zwei Probleme in aktuellen IDEs für Programmverifikationswerkzeuge angeht: geringe Reaktionsfreudigkeit und die mangelhafte Verständnisförderung bei nicht offensichtlichen Verifikationsfehlern. Zu diesem Zweck stellen wir neue Techniken vor und integrieren existierende Techniken zur Verbesserung der Benutzerfreundlichkeit. Dies erlaubt der IDE dem Benutzer Verifikationsrückmeldungen anzuzeigen, während er tippt, und ihm unauffällig und je nach Bedarf hilfreiche Informationen über das Programm oder fehlgeschlagene Verifikationsversuche zukommen zu lassen. Das führt dazu, dass es dem Benutzer möglich ist schnell Erkenntnisse über das Programm oder die Ursache der partiellen Verifikationsresulte zu erlangen.

Um die Reaktionsfreudigkeit des Programmverifikationswerkzeugs während solcher Interaktionen mit dem Benutzer zu erhöhen, haben wir ein System entwickelt, um feinkörnig

Verifikationsresultate zwischenzuspeichern. Dieses System benutzt sowohl den Call-Graph wie auch den Control-Flow-Graph des Programms um die Verifikationsbemühungen auf die Teile des Programms zu konzentrieren, die von den letzten Änderungen des Benutzers betroffen sind. Das Novum liegt dabei darin, wie das ursprüngliche Programm mit partiellen Verifikationsresultaten aus dem Zwischenspeicher instrumentiert wird, um unnötige Arbeit für das Verifikationswerkzeug zu vermeiden. Indem wir unsere Technik zum Ausdrücken von partiellen Verifikationsresultaten einsetzen, sind wir in der Lage Verifikationsresultate aus dem Zwischenspeicher selbst dann zu verwenden, wenn Annahmen im Programm (zum Beispiel aufgrund von modularer Beweisführung über Aufrufe mittels der Nachbedingung des Aufgerufenen) von der Änderung betroffen sind.

Acknowledgments

During the last years, I have met many people that influenced me and my work in one way or the other. Here, I would like to thank all of them for their advice, support, and friendship.

My advisor, Prof. Dr. Peter Müller, influenced this work in innumerable ways. For one, he gave me the freedom and time to identify the research topic I wanted to work on for the last few years. During this time, he gave me the freedom to explore different approaches and to look into different facets of this topic. Even when things did not work out as planned, his door was always open to discuss alternative approaches or to simply confirm that we were dealing with a challenging problem that would require more thought. Often, I would leave his office with more questions than answers. However, finding answers to those questions the hard way would usually lead me in the right direction or into new scientific territory. I am tremendously grateful for this learning experience.

I would also like to thank the other members of my Ph.D. committee—Prof. Dr. Işil Dillig, Dr. Francesco Logozzo, and Prof. Dr. Bertrand Meyer—for taking the time to review this thesis and for their feedback and support.

I am incredibly grateful to my other close collaborators and friends, Maria Christakis and Prof. Dr. Rustan Leino.

Without Maria our research project would have most certainly taken a completely different turn. She did not only bring a fresh and dynamic view to this project, but also inspired us to be more pragmatic and goal-oriented when it was called for. I will miss our office discussions and the exciting ideas that were often sparked by a seemingly mundane question. Thank you very much for the great times we had!

There are many things to thank Rustan for: for giving me a chance to spend a wonderful summer in Redmond, for showing me how to turn cool ideas into useful tools, for his hospitality and many more. However, I am probably most grateful for helping me regain my faith in research and tools that people will actually enjoy using.

I would also like to thank all the current and former members of our group that I had the pleasure of working with: Dimitar Asenov, Dr. Pontus Boström, Prof. Dr. John Tang Boyland, Lucas Brutschy, Dr. Adam Darvas, Prof. Dr. Werner Dietl, Dr. Cédric Favre, Dr. Pietro Ferrara, Uri Juhasz, Dr. Ioannis Kassios, Prof. Dr. Laura Kovács, Dr. Hermann Lehner, Milos Novacek, Dr. Arsenii Rudich, Dr. Joseph Ruskiewicz, Malte Schwerhoff, Dr. Alex Summers, and Marlies Weissert. Thank you for creating such a great working environment! The same holds for Prof. Dr. Martin Vechev and the members of his group: Pavol Bielik, Andrei Marian Dan, Dimitar Dimitrov, Veselin Raychev, and Gagandeep Singh.

I would also like to thank all the great researchers and fellow interns that I met at Microsoft Research in Redmond. In particular, I would like to thank the Clousot team (Dr. Michael Barnett, Dr. Manuel Fähndrich, Dr. Francesco Logozzo, and Herman Venter). Thank you very much for all those interesting discussions and for helping us with CCI and Clousot! Large parts of this work would not have been possible without your support. I would also like to thank the Pex team (Dr. Jonathan "Peli" de Halleux and Nikolai Tillmann) for their support.

I am also very grateful to all the students I supervised (Florian Egli, Roger Koradi, David Rohr, and Malte Schwerhoff). Thank you for all the hard work!

Last, but not least, I would like to thank my friends and family for their support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software affects our everyday life in more and more ways. Our phone's alarm clock may wake us up such that we get to work on time; on the way there, we might take a train that is still operated by a human, but controlled by software. Software defects may, thus, lead to anything from minor annoyances—such as being late for work—to catastrophic disasters—such as being injured, or worse, on our train ride to work.

Program analysis tools have shown to help programmers in preventing such defects before they negatively affect us. In recent years, such tools have been increasingly applied to real-world software due to major advances in the underlying program analysis techniques. Examples of such tools include both static analyzers—such as ASTREÉ [38] for analyzing embedded systems, Clousot [52] for analyzing .Net code, and SLAM [2] for analyzing device drivers—and automatic test case generation tools—such as Pex [109] for automatically generating unit tests for .Net code and SAGE [62, 63] for automatically testing applications that process files.

Test case generation tools traditionally explore a subset of all possible program executions to find defects. This *under-approximation* of the possible program executions guarantees that only *genuine* errors are reported to the user. On the other hand, static analyzers traditionally consider additional program executions that are not actually possible in the analyzed program. This makes it possible to efficiently analyze programs with a large or even infinite number of program executions and to prove the correctness of the analyzed program in case the analysis *over-approximates* the possible program executions (i.e., is sound). As a consequence, static analyzers may also report *spurious* errors that do not reveal real defects in the analyzed program.

In practice, many static analyzer neither under- nor over-approximate the program execu-

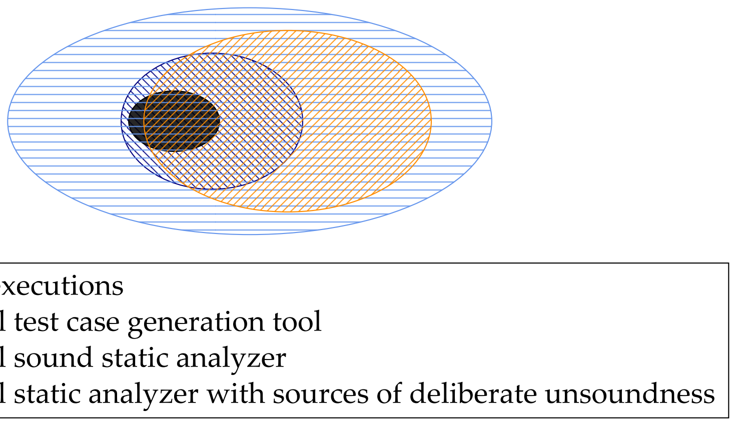

▨ possible program executions
■ analyzed by typical test case generation tool
☰ analyzed by typical sound static analyzer
▨ analyzed by typical static analyzer with sources of deliberate unsoundness

Figure 1.1: Differences between typical representatives of program analysis tools. Typical test case generation tools *under*-approximate the possible program executions, while typical sound static analyzers *over*-approximate the possible program executions. In contrast, typical static analyzers with sources of deliberate unsoundness neither over- nor under-approximate the possible program executions.

tions of the analyzed program. Their *designers* often decide to trade soundness for other qualities—such as precision, performance, and automation—by deliberately ignoring certain checks (e.g., that a method respects its write effect specification) or by deliberately making assumptions that do not hold for all program executions (e.g., that no arithmetic overflow occurs). For instance, Clousot [52] ignores certain side-effects due to aliasing, ESC/Java [56] unrolls loops a fixed number of times, HAVOC [1] uses write effect specifications without checking them, and Spec# [5] ignores arithmetic overflow and does not consider exceptional control flow, to name a few. As a result, such analyzers with deliberate sources of unsoundness may not report all errors to the user and a user cannot conclude that a program is correct if no errors are reported. Figure 1.1 summarizes the differences between typical representatives of program analysis tools by illustrating the program executions that are analyzed by each tool.

These characteristics of static analyzers motivate why verification results are often *partial* in practice: (1) some assertions of a program have neither been verified nor have been shown to lead to a defect (i.e., may be spurious errors) and (2) some program executions—including ones that may result in defects—have been ignored due to sources of deliberate unsoundness in the analysis.

We have already illustrated how these characteristics may provide the user with a false sense of correctness unless an analyzer explicitly warns the user about sources of deliberate unsoundness that could have affected the verification results. Obviously, the same applies if those results are shared with other program analysis tools. In principle, such collaboration between tools would be very tempting since it would allow users to combine the strengths of several such tools to obtain a program analysis tool chain that outperforms each individual component. However, in practice, it has not been possible to combine most practical, state-of-the-art tools both efficiently and soundly. Hence, most such tools are designed as standalone tools that primarily report errors to the *user*.

Our treatment of partial verification results solves this important problem by making it possible to annotate programs with verification results of static analyzers such that subsequent runs of program analysis tools can focus on properties that have not yet been verified statically; as mentioned earlier, this can be either due to failed verification attempts or due to sources of deliberate unsoundness.

This ability to express verification results for a wide range of tools sets our work apart from existing work on combining sound static analyzers. In other words, it provides a basis for formally annotating an analyzed program such that the verification results of a static analyzer with or without sources of deliberate unsoundness are *soundly* captured; i.e., a property is only marked as verified for a given program execution if a static analyzer has soundly verified the property on this program execution. In addition, it makes it possible to determine the *complete* set of errors that should be reported to the user without simply accumulating the errors—including duplicate or contradictory ones—of each tool. More specifically, each assertion that has not been fully verified reflects an error. Consequently, a user does not need to worry about errors that are *deliberately* missed by a tool—as opposed to accidentally due to bugs in a tool.

Since most tools are not completely sound *by design* our technique for expressing partial verification results, for the first time, enables a flexible approach for making collaborative program analysis feasible and attractive in practice. This opens up the possibility of developing new tools that effectively build on existing static analyzers.

Our technique is not only important on a technical level as a way for expressing the partial verification results of a static analyzer. It can also be seen as a tool for formally reasoning about static analyzers that do not always live up to the text book ideal of a sound analyzer, about which we are used to reason using techniques such as abstract interpretation [35]. In particular, we are now able to phrase and address fundamental research questions, such as:

- How can we soundly combine different program analysis tools such that each tool efficiently benefits from the partial verification results of the tools that have been run earlier?

- How can we express the partial verification results of a practical static analyzer?
- How can we use partial verification results to evaluate the deliberate unsoundness in a static analyzer?
- How can we help the user in understanding why the verification results of a static analysis run are only partial?
- How can we reduce the effort in collecting those partial verification results if a program is only changed slightly?

We provide answers to these questions in the following chapters. More specifically, the main contributions of this thesis can be summarized as follows:

Language constructs for encoding partial verification results To annotate programs with partial verification results, we introduce two programming language constructs for capturing under which unsound assumptions a property has been shown to hold by a static analyzer. For instance, a static analyzer may insert these language constructs into the analyzed program to express that it has verified some assertions in the program fully (i.e., soundly), some partially (e.g., since arithmetic overflow was ignored), and others not at all.

The semantics of the two new language constructs is defined in terms of assignments and regular `assume` statements. Since both of these statements are understood by essentially all existing program analysis tools, any such tool can benefit from partial verification results *immediately* and without changes to its inner workings.

To demonstrate that our technique for expressing partial verification results can be applied in a wide range of settings and provides a flexible way for sharing information between program analysis tools, we describe several novel use cases from diverse areas of program analysis—such as test case generation, specification inference, counterexample-based error reporting, and static analysis.

Architecture for collaborative static analysis and testing One such use case describes an architecture for combining static analyzers with other analyzers and with test case generation tools [25]. By expressing the partial verification results of a static analyzer as an annotated program, a subsequent static analyzer can analyze the annotated program and, thereby, focus its verification effort on the properties that have not been fully verified yet.

The same holds for test case generation tools—such as Pex [109]. Such tools can pick up the annotations in the annotated program to reduce the testing effort by not checking properties that have already been checked or inferred by static analyzers. For instance, a test generation tool can avoid testing assertions that have been fully verified or can only test them for cases that have not yet been checked statically.

While we do not discuss this use case in detail, test case generation tools can also be designed or extended to specifically target properties that are not soundly checked by static analyzers or are inherently difficult to deal with using static techniques. For instance, we have developed a technique that synthesizes parameterized unit tests for detecting object invariant violations [26, 23] by specifically targeting scenarios [94, 49] which are ignored by most static analyzers. Another technique in the same spirit is concerned with dynamic test generation in the presence of static fields and initializers [24, 23]. Both of these techniques were developed in response to questions addressed in this work about the limitations in static analyzers.

To reduce the testing effort even more, we have developed two sound inference techniques for inserting additional instrumentation into the program that allows test case generation tools based on dynamic symbolic execution to focus the effort on unverified program executions [28, 23]. This is achieved by pruning verified program executions and prioritizing unverified program executions. For instance, a test case generation tool or a second static analyzer should focus on program executions with arithmetic overflows if all assertions in a program have already been verified by a static analyzer that ignores arithmetic overflow. We provide an overview of this use case for expressing partial verification results in Section 2.3.1.

Documentation of all sources of deliberate unsoundness in a static analyzer We describe the first systematic effort to document all sources of deliberate unsoundness in an industrial-strength static analyzer [27]. We focus on Clousot, a widely-used, commercial static analyzer.

Based on this documentation, we developed a wrapper for Clousot that automatically annotates .Net programs with partial verification results by expressing most sources of deliberate unsoundness in Clousot. As described in Section 2.2, we can also use this wrapper to soundly share Clousot's partial verification results with Pex in order to reduce the test effort.

Experimental evaluation of deliberate unsoundness in a static analyzer We run an experimental evaluation [27] that, for the first time, sheds light on how often the unsound assumptions of a static analyzer are violated in practice and whether they cause the analyzer to miss bugs. For this purpose, we made use of our wrapper for Clousot and developed a suitable runtime instrumentation for detecting violations of unsound assumptions.

This allowed us to detect such violations in several open-source projects. Our manual inspection of all methods with such violations showed that no errors were missed due to an unsound assumption, which suggests that Clousot's unsoundness does not compromise its effectiveness. Such findings can guide users of static analyzers in using them fruitfully, and help their designers in striking a good balance between soundness and other qualities of an analyzer, such as precision, performance, and automation.

Sharing partial verification results with users and explaining them Our technique for expressing partial verification results provides a flexible and expressive basis for integrating program analysis tools by exchanging annotated programs. However, eventually the verification results need to be presented *and* explained to the user.

Nowadays, this task is frequently orchestrated by an integrated development environment (IDE) that incorporates one or more program analysis tools. In recent years, this task has become more important and more challenging due to the fact that program verifiers and interactive theorem provers have become more powerful and, thus, more suitable for verifying large programs or proofs. As a consequence, a wider audience of non-experts are interested in using such tools for improving software quality and correctness. This has demonstrated the need for improving the user experience of these tools to increase productivity and to make them more accessible to non-experts.

To illustrate this, we present the IDE for Dafny—a programming language, verifier, and proof assistant—that addresses two issues present in most state-of-the-art IDEs for program verifiers: low responsiveness and lack of support for understanding non-obvious verification failures [82]. To this end, we present both new techniques and integrate existing techniques to improve the user experience of the IDE.

We thereby push the state-of-the-art closer towards a verification environment that can provide verification feedback as the user types and can present more helpful information about the program or failed verification attempts in a demand-driven and unobtrusive way. This allows the user to quickly gain insights about the program and to understand the cause of partial verification results. Such insight and understanding is crucial for supporting users in developing fully-verified programs after a series of interactions with the program verifier.

Fine-grained caching of verification results To increase the responsiveness of the verifier during such interactions with the user, we designed a system for fine-grained caching of verification results [83]. The caching system uses the program's call graph and control-flow graph to focus the verification effort on just the parts of the program that were affected by the user's most recent modifications. The novelty lies in how the original program is instrumented with information from the cache to avoid unnecessary work for the verifier.

This instrumentation makes use of our technique for expressing partial verification results to capture which cached verification results are still valid in the current version of the program. For instance, it is designed to reuse cached verification results for the current version of a program in two common scenarios: when an isolated part (e.g., one of two branches or a loop body) has been changed, and when the specification of a callee has been changed.

A key insight behind this work is that by expressing partial verification results we are able to reuse parts of the cached verification results even if assumptions in the program (e.g., due to modular reasoning about calls by means of the callee's postcondition) are affected by a change. We describe the architecture and algorithms of the caching system and our experimental evaluation sheds light on how much caching improves the performance of the verifier in practice.

Outline

Chapter 2 introduces an architecture for analyzing programs collaboratively and defines two programming language constructs that allow us to annotate programs with partial verification results from a wide range of static analyzers—including ones with sources of deliberate unsoundness. It demonstrates the expressiveness of these language constructs by illustrating how to capture common sources of unsoundness in static analyzers and program verifiers and by outlining several other novel use cases for expressing partial verification results in test case generation tools, static analyzers, or inference tools.

Chapter 3 provides a more comprehensive case study for how one can express the verification results of a practical static analyzer with sources of deliberate unsoundness and how one can evaluate deliberate unsoundness in a static analyzer. In particular, our evaluation sheds light on whether Clousot's unsound assumptions are violated in practice and whether such violations cause Clousot to miss bugs.

Chapter 4 presents the IDE for the Dafny programming language and verifier. The IDE's primary focus is on *swiftly* sharing partial verification results and explaining them to the user. To this end, it provides feedback as the user types and is able to present more helpful information about the program or failed verification attempts in a demand-driven and unobtrusive way.

Chapter 5 presents a system for fine-grained caching of verification results to improve the responsiveness of the Dafny IDE. Our technique instruments the program with cached verification results by expressing which properties still hold for the current version of the program using partial verification results.

We discuss related work separately in each chapter and conclude in Chapter 6.

Chapter 2

Partial Verification Results

Program analysis tools are increasingly applied to detect defects in real-world programs and are starting to make their way into the development workflow of software developers. Among other things, this manifests itself in a deeper and more unobtrusive integration into widely-used IDEs. These tools range from relatively simple heuristic tools, over tools based on abstract interpretation, dynamic symbolic execution, or symbolic model checking, to verifiers based on automatic theorem proving.

Usually, the primary focus of such tools is on sharing their results (e.g., errors and inferred invariants) with the user. In addition, one could imagine sharing those results with other program analysis tools to effectively combine their strengths. Ideally, such collaboration would enable a program analysis tool chain that outperforms each individual tool. However, in practice, it is not possible to combine most practical program analysis tools both soundly and efficiently due to two inherent characteristics of most practical static analyzers.

First, static analyzers traditionally also consider some program executions that are actually not possible in the analyzed program. This allows static analyzers to efficiently analyze programs with many or even infinitely many program executions and makes them suitable for proving the absence of errors if an *over-approximation* of all possible program executions is considered (i.e., if the static analyzer is sound). However, this comes at a price: static analyzers may end up also reporting *spurious* errors that cannot occur in any of the possible program executions.

Second, in practice, many static analyzers do not always over-approximate the possible program executions. The designers of the analyzer often decide to trade soundness for other important qualities of static analyzers—such as precision, performance, and automation—by deliberately ignoring certain checks (e.g., that a method respects its write effect specification) or by deliberately making assumptions that do not hold for all program executions (e.g., that no arithmetic overflow occurs). For instance, Clousot [52] ignores certain side-effects due to aliasing, ESC/Java [56] unrolls loops a fixed number of times, HAVOC [1] uses write effect specifications without checking them, and Spec# [5] ignores arithmetic overflow and does not consider exceptional control flow, to name a few. Consequently, static analyzers with such sources of deliberate unsoundness do not provide definite guarantees about the correctness of programs and cannot always ensure the absence of errors.

As a result of these two inherent characteristics, most practical static analyzers only provide *partial* verification results since (1) some assertions of a program have neither been verified nor have been shown to lead to a defect (i.e., may correspond to spurious errors) and (2) some program executions—including ones that may result in defects—have been ignored due to sources of deliberate unsoundness.

In practice, existing static analyzers to not make these partial verification results explicit in their output which prevents a user from getting definite correctness guarantees and a program analysis tool from making effective use of those results. More specifically, a tool cannot easily build on the analysis of another tool to reduce its analysis effort. This is ineffective and, instead of consolidating the set of errors that are reported to the user, each tool will only increase the set of errors, which may include duplicate, spurious or even contradictory ones.

With this in mind, we propose a technique that enables the sound and effective combination of multiple, complementary program analysis tools by making explicit in its output which properties have been checked and under which unsound assumptions. In particular, by identifying and documenting all sources of unsoundness, a static analyzer becomes sound relatively to its sources of unsoundness. In other words, one could prove that it is sound for program executions where all checks hold that are deliberately ignored and where all deliberate, unsound assumptions hold.

Such a proof would ensure that all sources of deliberate unsoundness have been identified. However, proving this for a industrial-strength static analyzer is often not practical. In such cases, the designers need to make sure that all sources of deliberate unsoundness have been made explicit in the tool's output. We have used this approach for identifying the sources of deliberate unsoundness in the static analyzer Clousot with help from their designers (see Section 3.1 for more details on this process). Even if a tool's output fails to capture a source of deliberate unsoundness, the output is still more useful for subsequent tools than if none of the sources of deliberate unsoundness were disclosed. Note that, the same applies to accidental unsoundness, for instance due to bugs in the implementation of the static analyzer.

The two contributions of the work described in this chapter are:

1. We propose a simple language extension for sharing *partial* verification results between program analysis tools (including ones with sources of deliberate unsoundness) by annotating a program. These annotations are expressed via two new programming language constructs whose semantics is defined in terms of assignments and assumptions. They are, thus, easy to support by a wide range of tools. The first construct is used for expressing *explicit (unsound) assumptions* at the program points where they are made during the analysis. This allows *modular* program analysis tools to express their verification results *locally* in the checked module (for instance, locally within a method). This is crucial for allowing subsequent analysis tools to also operate modularly. For instance, such tools may include automatic test case generation tools that generate *unit tests* for the module. The second construct is used for expressing that a property was found to hold soundly or under certain unsound assumptions at a given program point. In particular, this allows us to mark assertions as fully verified, partially verified (that is, verified under certain unsound assumptions), or not verified. We demonstrate that our language constructs can be used to effectively express partial verification results of mainstream static analyzers such that they can subsequently be used by other tools. In particular, we show how the first construct can be used to express and evaluate most sources of deliberate unsoundness in the static analyzer Clousot in Chapter 3. To demonstrate that our technique for expressing partial verification results can be applied in a wide range of settings and provides a flexible way for sharing information between program analysis tools, we describe several novel use cases from diverse areas of program analysis—such as test case generation, specification inference, counterexample-based error reporting, and static analysis.

2. We present an architecture for combining static program analysis tools with test case generation tools by exchanging the partial verification results. This makes it possible to automatically generate unit tests for execution traces that have not yet been fully verified, thereby providing the user with a choice on how much effort to devote to static checking and how much to testing. For example, a user might run a static analyzer without devoting any effort to making the verification succeed (for instance, without providing auxiliary specifications, such as loop invariants). The static analyzer may prove some properties correct, and our technique enables the effective testing of all others. Alternatively, a user might try to verify properties about critical components of a program and leave any remaining properties (e.g., about library components) for testing. Consequently, the degree of static checking is configurable and may range

from zero to complete. Our architecture enables a tool chain that directs the effort of subsequent program analysis tools to partially-verified or unverified properties. This makes any subsequent analysis more targeted and, therefore, more effective. In the case of test case generation tools it may lead to smaller and more effective test suites. We have implemented a tool chain that combines the static analyzer Clousot [52] with the automatic test case generation tool Pex [109], which makes use of dynamic symbolic execution [60, 19]. This allowed us to identify situations in which this combination finds more errors and proves more properties than static checking alone, testing alone, and combined static checking and testing without our technique. We have taken this combination one step further by developing two static analyses that use the partial verification results to infer conditions that can be used to guide dynamic symbolic execution toward unverified executions [28, 23]. This makes it possible to focus the testing effort even more effectively. In Chapter 5 we demonstrate how our technique can be used for fine-grained caching of verification results in the Dafny IDE [82] and the underlying Boogie [4] verification engine by directing the re-verification effort to properties that have been affected by the most recent edits to a program.

This chapter is based on a paper that was presented at the *International Conference on Formal Methods* in 2012 [25].

Outline

Section 2.1 introduces our technique for annotating programs with partial verification results and demonstrates how this allows us to capture common sources of deliberate unsoundness in static analyzers. Section 2.2 provides an overview of how partial verification results can be used to combine static analyzers with test case generation tools. Section 2.3 provides an overview of various ways in which other tools can make use of partial verification results. We review related work in Section 2.4 and summarize our results in Section 2.5.

2.1 Language Constructs for Encoding Partial Verification Results

In this section, we present the general technique for expressing partial verification results and explain how it can be used for capturing common sources of deliberate unsoundness. For this purpose, we extend the programming and specification language with two new constructs: (1) `assumed` statements to make unsound assumptions of a static analyzer explicit at the program points where they are made, and (2) partially-justified `assume` statements, which generalize traditional `assume` statements, to encode properties that have been found to hold at certain program points either soundly or under certain explicit assumptions.

For convenience, we also introduce partially-verified assertions, which generalize traditional assertions and can be expressed using partially-justified `assume` statements and regular assertions. We define the semantics of these new language constructs by expressing them in terms of assignments and regular `assume` statements, which have a well-defined semantics (for instance, in terms of weakest preconditions [45]).

2.1.1 Language Constructs

An `assumed` statement of the form `assumed` P `as` a records that a static analyzer assumed property P at a given program point. P is a predicate of the assertion language, and a is a unique *assumption identifier*, which can be used in partially-justified `assume` statements or partially-verified assertions to express that a property has been shown to hold under a Boolean expression over assumption identifiers. `assumed` statements do not affect the semantics of the program, but they are used to define the semantics of partially-justified `assume` statements and partially-verified assertions, as we discuss below. In particular, our `assumed` statements are different from the classical `assume` statements, which express properties that any program analysis tool may take for granted and need not check.

To illustrate, let us consider the C# method of Figure 2.1 which has been annotated using .NET Code Contracts [51]. The gray boxes express the partial verification results of a static analyzer that ignores arithmetic overflow (e.g., Spec#). Method `Transfer` from class `Account` transfers an amount of money from the current account to a different account `rcvr` provided that the positive amount is less than or equal to 50'000 and there is enough money in the current account; otherwise, the transfer needs to be reviewed. The assertion on line 17 states that the balance of the account `rcvr` should be increased.

In this example, we use two `assumed` statements (on lines 8 and 11) to express that the static analyzer ignored arithmetic overflow when verifying the method (see Section 2.1.3 for more details on how predicates `NoOverflow_sub` and `NoOverflow_add` can be implemented). Note that we also allow programmers, in addition to static checkers, to add `assumed` statements in their code. This application is explained in more detail in Section 2.3.7.

In order to record partial verification results, we use partially-justified `assume` statements of the form `assume` P `provided` A, where P is a predicate of the assertion language and A is a Boolean expression over assumption identifiers. This construct can be used to mark

```
 1 public void Transfer(Account rcvr, int amount) {
 2   requires rcvr != null && rcvr != this;
 3
 4   if (amount <= 0 || 50000 < amount || balance < amount) {
 5     ReviewTransfer(rcvr, amount);
 6   } else {
 7     assumed NoOverflow_sub(balance, amount) as o0;
 8     balance -= amount;
 9     assume rcvr != null provided o0;
10     assumed NoOverflow_add(rcvr.balance, amount) as o1;
11     rcvr.balance += amount;
12     assume rcvr != null provided o0 && o1;
13     if (balance < 500 && balance < rcvr.balance) {
14       SuggestLoanFrom(rcvr);
15     }
16     assume rcvr != null provided o0 && o1;
17     assert old(rcvr.balance) < rcvr.balance verified o0 && o1;
18   }
19 }
```

Figure 2.1: Example program that shows the partial verification results of a static analyzer that ignores arithmetic overflow. We use the keyword `requires` to denote preconditions and we use the keyword `old` to denote expressions that should be evaluated in the pre-state of the method. The assertion is violated if the addition overflows. The gray boxes represent annotations that document the partial verification results.

an assertion `assert` Q as partially-verified under an expression B by inserting a partially-justified assumption `assume` Q `provided` B before the assertion.

In our example from Figure 2.1, both assumption identifiers are used later on (e.g., in the partially-justified `assume` statement on line 12) to express that some properties have only been shown to hold under those assumptions.

For convenience and to support this common use case, we use partially-verified assertions of the form `assert` P `verified` A, where P is a predicate of the assertion language and A is a Boolean expression over assumption identifiers. For instance, in our example from Figure 2.1, the assertion on line 17 is verified under both explicit assumptions made by the static analyzer. In general, we mainly use partially-justified `assume` statements to express properties that were inferred by a static analyzer and to mark implicit assertions (e.g., about the absence of null-dereferences or division-by-zero) as partially-verified.

When a static analyzer verifies an assertion, it marks it as verified under all the assumptions used for its verification. By default, an assertion is unverified; i.e., A is the expression **`false`** since no static analyzer has verified the assertion. To mark an assertion as verified a static analyzer needs to update the expression A to be the disjunction of the old expression and the condition under which it verified the assertion. The assertion is fully verified if the expression A is true; e.g., this may be the case if at least one static analyzer has verified the assertion without making any assumptions. Otherwise, the assertion is partially verified.

Note that it is up to each individual static analyzer to determine which assumptions it used to show that a certain property holds. For instance, a verifier based on weakest preconditions could collect all assumptions that are on any path from the start of a method to the assertion; additionally, it could try to minimize the set of assumptions using techniques such as slicing to determine which assumptions actually influence the truth of the assertion.

2.1.2 Semantics

The goal behind annotating programs with partial verification results is to let program analysis tools benefit from the results of previous runs of static analyzers. This is achieved by defining a semantics for partially-justified `assume` statements which expresses that a property holds under a Boolean condition over assumption identifiers. A program analysis tool gets to assume the property provided that the Boolean condition holds.

By expressing partially-verified assertions in terms of partially-justified `assume` statements and regular assertions we achieve similar benefits for partially-verified assertions. In particular, for fully verified assertions a program analysis tool does not have to show anything. For partially-verified assertions, it is sufficient for a tool to show that the assertion holds even if the condition under which it has been verified does not hold.

We formalize this intuition by demonstrating how to express the two new language constructs in terms of assignments and regular `assume` statements, which have a well-defined semantics (for instance, in terms of weakest preconditions [45]) and are available in most existing tools. The latter allows any such tools to benefit from partial verification results *immediately* and without changes to their inner workings.

For expressing explicit assumptions, we introduce a Boolean *assumption variable* for each assumption identifier that occurs in an `assumed` statement; all assumption variables are initialized to `true`. For the example from Figure 2.1, this can be seen on line 4 of Figure 2.2.

For modular static checking, which checks each method individually, assumption variables are local variables of the method that contains the `assumed` statement. Assumptions of whole-program checking may be encoded via global variables, that are, for instance, initialized in a main method. Unless stated explicitly, we assume that explicit assumptions are

```
1  public void Transfer(Account rcvr, int amount) {
2    requires rcvr != null;
3
4    bool o0 = true; bool o1 = true;
5    if (amount <= 0 || 50000 < amount || balance < amount) {
6      ReviewTransfer(rcvr, amount);
7    } else {
8      o0 = o0 && NoOverflow_sub(balance, amount);
9      balance -= amount;
10     assume !(o0) || (rcvr != null);
11     o1 = o1 && NoOverflow_add(rcvr.balance, amount);
12     rcvr.balance += amount;
13     assume !(o0 && o1) || (rcvr != null);
14     if (balance < 500 && balance < rcvr.balance) {
15       SuggestLoanFrom(rcvr);
16     }
17     assume !(o0 && o1) || (rcvr != null);
18     assume !(o0 && o1) || (balance < old(balance));
19     assert old(rcvr.balance) < rcvr.balance;
20   }
21 }
```

Figure 2.2: Method `Transfer` after expressing partial verification results (gray boxes) in terms of assignments and regular `assume` statements. We use Boolean *assumption variables* that are initialized to `true` and are assigned to once (at the corresponding `assumed` statement) to track explicit assumptions. We express partially-justified `assume` statements using regular `assume` statements.

local. However, one may, for instance, decide to use an additional qualifier `globally` after the Boolean condition in an `assumed` statement to distinguish global explicit assumptions. A statement of the form `assumed` P `as` a can now be expressed as the following assignment, where `a` is the assumption variable that corresponds to the assumption identifier a (see, for instance, line 8 in Figure 2.2):

```
a = a && P;
```

Intuitively, we use the assumption variable to accumulate each property that is assumed when executing the corresponding `assumed` statement. Note that it is necessary to accumulate the assumed properties, for instance, in the presence of loops. An assumption variable will remain true until condition `P` evaluates to false when executing the `assumed` statement. In this case, the execution has reached a state which was ignored by the static analyzer that introduced the `assumed` statement. Our semantics ensures that an assumption is evaluated in the state in which it is made rather than the state in which it is used.

The assumption variables allow us to define the semantics of partially-justified `assume` statements. A statement of the form `assume` P `provided` A can be expressed as the following regular `assume` statement:

```
assume !A || P;
```

The assumed implication expresses that the property P has been (soundly) shown to hold provided that condition A holds and, therefore, any subsequent tool gets to rely on this fact. For instance, on line 13 of Figure 2.2 we use this to express that the implicit assertion for the field dereference on line 14 has been verified under the both explicit assumptions `o0` and `o1`. We can even express that a static analyzer determined a certain program point to be unreachable under condition A by inserting the following statement:

```
assume false provided A;
```

By expressing partially-verified assertions using partially-justified `assume` statements, we effectively weaken the property that still needs to be verified. We can demonstrate this more formally by deriving the weakest precondition under which a statement of the form `assert` P `verified` A will not fail and establish the postcondition R after the assertion:

$$\begin{aligned} wp(\texttt{assert } P \texttt{ verified } A, R) &\equiv wp(\texttt{assume } P \texttt{ provided } A; \texttt{assert } P, R) \\ &\equiv wp(\texttt{assume } A \Rightarrow P; \texttt{assert } P, R) \\ &\equiv (A \Rightarrow P) \Rightarrow (P \wedge R) \\ &\equiv (A \Rightarrow P) \Rightarrow ((A \vee P) \wedge R) \qquad (2.1) \end{aligned}$$

Here, the last step makes use of the additional assumption $A \Rightarrow P$ to weaken the property P to $A \vee P$. More intuitively, this shows that a partially-verified assertion will not fail if condition A holds *or* the asserted property P holds anyway. The disjunction weakens the property that still needs to be verified and therefore, lets tools benefit from the partial results collected during previous static analysis runs. Note that the final formula (2.1) can be further simplified to R in the special case that A is true; i.e., the assertion has been fully verified and nothing remains to be verified about the assertion itself.

Note that, due to the way in which each static analyzer marks an assertion as verified, the expression A will be a disjunction with one disjunct A_i for each static analyzer that verified the assertion under condition A_i. Consequently, we do not merely accumulate the results of independent static analysis runs. Thanks to the above semantics for partially-verified assertions and the way in which the expression A is updated by each static analyzer, the property to be verified typically becomes weaker with each static analysis run. Therefore, many properties can eventually be fully verified, without making any further assumptions. The remaining ones can be tested or verified interactively.

2.1.3 Capturing Common Sources of Deliberate Unsoundness

In this section, we demonstrate how our language constructs from Section 2.1.1 can be used to express partial verification results for static analyzers with sources of deliberate unsoundness. To this end, we describe how to capture three sources of deliberate unsoundness using partial verification results. Chapter 3 provides a more comprehensive description of how to capture the sources of deliberate unsoundness in Clousot.

Unbounded Integers

A common source of unsoundness in static analyzers is caused by ignoring overflow in bounded integer arithmetic, as in the case of Clousot, ESC/Java and Spec#. By ignoring arithmetic overflow, a tool is able to reason about mathematical integers. This is, for instance, more efficient when using SMT solvers in deductive verifiers or when using non-disjunctive numerical abstract domains in static analyzers based on abstract interpretation. Independently, since arithmetic overflows do not immediately lead to exceptions in many languages (e.g., Java, C#) many tools decide not to report them as errors by default to avoid spurious errors.

To capture this source of unsoundness using our language constructs, we introduce an explicit assumption for each operation that may result in an arithmetic overflow. The assumed condition can be expressed efficiently by stating that the bounded arithmetic operation returns the same result as its mathematical counterpart.

For instance, in our example from Figure 2.1 we can express the abstract condition `NoOverflow_add(rcvr.balance, amount)` more concretely as:

```
(long)(rcvr.balance + amount)
  == (long)rcvr.balance + (long)amount
```

In this case, we compare the result of performing the addition on values of type `int` to the result of performing the addition on value of type `long`, which does not lead to an overflow.

Original loop:

```
while (C) {
  B
}
```

Transformed loop:

```
if (C) {
  B
}
assumed !C as a;
while (C) {
  B
}
```

Figure 2.3: Loop transformation and explicit assumption about loops that are unrolled 1.5 times. The transformation actually unrolls the loop as well. However, without changing its semantics (i.e., soundly).

Loop Unrolling

To avoid the annotation overhead of loop invariants, some static analyzers unroll loops a fixed number of times. For instance, ESC/Java unrolls loops 1.5 times by default: first, the condition of the loop is evaluated and in case it holds, the loop body is checked once; then, the loop condition is evaluated again after assuming its negation. As a result, the code following the loop is checked under the assumption that the loop iterates at most once.

There are at least two ways to capture this source of unsoundness. Both of them require some additional transformations of the program. For the first one, we actually unroll the loop once and subsequently introduce an `assumed` statement the states that the loop condition does not hold. The corresponding transformation is shown in Figure 2.3. Any verified assertions following the `assumed` statement are verified under this assumption. Note that the loop is still part of the transformed program so that the original semantics is preserved for downstream static analyzers, which might not make the same compromise, and test case generation tools.

To avoid the actual loop unrolling in the annotated program, one can use an alternative encoding that makes use of a ghost variable for each loop which is initialized to `0` and is incremented before every execution of the loop body. Thanks to this ghost variable we can introduce an `assumed` statement after the increment operation at the beginning of the loop body that states that the ghost variable is less than `2`. The corresponding transformation is shown in Figure 2.4. Both approaches demonstrate a powerful way to increase the expressiveness of partial verification results by enriching the program with additional *code* instrumentation.

Original loop:

```
while (C) {
  B
}
```

Transformed loop:

```
var cnt = 0;
while (C) {
  cnt++;
  assumed cnt < 2 as a;
  B
}
```

Figure 2.4: Alternative loop transformation and explicit assumption about loops that are unrolled 1.5 times. The alternative transformation does not unroll the loop, but makes use of a ghost variable `cnt` to count the number of loop iterations.

Write Effects

Another source of unsoundness that can be found in several static analyzers, such as HAVOC and ESC/Java, involves assuming write effect specifications at call sites without checking them in the callees. We can encode this by simply leaving all the required checks unverified, that is, by not annotating them with partial verification results.

2.2 Collaborative Static Analysis and Testing

This section describes how our technique for expressing partial verification results can be used to combine static analyzers with test case generation tools. To this end, we propose a tool chain that makes it possible to focus the testing effort on execution traces that have not yet been fully verified by taking partial verification results into account. This can lead to smaller and more effective test suites. In contrast, without our technique, a user would need to test programs as if no static analysis had been performed unless the analysis is actually sound. Through a running example, we discuss the motivation behind the approach and the stages of the tool chain.

2.2.1 Running Example

Let us consider the C# method in Figure 2.5 which is a minor variation of the example from Figure 2.1. Unlike in Figure 2.1, we do not require that object `rcvr` is different from the current object. Consequently, the assertion on line `verifiedAssertion` may be violated in

```
 1 public void Transfer(Account rcvr, int amount) {
 2   requires rcvr != null;
 3 
 4   if (amount <= 0 || 50000 < amount || balance < amount) {
 5     ReviewTransfer(rcvr, amount);
 6   } else {
 7     balance -= amount;
 8     rcvr.balance += amount;
 9     if (balance < 500 && balance < rcvr.balance) {
10       SuggestLoanFrom(rcvr);
11     }
12     assert old(rcvr.balance) < rcvr.balance;
13   }
14 }
```

Figure 2.5: Example program that illustrates the motivation for our technique. The assertion is violated if the addition overflows or if the current object is the same as the object `rcvr` when performing the field updates on lines 7 and 8.

two cases: (1) an overflow happens in the addition, thereby making the balance negative or (2) the current object and the object `rcvr` reference the same object, in which case the balance will not be changed (independently of arithmetic overflows).

Checking this program with the static analyzers Clousot will detect none of those errors because it ignores arithmetic overflow, it ignores that the field update on line 7 may affect the balance of object `rcvr` and, similarly, that the field update on line 8 may affect the balance of the current object. Note that the postcondition of method `SuggestLoanFrom` ensures that neither the balance of object `rcvr` nor the balance of the current object is changed. A user who is not familiar with the tool's implicit assumptions does not know how to interpret the absence of warnings. Given that errors might be missed, the code has to be tested as if the static analyzer had not run at all.

Running Pex, an automatic test case generation tool for .NET, on method `Transfer` happens to generate a test case that reveals the error due to aliasing, but misses the one due to arithmetic overflow. Which error is uncovered depends on the inputs that are generated by the underlying constraint solver for exploring the failing branch of the assertion. So, similarly to the static analyzer, errors may be missed. In this case, a user might decide to fix the error (e.g., by requiring in the precondition that the current object and the object `rcvr` are non-aliasing or by weakening the assertion). Only now, a second run of the testing tool would detect the error due to arithmetic overflow which might be considered more severe.

Instead of running tools on the program independently, we propose a tool chain that uses our technique for expressing partial verification results to share results between tools such that

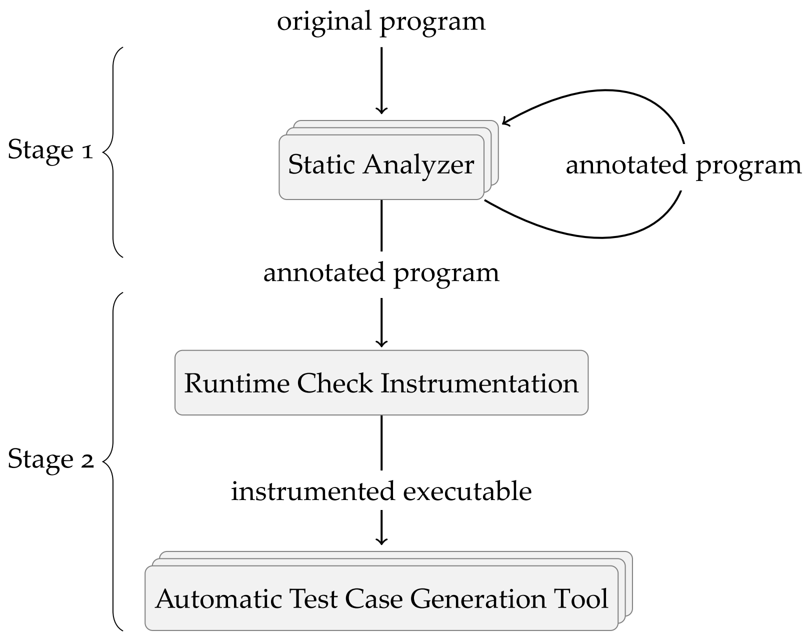


Figure 2.6: The collaborative static analysis and testing tool chain. Tools are depicted by boxes and edges represent information that is exchanged between them (e.g., programs with specifications). During the first stage, zero or more static analyzers with or without sources of deliberate unsoundness annotate the program with partial verification results. During the second phase and after instrumenting the program with runtime checks, zero or more automatic test case generation tools pick up the partial verification results to generate tests for execution traces that have not been fully verified yet.

subsequent tools can focus on unverified or partially-verified properties. This tool chain is illustrated in Figure 2.6 and consists of two stages that complement each other: collaborative static analysis and testing.

2.2.2 Stage 1: Collaborative Static Analysis

The static analysis (or verification) stage allows the user to run an arbitrary number (possibly zero) of static analyzers. Each analyzer reads the program, which contains the code, the specification, and annotations that document the partial verification results of prior static analyzers. More precisely, these annotations can express that a static analyzer found a property to hold at a certain program point either soundly or under certain (unsound) explicit assumptions. In particular, this makes it possible to mark assertions as either fully (that is, soundly) verified, partially verified under certain explicit assumptions, or not verified (that is, not attempted or failed to verify). A subsequent analyzer can then attempt to prove the assertions that have not yet been fully verified by upstream tools. For this purpose, it may assume the properties that have already been fully verified. For partially-verified assertions, it is sufficient to show that the assumptions made by a prior static analyzer hold or the assertions hold regardless of the assumptions, which simplifies the verification task. For instance, if the first checker verifies that all assertions hold assuming no arithmetic overflow occurs, then it is sufficient for a second (possibly specialized) static analyzer to confirm this assumption.

Each tool records its partial verification results in the program that serves as input to the next downstream tool. This representation is relatively compact (usually constant overhead in terms of the original program size) and universally understood by essentially all off-the-shelf program analysis tools.

The intermediate versions of the program precisely track which properties have been fully verified and which still need validation. This allows developers to stop the static analysis cycle at any time, which is important in practice, where the effort that a developer can devote to static checking is limited. Any remaining unverified or partially-verified assertions may then be covered by the subsequent testing stage.

The gray boxes in Figure 2.7 illustrate the verification result of running Clousot on the example from Figure 2.5. The static analyzer makes implicit assumptions in four places:

- about overflows for the subtraction (captured by an explicit assumption on line 7),
- about side-effects due to aliasing for the field update on line 9 (captured by an explicit assumption on line 8),
- about overflows for the addition (captured by an explicit assumption on line 11), and
- about side-effects due to aliasing for the field update on line 13 (captured by an explicit assumption on line 12).

Note that we document implicit assumptions at the place where they occur rather than where they are used to prove an assertion. This is convenient since some assumptions depend on the current execution state.

Running the static analyzer verifies several properties for this method:

- that no null-pointer is dereferenced for the field access on line 13 under explicit assumptions `o0` and `a0`, which is expressed using a partially-justified `assume` statement on line 10,

```
 1 public void Transfer(Account rcvr, int amount) {
 2   requires rcvr != null;
 3
 4   if (amount <= 0 || 50000 < amount || balance < amount) {
 5     ReviewTransfer(rcvr, amount);
 6   } else {
 7     assumed NoOverflow_sub(balance, amount) as o0;
 8     assumed NoAliasing(rcvr, this) as a0;
 9     balance -= amount;
10     assume rcvr != null provided a0 && o0;
11     assumed NoOverflow_add(rcvr.balance, amount) as o1;
12     assumed NoAliasing(this, rcvr) as a1;
13     rcvr.balance += amount;
14     assume rcvr != null provided a0 && o0 && a1 && o1;
15     if (balance < 500 && balance < rcvr.balance) {
16       SuggestLoanFrom(rcvr);
17     }
18     assume rcvr != null provided a0 && o0 && a1 && o1;
19     assert old(rcvr.balance) < rcvr.balance verified a0 && o0 && a1 && o1;
20   }
21 }
```

Figure 2.7: Example from Figure 2.5 annotated with partial verification results (in gray boxes). The annotations document the partial verification results of Clousot, a static analyzer that ignores arithmetic overflow and assumes that the current object and the object `rcvr` are not aliased when performing the field updates.

- that no null-pointer is dereferenced for the field access on line 15 under explicit assumptions `o0`, `a0`, `o1`, and `a1`, which is expressed using a partially-justified `assume` statement on line 14, and
- that no null-pointer is dereferenced for the field access on line 19 under explicit assumptions `o0`, `a0`, `o1`, and `a1`, which is expressed using a partially-justified `assume` statement on line 18, and
- that the assertion on line 19 holds under explicit assumptions `o0`, `a0`, `o1`, and `a1`, which is expressed by turning the assertion into a partially-verified `assert` statement.

Here, even though this would be possible in theory, we do not track precisely if certain explicit assumptions are indeed needed to show a given property. Instead, we conservatively include all explicit assumptions that have been introduced before to the property in the control-flow.

Note that Clousot works modularly, that is, it checks each method independently of its clients. Therefore, all explicit assumptions are local to the method being checked; for instance, method `Transfer` is analyzed independently of any assumptions in its callers. Consequently, the method's verification results are suitable for subsequent modular static analyzers or test case generation tools that produce unit tests.

Since our example actually contains errors, any subsequent static analyzer will neither be able to fully verify that all explicit assumptions always hold (this would, however, be the case for the explicit assumption about overflow on line 7 in Figure 2.7) nor that the above properties hold in case the assumptions do not. Nevertheless, the explicit assumptions document the partial verification results of the static analyzer, and we use this information to generate targeted test cases in the subsequent testing stage.

2.2.3 Stage 2: Testing

We apply dynamic symbolic execution [60, 19], also called concolic testing [103], to automatically generate parameterized unit tests from the program code, the specification, and the partial verification results collected during the static analysis stage.

Dynamic symbolic execution collects constraints describing the test data that will cause the program to take a particular branch in the execution or violate an assertion[1]. To use this mechanism, we instrument the program with assertions for those properties that have not been fully verified. That is, we assert all properties that have not been verified at all, and for partially-verified properties, we assert that the property holds in case the assumptions made by the static analyzers do not hold. This way, the properties that remain to be checked as well as the assumptions made by static checkers occur in the instrumented program, which causes the symbolic execution to generate the constraints and test data that exercise these properties.

In our example, the assertion on line 19 has been partially verified under four explicit assumptions. The instrumentation therefore adds the fact that this property has been shown to hold as the following assumption to the constraints that are used for generating test cases (where `o0`, `a0`, `o1`, and `a1` are the corresponding assumption variables):

$(\texttt{a0} \land \texttt{o0} \land \texttt{a1} \land \texttt{o1}) \Rightarrow (\textbf{old}(\texttt{rcvr.balance}) < \texttt{rcvr.balance})$.

Facts like the one above can help the testing tool by acting as lemmas that can be used by the underlying constraint solver to simplify constraints. In one case, when trying to generate inputs for the failing branch of the corresponding assertion, the lemma will tell the constraint solver that the branch is infeasible unless it can generate inputs that violate the assumptions.

[1]An assertion is viewed as a conditional statement, where one branch throws an exception. A test case generation tool aiming for branch coverage will therefore attempt to generate test data that violates the assertion.

As a consequence, the test case generation tool may spend less time trying to generate tests for failing branches that are infeasible. In another case, where the property would results in complex constraints when generating inputs for paths that do not take the failing branch, the constraint solver might find it easier to generate inputs that satisfy the assumptions, which, thanks to the lemma, will also satisfy the property itself. As a consequence, the test case generation tool might generate more tests (including ones that fail for subsequent assertions) within a given time bound for the constraint solver or generate the same number of tests more quickly.

For our simple example program, the additional facts do not result in a significant difference between running Pex on the original program and the annotated one. In particular, these facts are only used when symbolic path constraints that were collected during a concrete execution are solved, but not for guiding the dynamic symbolic execution to focus the effort on unverified program executions. To take this one step further, we have developed a technique to guide dynamic symbolic execution based on partial verification results [28, 23]. We provide an overview of this technique in Section 2.3.1.

In case the code needs to be be fully verified, an alternative second stage of the tool chain could involve proving the remaining, precisely documented program properties with an interactive theorem prover. The intention then is to prove as many properties as possible automatically and to direct the manual effort towards proving the remaining properties. Yet another alternative is to use the explicit assumptions and partial verification results for targeted code reviews.

In principle, one could even skip the second stage of the tool chain entirely. However, in this case, the partial verification results that are encoded in the program would need to be communicated to the user. This could, for instance, involve displaying error messages for all assertions that have not been fully verified. Optionally, one could prioritize the error messages based on how many static analyzers were not able to verify the corresponding assertion. This would allow users to focus on the most critical errors first.

If a user wants to understand why an assertion has been partially verified, it would even be possible to share under which condition it has been verified. For this purpose, it might make sense to show parts (e.g., relevant explicit assumptions) of the annotated program since our annotations are actually part of the programming language.

2.3 Other Use Cases for Partial Verification Results

In Section 2.2, we have presented one important use cases for expressing partial verification results: combining the partial results of several collaborative static analyzers in a way that allows them to be complemented by test case generation tools. However, there are many more applications of this technique in a variety of different settings. We present two such applications in more detail in later chapters.

First, in Chapter 3, we show how explicit assumptions can be used to express sources of deliberate unsoundness in the static analyzer Clousot and how this can provide a practical way for evaluating unsoundness in static analyzers. Second, in Chapter 5, we demonstrate a very different use case that is not immediately related to sources of deliberate unsoundness. More specifically, we use our technique for expressing partial verification results to cache verification results that were collected by Dafny, a sound static analyzer, and Boogie, its underlying (sound) verification engine. On one hand, our caching system, which is integrated in the Dafny IDE, uses fully verified assertions to express properties that are still valid in the current version of the program. On the other hand, it uses explicit assumptions to capture assumptions that were made after assuming the postcondition at a call site in an earlier version of the program. Since the postcondition in the current program may be different, we can use such explicit assumptions to mark assertions as partially-verified. This makes it possible to reduce the effort when verifying the current version of the program.

In this section, we outline a few other promising use cases for expressing partial verification results. While this collection of use cases is far from complete, it should demonstrate that our technique can be applied in a wide range of settings and for a wide range of program analysis tools.

2.3.1 Testing Unverified Program Executions

Without built-in support for partial verification results, test case generation tools merely benefit from partial verification results thanks to the additional facts that can be picked up during the constraint solving (as explained in Section 2.2.3). In particular, these facts are merely used when symbolic path constraints that were collected during a concrete program execution are solved. To gain additional benefits from partial verification results, we would like to guide the dynamic symbolic execution toward unverified program executions.

For instance, assume that the current concrete execution explored the then-branch of the outermost conditional in the program from Figure 2.7. Since all assertions on that execution have been explored under conditions for which they have been (soundly) verified by a static analyzer, one can consider the corresponding test case to be *redundant*; i.e., it does not explore an *unverified execution*. However, we only realize *after* already generating and executing the corresponding test case. Ideally, we would like to realize as soon as possible if some execution will not keep exploring assertions under conditions for which they were already verified statically. This would allow us to abort and prune such executions early on and guide the subsequent explorations toward unverified executions.

To achieve this, we have developed an efficient static analysis [28, 23] based on abstract interpretation [35] that infers a (sufficient) *may-unverified* condition at each program point that guarantees that all executions from there on will be verified. Those conditions are subsequently used to instrument the program that we generate tests for using dynamic symbolic execution. Figure 2.8 shows this instrumentation for our example from Figure 2.7. The instrumentation consists of additional partially-justified `assume` statements. As before, these will be picked up by the dynamic symbolic execution. However, at the same time they will abort and prune verified executions. For instance, on line 5 the inferred may-unverified condition is `!true` and the instrumentation tells us that the execution can be aborted unless it holds. Since the condition is false and will therefore never hold here, this allows us to skip the exploration of method `ReviewTransfer` entirely. Similarly, on line 14 the inferred may-unverified condition is `!(a0 && o0 && a1 && o1)`. The corresponding instrumentation tells us that the execution can be aborted unless one of the explicit assumptions is violated. If one such execution is aborted the instrumentation and the corresponding additional constraints instruct the dynamic symbolic execution to try generating new inputs that satisfy the may-unverified condition. In our example, this will prevent the dynamic symbolic execution from exploring the then-branch of the innermost conditional, which turns out not to be feasible unless all explicit assumptions hold. At the same time the assertion on line 21 will only be explored if one of the explicit assumptions is violated, which will subsequently lead to a failure.

In our example, the additional instrumentation will help Pex in generating test cases for both the error due to aliasing and the one due to arithmetic overflow. This has to do with the fact that Pex explores two cases for the else-branch of the innermost conditional due to the use of the short-circuiting Boolean operator: the case where `!(balance < 500)` holds and the case where `balance < 500 && !(balance < rcvr.balance)` holds. The constraints for the latter case will force Pex to detect the error due to aliasing, while for the former case the constraint solver simply happens to pick inputs that reveal the error due to arithmetic overflow. However, note that the additional instrumentation makes it impossible for Pex—

```
public void Transfer(Account rcvr, int amount) {
  requires rcvr != null;

  if (amount <= 0 || 50000 < amount || balance < amount) {
    assume false provided true;
    ReviewTransfer(rcvr, amount);
  } else {
    assumed NoOverflow_sub(balance, amount) as o0;
    assumed NoAliasing(rcvr, this) as a0;
    balance -= amount;
    assume rcvr != null provided a0 && o0;
    assumed NoOverflow_add(rcvr.balance, amount) as o1;
    assumed NoAliasing(this, rcvr) as a1;
    assume false provided a0 && o0 && a1 && o1;
    rcvr.balance += amount;
    assume rcvr != null provided a0 && o0 && a1 && o1;
    if (balance < 500 && balance < rcvr.balance) {
      SuggestLoanFrom(rcvr);
    }
    assume rcvr != null provided a0 && o0 && a1 && o1;
    assert old(rcvr.balance) < rcvr.balance verified a0 && o0 && a1 && o1;
  }
}
```

Figure 2.8: Example from Figure 2.7 with instrumentation (in light gray boxes) derived by computing may-unverified conditions.

which merely tries to achieve full coverage of the method—to cover the former case without detecting one of the errors.

One can think of this additional static analysis and the corresponding instrumentation either as an extension of the test case generation tool or as an additional static analyzer that infers additional verification results based on Clousot's partial verification results (as in the first stage of our tool chain from Figure 2.6). In fact, this instrumentation is not only useful for test case generation tools, but other static analyzers (e.g., static symbolic execution tools or deductive verifiers) may also benefit from it to avoid unnecessary work and to guide the verification effort.

The may-unverified instrumentation is conservative since we do not want to abort or prune executions that *may* be unverified. For instance, if there was an additional unverified assertion at the end of method `Transfer` all executions may be unverified. Therefore, the may-unverified condition would be `true` for all previous program points in the control flow and, consequently, no instrumentation would be added. However, to find bugs more quickly it may be beneficial to heuristically prefer executions that are *definitely* unverified.

For this purpose we have developed a second efficient static analysis that infers a *must-*

unverified condition for each program point. This condition guarantees that (1) all executions from that point onward lead to an assertion and (2) none of those assertions will be explored under conditions for which it has been verified before.

Those must-unverified conditions are never weaker than the may-unverified conditions. Therefore, if we were to use the same instrumentation as for the may-unverified conditions, we might end up pruning some unverified executions (which may include failing ones). To avoid this, we use them to instrument the program such that the dynamic symbolic execution will *preferably* explore executions for which the condition holds. Intuitively, this instrumentation adds weak constraints to the usual (strong) path constraints that are solved when generating new test cases. The constraint solver tries to satisfy all strong constraints and as many weak constraints as possible. Overall, unlike the may-unverified instrumentation, the must-unverified instrumentation primarily affects the order in which test cases are generated. However, since dynamic symbolic execution usually cannot explore all program paths (e.g., due to input-dependent loops) before reaching an exploration bound (e.g., maximum number of runs/paths), the order is an important factor for making the tool effective in finding bugs quickly.

Both of these techniques can be combined to guide dynamic symbolic execution toward unverified executions. We were able to show [28, 23] that it can lead to: (1) smaller test suites by reducing the number of redundant test cases, (2) test suites that cover more unverified executions, (3) a reduction in testing time, and (4) a reduction in the number of exploration bounds that are encountered (e.g., maximum number of concrete test runs, maximum number of branches per execution path).

2.3.2 Refining Partial Verification Results

We have seen two examples of *sound* static analyses that specifically target partial verification results in Section 2.3.1. Both of them work on a very coarse abstraction of the program to make them efficient and light-weight. However, since partial verification results can be expressed easily using standard programming language constructs, it is possible to apply widely-used inference techniques—such as data flow analysis, abstract interpretation [35], or predicate abstraction [64]—to infer additional invariants about *assumption variables*. This can, for instance, allow us to determine that some assumption variable is always true since the corresponding explicit assumption is always true or even unreachable. Such additional invariants can be seen as refining the existing partial verification results. As before, we can express them as partial verification results, which makes them immediately usable for any

```
1  public void Transfer(Account rcvr, int amount) {
2    requires rcvr != null;
3
4    if (amount <= 0 || 50000 < amount || balance < amount) {
5      ReviewTransfer(rcvr, amount);
6    } else {
7      assumed NoOverflow_sub(balance, amount) as o0;
8      assume false provided !o0;
9      assumed NoAliasing(rcvr, this) as a0;
10     balance -= amount;
11     assume rcvr != null provided a0 && o0;
12     assumed NoOverflow_add(rcvr.balance, amount) as o1;
13     assumed NoAliasing(this, rcvr) as a1;
14     rcvr.balance += amount;
15     assume rcvr != null provided a0 && o0 && a1 && o1;
16     if (balance < 500 && balance < rcvr.balance) {
17       SuggestLoanFrom(rcvr);
18     }
19     assume rcvr != null provided a0 && o0 && a1 && o1;
20     assert old(rcvr.balance) < rcvr.balance verified a0 && o0 && a1 && o1;
21   }
22 }
```

Figure 2.9: Example from Figure 2.7 after running an inference tool. The tool soundly determined that the explicit assumption about overflow always holds for the subtraction on line 10. The corresponding invariant can be expressed using a partially-justified `assume` statement on line 8 (in light gray box).

tool. For instance, for the example from Figure 2.7 such a tool could determine that the explicit assumption about overflow always holds for the subtraction on line 10. The output of such a tool is shown in Figure 2.9. The corresponding invariant is expressed using a partially-justified `assume` statement on line 8. As usual, such a fact could, for instance, tell Pex not bother with generating inputs that violate the corresponding explicit assumption.

Since predicate abstraction tools are designed for inferring invariants over a set of arbitrary predicates, we will describe how one could use the Houdini [55] predicate abstraction technique for inferring the invariant from above. As noted before, other inference tools and techniques could also be used instead.

To make use of the Houdini approach the program first needs to be annotated with candidate invariants. These will be rejected gradually by running a verifier that will report invariants that do not hold. The candidate invariants can be chosen freely, but in our example we decided to check if explicit assumptions always hold after they have been made or never hold after they have been made. Note that there are also more systematic approaches (for instance, using abductive inference [84] or weakest preconditions [98]) for selecting such

```
 1 public void Transfer(Account rcvr, int amount) {
 2   requires rcvr != null;
 3
 4   if (amount <= 0 || 50000 < amount || balance < amount) {
 5     ReviewTransfer(rcvr, amount);
 6   } else {
 7     assumed NoOverflow_sub(balance, amount) as o0;
 8     assert false verified !o0; assert false verified o0;
 9     assumed NoAliasing(rcvr, this) as a0;
10     assert false verified !a0; assert false verified a0;
11     balance -= amount;
12     assume rcvr != null provided true;
13     assumed NoOverflow_add(rcvr.balance, amount) as o1;
14     assert false verified !o1; assert false verified o1;
15     assumed NoAliasing(this, rcvr) as a1;
16     assert false verified !a1; assert false verified a1;
17     rcvr.balance += amount;
18     assume rcvr != null provided true;
19     if (balance < 500 && balance < rcvr.balance) {
20       SuggestLoanFrom(rcvr);
21     }
22     assume rcvr != null provided true;
23     assert old(rcvr.balance) < rcvr.balance verified true;
24   }
25 }
```

Figure 2.10: Example from Figure 2.7 after inserting candidate invariants (in light gray boxes). We use partially-verified assertions to express candidate invariants that state that an assumption variable is either always true or always false after the corresponding `assumed` statement.

candidate invariants. After adding the corresponding candidate invariants we end up with the program shown in Figure 2.10. We express the candidate invariants by inserting separate assertions that check whether an assumption variable is true or false (e.g., on line 8). Note that we marked all implicit or explicit assertions as fully verified to prevent the verifier from complaining that they do not hold (e.g., for the failing assertion on line 23).

Now, the Houdini algorithm will repeatedly run a (sound) verifier to narrow down the candidate invariants we want to keep by rejecting ones that lead to verification errors. Since the Boogie verification engine ships an implementation of the Houdini algorithm and soundly verifies Boogie programs, we encoded our C# program as a Boogie program. Note that, unlike in other verifiers that build on Boogie (e.g., Spec#), we have made use of bit-vectors to model the arithmetic operations *soundly*. By running the Houdini algorithm on the corresponding Boogie program, we were able to determine that only one candidate invariant is an actual invariant: the one about overflows for the subtraction on line 11 from Figure 2.10.

This is the candidate invariant we would keep as described earlier. Note that the Houdini algorithm might also not reject either of the two assertions that are inserted per explicit assumption. In this case, both assertions are kept and any down-stream tool can use this to determine that the corresponding program point is unreachable. Besides keeping the candidate invariants that were not rejected in the program, we can also propagate such results directly to affected partially-justified `assume` statements and partially-verified assertions. For instance, in our `Transfer` method we could replace `a0` with `true` everywhere or we could even eliminate the corresponding explicit assumption entirely to refine the partial verification results.

Note that, this approach can easily be generalized for tools with sources of deliberate unsoundness by including additional explicit assumptions (e.g., due to ignoring arithmetic overflow) when expressing the candidate invariants.

2.3.3 Inferring Sufficient Preconditions to Justify Explicit Assumptions

In the previous section, we have seen how inference tools can be used for inferring invariants about assumption variables. In contrast, if explicit assumptions are not guaranteed to hold it might make sense to infer a sufficient precondition under which the explicit assumptions will hold. For one, such preconditions may actually be in line with the user's design intentions and will turn some partially-justified `assume` statements or partially-verified assertions into fully-verified ones. Even if the user does not agree with the suggested preconditions (for instance, because it is too strong), they may turn out to be useful for down-stream tools, much like our may-unverified conditions from Section 2.3.1.

There are several tools that infer such conditions using different techniques. For instance, Clousot infers both necessary and sufficient preconditions using abstract interpretation [37] and the Infer tool [20] uses a shape analysis based on bi-abduction [21]. Another technique for logic programs [58] makes use of abductive inference [99]. In loop-free programs or ones with loops that are annotated with loop invariants, such conditions could also be inferred by computing weakest preconditions [45]. While such techniques are designed to infer preconditions that ensure the correctness of the corresponding method, it should be relatively straightforward to adapt them to this use case.

In fact, one can even use the Houdini approach from the previous section provided that one can come up with good candidate invariants. For instance, such candidate invariants may consist of the expression in the corresponding `assumed` statements if they can be evaluated in the pre-state. This can be seen in Figure 2.11. We make use of an auxiliary Boolean variable

```
 1 public void Transfer(Account rcvr, int amount) {
 2   requires rcvr != null;
 3 
 4   bool o0Pre = NoOverflow_sub(balance, amount); bool a0Pre = NoAliasing(rcvr, this);
 5   bool o1Pre = NoOverflow_add(rcvr.balance, amount); bool a1Pre = NoAliasing(this, rcvr);
 6   if (amount <= 0 || 50000 < amount || balance < amount) {
 7     ReviewTransfer(rcvr, amount);
 8   } else {
 9     assumed NoOverflow_sub(balance, amount) as o0;
10     assert !o0Pre verified !o0;
11     assumed NoAliasing(rcvr, this) as a0;
12     assert !a0Pre verified !a0;
13     balance -= amount;
14     assume rcvr != null provided true;
15     assumed NoOverflow_add(rcvr.balance, amount) as o1;
16     assert !o1Pre verified !o1;
17     assumed NoAliasing(this, rcvr) as a1;
18     assert !a1Pre verified !a1;
19     rcvr.balance += amount;
20     assume rcvr != null provided true;
21     if (balance < 500 && balance < rcvr.balance) { SuggestLoanFrom(rcvr); }
22     assume rcvr != null provided true;
23     assert old(rcvr.balance) < rcvr.balance verified true;
24   }
25 }
```

Figure 2.11: Example from Figure 2.7 after inserting candidate invariants for sufficient preconditions (in light gray boxes). We use partially-verified assertions to express candidate invariants that state that an assumption variable is true if the expression in the corresponding `assumed` statement holds in the pre-state.

for each assumption variable to evaluate the value of the expression in the corresponding `assumed` statement in the pre-state (see lines 4–5). This allows us to insert a candidate invariant for each explicit assumption that states that the assumption variable is true if the auxiliary variable is true. We can now use Houdini as described in Section 2.3.2 for checking those candidate invariants from method `Transfer`. As a result we will learn that all candidate invariants are valid and, therefore, the program would be fully verified if we would strengthen the precondition accordingly. Note that, much like the approach from Section 2.3.2, this approach can be generalized to work for tools with sources of deliberate unsoundness.

2.3.4 Expressing Results of Tools that Infer Preconditions

As discussed in the previous section, several approaches exist for inferring sufficient preconditions to ensure the correctness of a program. Unlike static analyzers that report errors for assertions that may fail, these tools return a preconditions that guarantees that all assertions

```
 1 public void Transfer(Account rcvr, int amount) {
 2   requires rcvr != null;
 3
 4   assumed WP as a0;
 5
 6   if (amount <= 0 || 50000 < amount || balance < amount) {
 7     ReviewTransfer(rcvr, amount);
 8   } else {
 9     balance -= amount;
10     assume rcvr != null provided a0;
11     rcvr.balance += amount;
12     assume rcvr != null provided a0;
13     if (balance < 500 && balance < rcvr.balance) {
14       SuggestLoanFrom(rcvr);
15     }
16     assume rcvr != null provided a0;
17     assert old(rcvr.balance) < rcvr.balance verified a0;
18   }
19 }
```

Figure 2.12: Example from Figure 2.7 after expressing results of a tool that infers a sufficient precondition *WP* (in gray boxes). We use an explicit assumption `a0` on line 4 to capture the inferred precondition *WP* and mark all implicit and explicit assertions as verified under `a0`.

will not fail. This precondition can, for instance, be suggested to users. Even if the user decides not to add the suggested precondition to the specification of the method (e.g., because it is too strong), the condition can alternatively be used to express partial verification results for the method.

To this end, the inferred condition can be captured using an explicit assumption at the beginning the corresponding method. Since this condition is sufficient for ensuring the correctness of the method body, we can now mark all assertions as verified under the corresponding explicit assumption. For instance, a sound verifier based on weakest preconditions, would compute a relatively complex (e.g., due to the sound reasoning about integer arithmetic) condition *WP* for method `Transfer` from Figure 2.7. This condition can be suggested to the user or it can be used to express partial verification results as shown in Figure 2.12. As discussed before, these can be used by down-stream program analysis tools (e.g., a test case generation tool) to check if any assertions in the method may fail under the existing precondition. This would work particularly well when used in combination with the may-unverified instrumentation from Section 2.3.1 since the condition would help to prune the search space already at the beginning of method.

A similar approach could be used for tools that infer *necessary* preconditions (for instance, as

```
 1  public void Transfer(Account rcvr, int amount) {
 2    requires rcvr != null;
 3
 4    assumed NP as a0;
 5
 6    if (amount <= 0 || 50000 < amount || balance < amount) {
 7      ReviewTransfer(rcvr, amount);
 8    } else {
 9      balance -= amount;
10      assume rcvr == null provided !a0;
11      rcvr.balance += amount;
12      assume rcvr == null provided !a0;
13      if (balance < 500 && balance < rcvr.balance) {
14        SuggestLoanFrom(rcvr);
15      }
16      assume rcvr == null provided !a0;
17      assume !(old(rcvr.balance) < rcvr.balance) provided !a0;
18      assert old(rcvr.balance) < rcvr.balance;
19    }
20  }
```

Figure 2.13: Example from Figure 2.7 after expressing results of a tool that infers a necessary precondition *NP* (in gray boxes). We use an explicit assumption `a0` on line 4 to capture the inferred precondition *NP*. Additionally, we introduce a partially-justified `assume` statements for every assertion that states that the assertion will fail if the assumption variable `a0` is false.

in Clousot [37]), which ensure that the program will definitely fail if the condition is violated. Again, we could capture this condition using an explicit assumption `a0` at the beginning of the method. However, instead of marking assertions as verified under that explicit assumption, we would introduce a statement `assume !P provided !a0` for every assertion with condition P. Figure 2.13 shows the result of this transformation for a tool that infers the necessary precondition *NP* for method `Transfer`.

Note that, these two approaches can easily be generalized for tools with sources of deliberate unsoundness by including additional explicit assumptions (e.g., due to ignoring arithmetic overflow) when producing the annotations and by not adding annotations for assertions that are ignored or not checked soundly by the tool.

Both approaches are not strictly limited to inferred preconditions, but could also handle inferred assumptions within a program that ensure its correctness. For instance, such assumptions are inferred in a technique for synthesizing circular compositional program proofs [84]. The general idea of this technique is to decompose the verification task into smaller fragments that may turn out to be more easily verified by some tools than others. Each fragment of the program may be analyzed by several tools. If one tool is able to verify the fragment the algorithm proceeds to the next fragment (i.e., all assertions in the fragment are fully verified). If a fragment cannot be verified by any tool abductive inference is used to infer sufficient assumptions under which the fragment is correct (i.e., assertions are partially-verified under those assumptions). While the existing algorithm would now eagerly try to prove those assumptions recursively, one could imagine to alternatively use partial verification results for capturing the fact that the assertions within a fragment are verified under those assumptions—much like in the case of preconditions. This alternative might lend itself for exploring a more lazy variant of the existing algorithm.

2.3.5 Expressing Verification Results based on Counterexamples

Many static analyzers—such as Boogie, Dafny, and Spec#—produce not only errors or warnings, but also counterexamples. These counterexamples can provide more details about when an error may occur; for instance, they can indicate which path leads to the error or which conditions need to hold to trigger a failure. So far, this information was not taken into account when producing the partial verification results, even though it can be very useful to down-stream tools. For instance, a test case generation tool may only need to target a particular path to check if a reported error is spurious.

To illustrate how counterexamples can be used for producing more precise verification results, we will focus on paths that lead to a failure. For simplicity, we assume that a sound tool was run and that it reported all paths that lead to the reported errors. This would, for instance, be the case for the Dafny verifier or for the sound Boogie encoding of method `Transfer` that we used in previous sections.

The general idea is the following: instead of not marking an assertion for which the static analyzer reported an error as verified, we can mark it as verified for all paths except the ones that were reported. This can be achieved by "paving the path with explicit assumptions" for each failing assertion: on the distinct path with program locations from 0 to n, where each location's predecessor in the control flow has more than one successor, we insert a statement `assumed false as` $\mathtt{a}_k$ for each program location k (i.e., on the failing path all those assumption variables will be false when reaching the failing assertion) and mark the failing assertion as verified under a disjunction $\bigvee_{i=0}^{n} \mathtt{a}_i$. That is, the assertion is verified if any of the assumption variables $\mathtt{a}_i$ is true; if so, the program execution did not explore the failing path, where all those assumption variables $\mathtt{a}_i$ are false. If there is more than one failing path to such an assertion, we instead mark it as verified under a conjunction of all

```
 1 public void Transfer(Account rcvr, int amount) {
 2   requires rcvr != null;
 3
 4   if (amount <= 0 || 50000 < amount || balance < amount) {
 5     ReviewTransfer(rcvr, amount);
 6   } else {
 7     assumed false as a0;
 8     balance -= amount;
 9     assume rcvr != null provided true;
10     rcvr.balance += amount;
11     assume rcvr != null provided true;
12     if (balance < 500 && balance < rcvr.balance) {
13       SuggestLoanFrom(rcvr);
14     } else {
15       assumed false as a1;
16     }
17     assume rcvr != null provided true;
18     assert old(rcvr.balance) < rcvr.balance verified a0 || a1;
19   }
20 }
```

Figure 2.14: Example from Figure 2.7 after annotating it with partial verification results based on a counterexample from a sound static analyzer. We introduce explicit assumptions `a0` and `a1` along the failing path and mark the failing assertion on line 18 as verified under the disjunction of those assumption variables (i.e., a non-failing path was taken).

those individual disjunctions.

Figure 2.14 demonstrates this technique on method `Transfer`. When we run the Boogie verifier on the Boogie encoding for that method, we end up with one error for the assertion on line 18 and the path takes the else-branches of both conditionals. We therefore insert the two explicit assumptions on lines 7 and 15 and mark the failing assertion as verified under `a0 || a1`. If we were to subsequently use our may-unverified instrumentation from Section 2.3.1 before running a test case generation tool, we could effectively prune all other paths to check if the reported error is spurious. Tools based on counterexample-guided abstraction refinement (CEGAR) [29]—such as SLAM [2]—use a similar approach by trying to explore the failing path to detect spurious counterexamples.

Note that this approach can be generalized for tools with sources of deliberate unsoundness by including additional explicit assumptions (e.g., due to ignoring arithmetic overflow) when marking the failing assertion as verified.

2.3.6 Expressing Intermediate Results of Static Analysis Runs

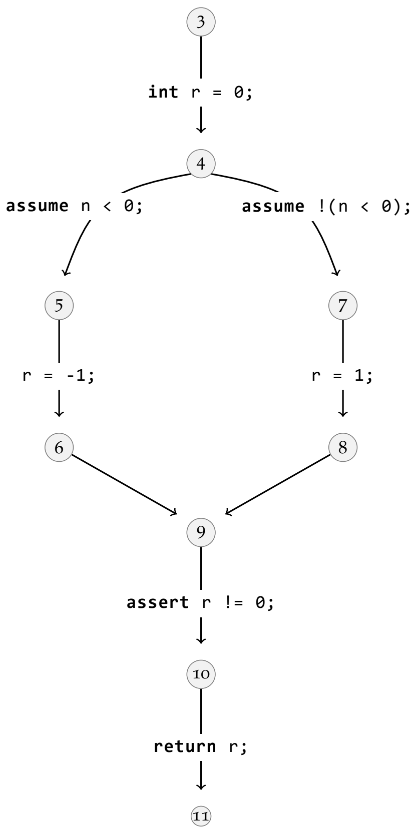
Many static techniques—such as data flow analysis, abstract interpretation [35], predicate abstraction [64], or counterexample-guided abstraction refinement (CEGAR) [29]—are based on a fixed-point computation to converge on its results. However, most often those results are only valid after reaching this fixed-point. Consequently, they are not suitable for being used soundly by other tools before. Since many of these static analysis techniques, such as the polyhedra abstract domain [40], are computationally expensive it may take some time until those results are available and until other tools can use them. Timeouts are a pragmatic way for keeping such analysis tool chains reactive despite this. However, when timeouts happen all intermediate results are simply lost. To avoid this, we show how our technique for expressing partial verification results can be used to soundly share intermediate results during runs of a static analyzer.

As hinted at already, timeouts are not the only situation where this may be useful. At any intermediate point during an analysis down-stream tools may already benefit from results computed so far. A particularly promising situation where this may prove useful is points where the analysis gives up some precision: a necessity for achieving convergence in most practical static analyses. In data flow analysis and abstract interpretation this situation usually occurs for joins in the control flow and, in the case of abstract interpretation, when a widening operation is performed.

In the case of joins, a static analysis may have, for instance, shown that some down-stream assertion holds after executing one branch of a conditional, but may not be able to do so anymore after joining the abstract states of both branches. For instance, consider method `Sign` from Figure 2.15 that returns the sign of its input. A static analysis based on the interval abstract domain [35], which tracks the bounds of integer variables, would use a join operation for determining the state after the conditional (at program location 9). In this case, the result ($[-1, 1]$, where -1 is the lower and 1 the upper bound) strictly over-approximates the concrete set of possible values for variable `r` ($\{-1, 1\}$), which does not include the value 0.

This loss of precision happens because of the join operation and it will prevent us from proving the assertion on line 9, even though it holds. However, at an intermediate state (middle column in the table from Figure 2.15) right before applying the join operation the abstract value of variable `r` on line 9 is still $[-1, -1]$. It was propagated there after the then-branch had been analyzed and could be used to show that the assertion holds assuming that the then-branch is executed. However, since it under-approximates the concrete set of possible values, we cannot simply share this knowledge before reaching the fixed-point. Note that

```
1  int Sign(int n)
2  {
3    int r = 0;
4    if (n < 0) {
5      r = -1;
6    } else {
7      r = 1;
8    }
9    assert r != 0;
10   return r;
11 }
```



Loc.	Interval state for r	
	Intermediate	Final
3	$\perp$	$\perp$
4	$[0, 0]$	$[0, 0]$
5	$[0, 0]$	$[0, 0]$
6	$[-1, -1]$	$[-1, -1]$
7	$[0, 0]$	$[0, 0]$
8	$[1, 1]$	$[1, 1]$
9	$[-1, -1]$	$[-1, 1]$
10	$[-1, -1]$	$[-1, 1]$
11	$[-1, -1]$	$[-1, 1]$

Figure 2.15: Example that demonstrates loss of precision after performing a join operation on program location 9. In the middle, we see the corresponding control flow graph (program locations are depicted as nodes and edges represent control flow with optional program instructions) and to the right we can see the corresponding abstract interval state for variable `r` at an intermediate state and the final one.

a fixed-point computation is also used in programs without loops, although a statement is analyzed at most once in this case.

To express this intermediate knowledge using partial verification results we consult the internal work queue that is used by the fixed-point computation of the analysis to keep track of program locations that still need to be (re-)analyzed. Note that the fixed-point computation terminates once this queue is empty. For these reasons, the intermediate results for a program location in this work queue are not necessarily sound. However, the intermediate results for a program location that is *not* in the work queue are sound provided that the intermediate results of all program locations in the work queue are sound. We can exploit this by capturing the current intermediate state for all program points in the work queue

```
 1 public int Sign(int n)
 2 {
 3   int r = 0;
 4   assume 0 <= r && r <= 0 provided a0;
 5   if (n < 0) {
 6     assume 0 <= r && r <= 0 provided a0;
 7     r = -1;
 8     assume -1 <= r && r <= -1 provided a0;
 9   } else {
10     assume 0 <= r && r <= 0 provided a0;
11     r = 1;
12     assume 1 <= r && r <= 1 provided a0;
13   }
14   assumed -1 <= r && r <= -1 as a0;
15   assume -1 <= r && r <= -1 provided a0;
16   assert r != 0;
17   assume -1 <= r && r <= -1 provided a0;
18   return r;
19 }
```

Figure 2.16: Annotated program after expressing the intermediate results before performing the join operation for program location 9 from Figure 2.15. We introduce an explicit assumption at that program location (on line 14) and introduce partially-justified `assume` statements for other program locations to express intermediate results that are sound under that explicit assumption.

logically as an explicit assumption. Note that this step requires a function that maps an abstract state to a predicate of the assertion language. For instance, in abstract interpretation the concretization function of an abstract domain could be used for this purpose. This will allow us to introduce partially-justified `assume` statements for other program points to logically capture the intermediate results and the condition under which they are sound (i.e., the conjunction of all assumption variables that were introduced for program locations in the work queue).

In our example from Figure 2.15 the work queue before performing the join operation only contains program location 9. We therefore add the following statement at that program location and introduce partially-justified `assume` statements for all other program locations (shown in Figure 2.16):

```
assumed -1 <= r && r <= -1 as a0
```

The same technique can also be used before performing a widening operation. This can be seen in the example from Figure 2.17, where method `Count` iterates over an array `a` to count the number of occurrences of value `e`. At the intermediate state shown, program location 6 is the only location in the work queue. Note that we do not prescribe a specific order in which

```
1  int Count(int e, int[] a)
2  {
3    requires a != null;
4
5    int r = 0; int i = 0;
6    while (i < a.Length) {
7      if (a[i] == e) {
8        r++;
9      }
10     i++;
11   }
12   assert 0 <= r;
13   return r;
14 }
```

Loc.	Interval state for r	
	Intermediate	Final
3	$\bot$	$\bot$
5	$\bot$	$\bot$
6	$[0, 0]$	$\top$
7	$[0, 0]$	$\top$
8	$[0, 0]$	$\top$
9	$[1, 1]$	$\top$
10	$[0, 1]$	$\top$
11	$[0, 1]$	$\top$
12	$[0, 0]$	$\top$
13	$[0, 0]$	$\top$
14	$[0, 0]$	$\top$

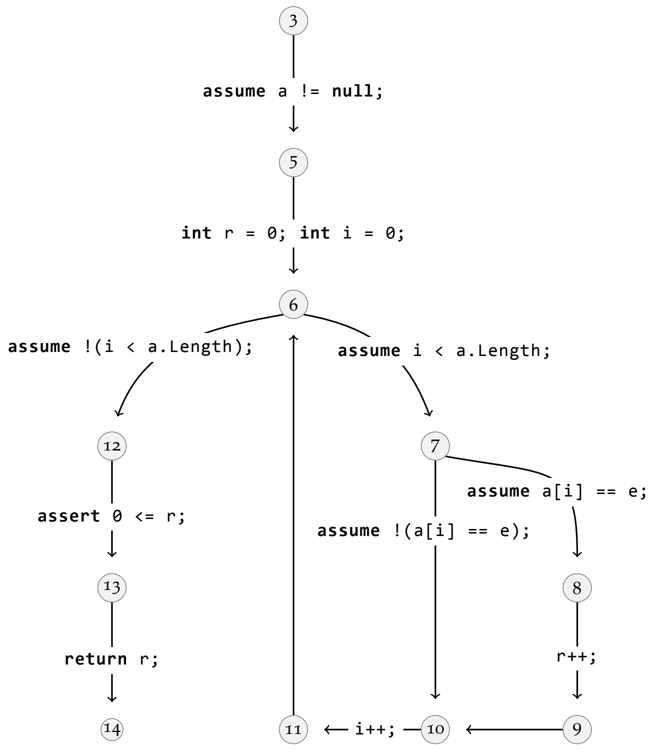


Figure 2.17: Example that demonstrates loss of precision after performing a widening operation on program location 6. To the right, we see the corresponding control flow graph (program locations are depicted as nodes and edges capture control flow and code) and below the source code we can see the corresponding abstract interval state for variable r at an intermediate state and the final one.

program locations are retrieved from the work queue. For instance, in our example, program locations after the loop happened to be analyzed before reaching a fixed-point for program locations within the loop. To continue from the intermediate state, the analyzer would now join the abstract states for the incoming edges of program location 6, which would result in the new abstract state $[0, 1]$. To speed up convergence, it might now decide to perform widening, which would instead result in the new abstract state $\top$ if arithmetic overflow was taken into account. After this step, we would not be able to prove the assertion on line 12 in the final state, even though it holds. Note that the assertion always holds since actually no arithmetic overflow can happen.

To preserve intermediate results despite this loss of precision due to widening, we would

```
1  int Count(int e, int[] a)
2  {
3    requires a != null;
4
5    int r = 0; int i = 0;
6    assume 0 <= r && r <= 0 provided a0;
7  LH:
8    assumed 0 <= r && r <= 0 as a0;
9    if (i < a.Length) {
10     assume 0 <= r && r <= 0 provided a0;
11     if (a[i] == e) {
12       assume 0 <= r && r <= 0 provided a0;
13       r++;
14       assume 1 <= r && r <= 1 provided a0;
15     }
16     assume 0 <= r && r <= 1 provided a0;
17     i++;
18     goto LH;
19   }
20   assume 0 <= r && r <= 0 provided a0;
21   assert 0 <= r;
22   assume 0 <= r && r <= 0 provided a0;
23   return r;
24 }
```

Figure 2.18: Annotated program after expressing the intermediate results before performing the widening operation for program location 6 from Figure 2.17. We introduce an explicit assumption at that program location (on line 8) and introduce partially-justified `assume` statements for other program locations to express intermediate results that are sound under that explicit assumption.

insert an explicit assumption `0 <= r && r <= 0` (based on the old abstract state) for program location 6 and insert partially-justified `assume` statements for other program locations. The annotated program can be seen in Figure 2.18. Note that, to reflect the control flow graph more closely, we rewrote the loop by introducing a label `LH` for the loop header on line 7 and inserting a `goto` statement on line 18 to transfer control back to the loop header after the loop body. Alternatively, one could introduce two separate `assumed` statements immediately after the loop and at the beginning of the loop body.

Note that this approach can be generalized for tools with sources of deliberate unsoundness by including additional explicit assumptions (e.g., due to ignoring arithmetic overflow) when inserting partially-justified `assume` statements to express properties that were inferred.

2.3.7 Dealing with User-provided `assume` Statements more Sensibly

Most programming languages that are designed with verification in mind—such as .NET Code Contracts [51], Dafny [76, 73], or Spec# [5]—support user-provided `assume` statements. Often these are used to suppress errors that a static analyzer reports and that are (possible wrongly) categorized as spurious errors by the user. For instance, a static analyzer might not be able to show that some condition holds after a loop, which is necessary for proving the correctness of subsequent assertions. While these statements are not checked by the static analyzer, they are often checked at runtime just like assertions. We believe that this semantic ambiguity is somewhat dissatisfactory (let alone possibly confusing to users) and we show how user-provided explicit assumptions let us deal with this use case more sensibly.

Instead of adding an `assume` statement to the code, a user would add an `assumed` statement (possibly with a special annotation or a more appropriate keyword to distinguish them from ones that are inserted by tools). A static analyzer is now free to make use (e.g., by assuming the condition) of any such user-provided `assumed` statements as long as they are taken into account when inserting partially-justified `assume` statements and partially-verified assertions.

Now, there is no inherent need for checking those assumed properties anymore where they are made, since any partially-verified assertion can still fail if the assumed property indeed does not hold. Such errors can still be caught by other program analysis tools or at runtime.

If the user prefers to check an assumed property where it is made, an assertion can be added instead. This resolves the ambiguity by separating the two concerns of (possibly unsoundly) assuming the property and checking the property at runtime. Consequently, this setup would allow us not to check partially-justified `assume` statements at runtime since these should only be inserted if the property has independently and soundly been shown to hold (e.g., by a sound static analyzer or one that disclosed any deliberate unsoundness as partial verification results).

This use of explicit assumptions would also blend nicely with other use cases presented in earlier sections. For instance, an inference tool might be able to show that a user-provided explicit assumption always holds (i.e., would be unnecessary if the inference tool was run before any tools that cannot infer the property themselves), never holds (as in Section 2.3.2) or could be added as a precondition (as in Section 2.3.3).

2.4 Related Work

Many automatic static checkers analyzers target mainstream programming languages are deliberately not fully sound to improve performance and reduce the number of false positives and the annotation overhead. We already mentioned some of the sources of deliberate unsoundness made by HAVOC, Spec#, ESC/Java, and Clousot. In addition to those, KeY [7] does not soundly support multi-object invariants, Krakatoa [53] does not handle class invariants and class initialization soundly, and Frama-C [34] uses plug-ins for various analyses with possibly conflicting assumptions. Our technique would allow these tools to collaborate and be effectively complemented by automatic test case generation.

2.4.1 Integration of Static Analyzers

The work most closely related to ours is conditional model checking (CMC) [10, 11], which is an independently developed line of work and combines complementary model checkers to improve performance and state-space coverage. Their approach, like ours, makes the results of static checking precise by tracking which properties have been verified, and under which assumptions. Moreover, they also promote the collaboration of complementary static analyzers and direct the static checking to the properties that have not been soundly verified. A conditional model checker takes as input the program and specification to be verified as well as a condition that describes the states that have already been checked, and it produces another such condition to encode the results of the verification. The focus of CMC is on encoding the typical limitations of model checkers, such as space-out and time-out, but it can also use the condition to express that certain scenarios (e.g., arithmetic overflow) have not been explored. This can be seen as a special case of the use case described in Section 2.3.6. Beyer et al. performed a detailed experimental evaluation that demonstrates the benefits of making assumptions and partial verification results explicit, which is in line with our findings.

Despite these similarities, there are significant technical differences between CMC and our approach. First, if their input condition holds the analyzed program is correct and, thus, a subsequent analyzer can prune parts of the search space by essentially assuming its negation. This is not generally the case after `assumed` statements since some assertions may not be verified by any static analyzer. In that sense, their input condition is conceptually closer to the inferred may-unverified conditions from Section 2.3.1. Second, our representation of partial verification results as program annotations is relatively compact (usually constant overhead in terms of the original program size) and universally understood by essentially all off-the-shelf program analysis tools (as opposed to other conditional model checkers). Third, it is well-suited for annotating programs with inductive properties (e.g., loop invariants) that have been discovered by a static analyzer with or without sources of unsoundness. Fourth, as is common in model checking, CMC is presented as a whole-program analysis, and the resulting condition may contain assumptions about the whole program. For instance, the verification of a method may depend on assumptions made in its callers. In contrast, we have demonstrated how to integrate modular static analyzers, such as Clousot, and deductive verifiers, such as Dafny and Spec#. Fifth, although Beyer et al. mention test case generation as a possible application of CMC, they do not explain how to generate test cases from the conditions. Since these conditions may include non-local assumptions, they might be used to generate *system* tests, whereas the generation of *unit* tests seems challenging. However, test case generation tools based on constraint solving (such as symbolic execution and concolic testing) do not scale well to the large execution paths that occur in system tests. By contrast, we have demonstrated how to use concolic testing to generate unit tests from our local assumptions and verification results.

A common form of tool integration is to support static analyzers with inference tools, such as Houdini [55] for ESC/Java or Daikon [50] for the Java PathFinder [66] tool. Such combinations either assume that the inference is sound and thus, are not suitable for tools with sources of deliberate unsoundness, or they verify every property that has been inferred, which is overly conservative and increases the verification effort. Our technique enables a more effective tool integration by making all sources of deliberate unsoundness explicit.

Work on synthesizing circular compositional program proofs [84] is an interesting approach for combining *sound* tools with different strengthens and weaknesses by decomposing programs into smaller fragments that may turn out to be more easily verified by some tools than others. We believe that our technique for expressing partial verification results could be used to extend this approach to also deal with tools with sources of deliberate unsoundness. Additionally, as hinted at earlier, explicit assumptions might provide an alternative way for capturing assumptions that are inferred using abductive inference.

2.4.2 Integration of Static Analyzers and Testing

Various approaches combine verification and testing mainly to determine whether a static verification error is spurious. Check 'n' Crash [42] is an automated defect detection tool that integrates the ESC/Java static checker with the JCrasher [41] testing tool in order to decide whether errors emitted by the static checker are spurious. Check 'n' Crash was later integrated with Daikon in the DSD-Crasher tool [43]. DyTa [57] integrates the static analyzer Clousot with Pex to reduce the number of spurious errors compared to static verification alone and perform more efficiently compared to dynamic test generation alone. To do so, dynamic symbolic execution is guided toward the errors that were reported by Clousot. However, unlike in our approach, sources of deliberate unsoundness in Clousot are not taken into account. As a result, some defects may not be uncovered due to the way in which the reported errors are used for pruning execution paths. Confirming whether a failing verification attempt refers to a real error is also possible in our technique: The instrumentation phase of the architecture introduces assertions for each property that has not been statically verified (which includes the case of a failing verification attempt). The testing phase then uses these assertions to direct test case generation towards the unproved properties. Eventually, the testing tools might generate either a series of successful test cases that will boost the user's confidence about the correctness of their programs or concrete counterexamples that uncover an error.

YOGI [97], a tool for checking properties of C programs, follows a slightly different approach by using static analysis and dynamic symbolic execution alternately in a way that resembles counterexample-guided abstraction refinement (CEGAR) [29]. Like the previous approaches and the one presented here, YOGI uses testing to prove the existence of bugs. However, it also uses a *sound* static analysis to prove the absence of bugs. SANTE [22] presents another approach to reduce the test effort by performing a sound value analysis and then slicing the program to eliminate paths that do not lead to assertions that resulted in errors during the static analysis. In contrast, our approach is not limited to verification results that were obtained by running a sound static analysis. A recent approach [44] proposes to run a conditional model checker on a program before using a testing tool to test the parts of the state-space have not been explored by the model checker (e.g., due to timeouts). More precisely, they use the output condition, which is produced by the model checker and captures the safe states, to produce a residual program that can be fed to an off-the-shelf testing tool.

EVE [110], the Eiffel verification environment, uses correctness scores to accumulate the results of static and dynamic program analysis tools (e.g., AutoProof [111] and AutoTest [92]) that are run independently. The correctness scores for a static analyzer are decreased if a program contains constructs (e.g., integer arithmetic) that are not handled soundly. Unlike our technique for expressing partial verification results, such correctness scores are coarse-grained and, therefore, less suitable for reducing the verification or testing effort of other tools.

2.5 Summary

We have presented a technique for expressing partial verification results for a wide range of different static analyzers, including ones with sources of deliberate unsoundness. This technique offers a flexible way for annotating programs with results from different tools via two new language constructs. Both constructs can be expressed easily using standard programming language constructs and, therefore, can be supported by a wide range of program analysis tools. Our annotations allow tools to benefit from results that were collected by other tools by directing their effort to verifying properties that have not been verified yet or that have only been verified partially.

As one such use case, we have presented a technique for collaborative static analysis and testing that integrates results from different static analyzers and eventually uses test case generation tools to test properties that have not been fully verified yet. In our approach, the verification results give definite answers about program correctness allowing for the integration of multiple, complementary static analyzers and the generation of more effective unit test suites. This allowed us to identify situations in which this combination finds more errors and proves more properties than static checking alone, testing alone, and combined static checking and testing without our technique. We have also presented several other use cases for partial verification results—including one that guides test case generation tools based on dynamic symbolic execution to cover unverified program executions— that demonstrate their flexibility for capturing verification results from a wide range of program analysis tools. Other use cases are presented in more detail in Chapters 3 and 5.

Chapter 3

An Experimental Evaluation of Deliberate Unsoundness in a Static Analyzer

As we have seen in Chapter 2, many practical static analyzers are not completely sound by design. Their designers often trade soundness in order to increase automation, improve performance, reduce the number of false positives or the annotation overhead, and achieve a modular analysis. By giving up soundness, such static analyzers become precise and efficient in detecting software bugs, but at the cost of making implicit, unsound assumptions about certain program properties. For example, ESC/Java uses bounded loop unrolling to reduce the overhead of writing loop invariants, and Spec# ignores exceptional control flow to speed up verification.

Despite how common such design decisions are, their practical impact on the effectiveness of static analyzers is not well understood. There are various approaches in the literature that study the efficiency and precision of static analyzers by measuring, for instance, their performance and the number of false positives [8]. In this chapter, we focus on a different perspective: we report on the first systematic effort to document and evaluate the sources of deliberate unsoundness in a static analyzer. We present a code instrumentation that reflects the sources of unsoundness in the static analyzer Clousot [52], an abstract interpretation tool for .Net and Code Contracts [51]. This instrumentation adapts the technique from Chapter 2 for making the unsound assumptions of a static analyzer explicit where they occur by automatically inserting annotations into the analyzed code, in the form of `assumed` statements. Most of these assumptions are motivated by Clousot's design goal to analyze programs modularly without imposing an excessive annotation overhead. To evaluate the impact of Clousot's unsound assumptions, we instrumented code from six open-source projects, measured how often the unsound assumptions were violated during executions of the projects' test suites, and determined whether Clousot missed bugs due to unsound assumptions.

The contributions of the work described in this chapter are the following:

- We report on the first systematic effort to document all sources of unsoundness in an industrial-strength static analyzer. We focus on Clousot, a widely-used, commercial static analyzer.
- We present a code instrumentation that reflects the unsoundness in Clousot. Most sources of unsoundness in Clousot are precisely captured by our encoding.
- We perform an experimental evaluation that, for the first time, sheds light on how often the unsound assumptions of a static analyzer are violated in practice and whether they cause the analyzer to miss bugs.

In our experiments we applied our technique to code from six open-source projects. We found that 33 % of the instrumented methods were analyzed soundly. In the remaining methods, Clousot made unsound assumptions, which were violated in 2–26 % of the methods during concrete executions. Three sources of unsoundness were never violated in our evaluation. Manual inspection of those methods with violations showed that no errors were missed due to an unsound assumption, which suggests that Clousot's unsoundness does not compromise its effectiveness. We expect these results to guide users of static analyzers in using them fruitfully, for instance, in deciding how to complement static analysis with testing, and to assist designers of static analyzers in finding good trade-offs. As described in Chapter 2, our results can also facilitate collaboration of static analyzers; new analyzers can now focus on advanced features and rely on existing tools for those properties that are already handled in a sound way.

This chapter is based on a paper that was presented at the *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)* in 2015 [27].

Outline

Section 3.1 explains all sources of unsoundness in Clousot and how we instrument most of them. Section 3.2 gives an overview of our implementation. In Section 3.3, we present and discuss our experimental results. We review related work in Section 3.4 and summarize our results in Section 3.5.

3.1 Unsoundness in Clousot

In this section, we present a complete list of Clousot's sources of deliberate unsoundness and demonstrate how most of these can be expressed as partial verification results. We have elicited Clousot's unsound assumptions during the last two years by studying publications, extensively testing the tool, and having numerous discussions with its designers. Note that a formal proof that Clousot is sound modulo the issues we document here is beyond the scope of this work. In general, for such a proof, one would need to show that the static analyzer is sound for program executions where all checks hold that are deliberately ignored and where all deliberate, unsound assumptions hold.

We make the unsoundness of a static analyzer explicit by automatically annotating the analyzed code with *explicit assumptions* that are expressed using `assumed` statements (see Chapter 2). Note that in the context of this chapter we do not make use of the assumptions identifiers that are usually associated with `assumed` statements. For simplicity, we will therefore use statements of the form `assumed` P, where P is a Boolean expression, and denotes that a static analyzer *unsoundly* assumed property P at this point in the code; that is, the analyzer assumed P without checking that it actually holds.

Each unsound assumption in Clousot applies to a specific syntactic category such as a kind of statement or expression (for instance, because Clousot's abstract transformer does not soundly reflect the semantics of that syntactic category). We say that an explicit assumption *precisely* captures the unsound assumption for a syntactic category if for all elements e of that category and all executions τ of e, Clousot's analysis is sound iff the execution τ does not violate e's explicit assumption. Here, *sound* means that the concrete states of τ lie within the concretization of the corresponding abstract states. We say that an explicit assumption *over-approximates* the unsound assumption if there is an element e and an execution τ of e such that Clousot's analysis is sound, but the execution τ violates e's explicit assumption. Conversely, an explicit assumption *under-approximates* the unsound assumption if there is an element e and an execution τ of e such that Clousot's analysis is *not* sound, but the execution τ *does not* violate e's explicit assumption.

In the following subsections, we present all sources of unsoundness in Clousot divided into four categories: those related to (1) the heap, (2) properties local to a method, (3) static class members, and (4) ones that we do not instrument.

```
class C
{
  bool b;

  invariant !b;

  void M()
  {
    assumed invariant(this, typeof(C));
    b = true;
    N();
    assert !b;
  }

  void N()
  {
    assumed invariant(this, typeof(C));
    assert !b;
  }
}
```

Figure 3.1: Example of explicit assumptions about "invariants at method entries" (**IE**). We use the keywords **`invariant`** and **`assert`** to denote Code Contracts' object invariants and assertions. We introduce `assumed` statements (shown in the gray boxes) at the beginning of methods for which Clousot assumes the invariant of the current object without checking it at call sites .

3.1.1 Heap Properties

Clousot treats the following aspects of the heap unsoundly: object invariants, aliasing, write effects, and method purity.

Object invariants

Code Contracts provide object (or class) invariants [69, 91] to express which objects are considered valid. Clousot checks the invariant of the receiver at the end of a method or constructor, and assumes it in the pre-state of a method execution and after a call. However, the checks are insufficient to justify these assumptions [49]. That is, Clousot makes the following unsound assumptions to facilitate modular checking: *Clousot assumes the invariant of the receiver object in the pre-state of instance methods, without checking it at call sites*; moreover *Clousot assumes the invariant of the receiver after a call to an inherited method on* `this`*, without fully checking it.*

The C# code in Figure 3.1 illustrates the first unsoundness. Method `M` violates the invariant of its receiver before calling `N`. (We use the keywords `invariant` and `assert` to denote Code Contracts' object invariants and assertions.) The gray boxes in the code are discussed later. Clousot assumes the invariant of the receiver in the pre-state of method `N`, which is unsound since it does not check this invariant at call sites of `N`, in particular, before the call to `N` in `M`. Therefore, Clousot emits no warning for the assertion in `N`, although it will not hold when `N` is called from `M`. The fact that there is no warning for the assertion in `M` is a consequence of the same unsoundness. Clousot checks the receiver's invariant in the post-state of method `N`; this check succeeds because of the same unsound assumption in `N`'s pre-state. The check in the post-state justifies assuming the invariant after the call.

We capture this unsoundness by introducing an `assumed` statement at the beginning of each instance method in classes that declare or inherit object invariants. As shown in the gray boxes in the code, these explicit assumptions use a predicate `invariant`(o, t), which holds iff object o satisfies the object invariants defined in class t in conjunction with all invariants inherited from t's super-classes. Here, type t is the type of the class in which the method is defined; the corresponding type object is retrieved with the `typeof` expression in C#. We label this kind of explicit assumption as "invariants at method entries" (**IE**). We will refer to such labels in our experimental evaluation.

This explicit assumption captures the first unsoundness precisely because any method execution in which the explicit assumption is violated (that is, where the receiver's invariant does not hold in the pre-state), will be analyzed with an unsound abstraction of the initial state (unless Clousot's abstract domains do not reflect the invariant anyway, which we ignore here). This does not necessarily mean that Clousot misses errors because the unsoundness might be irrelevant for the checks performed on the method body. Conversely, if the abstraction of the initial state is unsound because the receiver's invariant is violated, the explicit assumption will be false. Note that there are programs for which this will never happen; some explicit assumptions may always hold in these programs (and still be precise according to our definition).

The code in Figure 3.2 illustrates the second source of unsoundness. Method `M` of the sub-class calls the inherited method `N` of the super-class on the current receiver, and `N` violates the invariant declared in the sub-class. However, since Clousot's analysis is modular, `Sub`'s invariant is not considered when analyzing `Super` and, therefore, Clousot does not detect this invariant violation. Nevertheless, Clousot assumes the invariant of `this` after the call to `N` in `M`, which is unsound. As a result, no warnings are emitted.

We precisely capture this unsoundness by introducing an `assumed` statement after each call to an inherited method on the current receiver in classes that declare or inherit object invari-

```
class Super
{
  bool b;

  void N() { b = true; }
}

class Sub : Super
{
  invariant !b;

  Sub() { b = false; }

  void M()
  {
    N();
    assumed invariant(this, typeof(Sub));
    assert !b;
  }
}
```

Figure 3.2: Example of explicit assumptions about "invariants at call sites" (**IC**). We introduce `assumed` statements (shown in the gray box) at call sites where Clousot assumes the invariant of the current class (here, `Sub`), which might not hold if the receiver is of a different type (e.g., of type `Super`).

ants. The explicit assumption states that the object invariant of `this` holds for the enclosing class (here, `Sub`) and its super-classes. We label this kind of explicit assumption as "invariants at call sites" (**IC**).

Aliasing

To avoid the overhead of a precise heap analysis, *Clousot ignores certain side-effects due to aliasing*. For operations with side-effects, such as field updates, Clousot unsoundly assumes that heap locations not explicitly aliased in the code are non-aliasing and, thus, not affected.

As an example of this unsoundness, consider method `M` from Figure 3.3. (We use the keyword **`requires`** to denote preconditions.) Clousot assumes that the object `a` is not modified during the update of field `f` on line 7, although `a` and `b` might point to the same object in some calls to `M`. As a result, no warning is emitted for the assertion on line 8.

Clousot abstracts the heap by a *heap-graph*, which maintains equalities about access paths. The nodes of the heap-graph denote symbolic values, which represent concrete values, such as object references and primitive values. An edge of the heap-graph denotes how the sym-

```
1 void M(Cell a, Cell b)
2 {
3   requires a != null && b != null;
4
5   if (a.f) {
6     assumed b == null || !object.ReferenceEquals(b, a);
7     b.f = false;
8     assert a.f;
9     ...
```

Figure 3.3: Example of explicit assumptions about "aliasing" (**A**). We introduce an `assumed` statement (shown in the gray box) to capture the fact that Clousot ignores the possible side-effect for the field update on line 7 in case objects `a` and `b` are aliases.

bolic value of the target node is retrieved from the symbolic value of the source node, for instance, by dereferencing a field or calling a pure method. (Programmers may declare a method as pure to indicate that it makes no visible state changes.) All access paths in the heap-graph are rooted in a local variable or a method parameter. When two access paths lead to the same symbolic value, they represent the same concrete value, that is, must be aliases. However, when two access paths lead to distinct symbolic values, they may represent the same or different concrete values, that is, may or may not be aliases. Nevertheless, Clousot unsoundly assumes in this case that updating the heap through one path will not affect values read through the other.

We precisely capture this unsoundness by introducing an `assumed` statement before every side-effecting operation that unsoundly affects the values in the heap-graph, that is, when the side-effect is reflected only on some symbolic values, although other symbolic values may represent the same heap locations. Specifically, for each field, property, or array update (side-effects via calls are discussed below), we determine the set of symbolic values that are distinct from the symbolic value for the receiver r of the update, but may be aliases of r. This set is computed based on the heap-graph in the pre-state of the update and on type information. For each element s of this set, our explicit assumption has a conjunct expressing that the concrete values represented by r and s (and given by the access paths leading to the symbolic values) are non-aliasing.

In our example, Clousot's heap abstraction uses distinct symbolic values for the objects `a` and `b` in the initial heap-graph (see left-most heap graph in Figure 3.4). Thus, for the field update on line 7, r represents `b` and the set of possible aliases consists of `a` (see middle heap graph in Figure 3.4). Hence, the explicit assumption expresses that `a` and `b` are not aliases. This allows Clousot to perform a strong-update of field `f` for the symbolic value pointed to

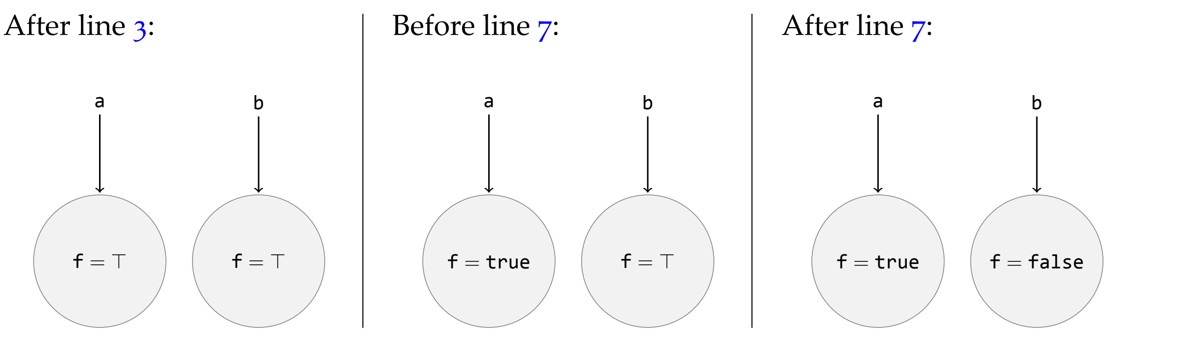


Figure 3.4: Simplified heap graphs for different program points when analyzing the code from Figure 3.3. Nodes represent symbolic values and edges represent access-paths. This shows that the update on line 7 only affects the symbolic value of object `b`, even though object `a` could also be affected in case they are aliases.

by `b`, thereby changing its abstract value from the unknown value ⊤ to the value `false` (see right-most heap graph in Figure 3.4). In particular, it does not change the abstract value of field `f` for the symbolic value pointed to by `a`. Note that we call `ReferenceEquals` since the `==` operator may be overloaded in C#. We ensure that all explicit assumptions are *well-defined*, that is, insusceptible to runtime errors, such as null dereferences in access paths. We label this kind of explicit assumption as "aliasing" (**A**).

Write effects

To avoid a non-modular, inter-procedural analysis or having to provide explicit write effect specifications, *Clousot uses unsound heuristics to determine the set of heap locations that are modified by a method call*. Clousot then assumes that all other heap locations are not modified. This assumption is unsound since the heuristics in general may not include all heap locations that are modified by a call.

The code in Figure 3.5 illustrates this unsoundness. Clousot assumes that the call to method `N` in `M` modifies only the fields of the receiver object, and leaves the elements of the array unchanged. As a result, it does not emit a warning for the assertion. Note that this unsoundness is caused by Clousot's heuristics for write effects, regardless of whether `a` and `b` are aliases.

We capture this unsoundness by introducing an `assumed` statement after each call, stating that all heap locations in the heap-graph that Clousot assumes to remain unmodified by the call are indeed not modified. This is achieved by comparing all symbolic values in the heap-graph before and after the call and using their access paths to retrieve the concrete values they represent. The explicit assumption has a conjunct for each unmodified concrete object

```
class C
{
  int[] a;

  void M()
  {
    var b = new int[1];
    a = b;
    N();
    assumed b == null || !writtenObjects().Contains(b));
    assert b[0] == 0;
  }

  void N()
  {
    if (a != null && 0 < a.Length) { a[0] = 1; }
  }
}
```

Figure 3.5: Example of explicit assumptions about "write effects" (**W**). We introduce an `assumed` statement (shown in the gray box) to capture the fact that Clousot ignores that object `b` may be modified during the call to method `N`.

reference stating that it is not contained in the actual write effect of the method for the last call.

To obtain the actual write effect, we instrument the program to provide the function `writtenObjects`, which returns the set of objects that were modified by the most recently executed call (including any objects that were modified indirectly through method calls). We label this kind of explicit assumption as "write effects" (**W**). Note that this explicit assumption subsumes the aliasing unsoundness for calls because it covers all objects Clousot assumes to be left unchanged by a call, no matter whether this assumption is caused by ignoring certain aliasing situations or by the unsound heuristics for write effects. In method `M` above, `writtenObjects` returns the set consisting of array `a` and, since `a` and `b` refer to the same array, the explicit assumption is violated at runtime.

How precisely we capture this unsoundness depends on the definition of function `writtenObjects`. If the function returns an over- or under-approximation of the set of heap locations modified by the most recently executed call then our assumptions over- or under-approximate Clousot's unsoundness, respectively. In our implementation, `writtenObjects` is precise for methods that we instrument, but under-approximates the write effects of library methods (see Section 3.2).

Purity

Users may explicitly annotate a method with the Code Contracts attribute `Pure` to express that the method makes no visible state changes. To avoid the overhead of a purity analysis, *Clousot assumes that all methods annotated with the* `Pure` *attribute as well as all property getters indeed make no visible state changes.* (We will refer to property getters and methods annotated with `Pure` as "pure methods".)

Moreover, Clousot uses unsound heuristics to determine which heap locations affect the result of a pure method, that is, the method's *read effect*. *Clousot then assumes that all pure methods deterministically return the same value when called in states that are equivalent with respect to their assumed read effects.*

We capture the first unsoundness with the explicit assumptions about write effects described above. After each call to a pure method, we introduce an `assumed` statement stating that all heap locations in the heap-graph remained unmodified.

Method `M` from Figure 3.6 illustrates the second unsoundness. Clousot assumes that both calls to the pure method `Random` in `M` deterministically return the same value, and no warning is emitted.

Method `N` on the right illustrates another aspect of this unsoundness. Clousot assumes that the result of the pure method `First` depends only on the state of its receiver, but not on the state of array `a`. Therefore, no warning is emitted about the assertion in `N` even though `a[0]` is modified after the first call to `First`.

Clousot's heap-graph maintains information about which values may be retrieved by calling a pure method. For instance, after the first call to `Random` in `M`, the heap-graph maintains an equality of `r` and a call to `Random`. This information becomes unsound if (1) the pure method is not deterministic, (2) an object is modified, but Clousot unsoundly assumes that the pure method does not depend on that object, or (3) an object is modified, but Clousot does not reflect the modification correctly in the heap-graph.

The latter case is covered by the explicit assumptions for aliasing and write effects. We capture the former two cases as follows: (1) We generate an explicit assumption after each call to a pure method stating that the method still yields the value stored in the heap-graph. This assumption under-approximates Clousot's unsoundness due to non-determinism since even a non-deterministic method might return the same result several times in a row. In our example, it will fail for method `Random`. (2) Whenever the heap-graph retains a value for a pure method call across a statement that may modify the heap, we generate an explicit assump-

```
class C {
  void M() {
    var r = Random();
    assumed r == Random();
    assert r == Random();
    assumed r == Random();
  }
  [Pure]
  int Random() { return (new object()).GetHashCode(); }
}

class D {
  int[] a;
  void N() {
    requires a != null && 0 < a.Length;
    var v = First();
    assumed v == First();
    a[0] = v + 1;
    assumed v == First();
    assert v == First();
    assumed v == First();
  }
  [Pure] int First() {
    requires a != null && 0 < a.Length;
    return a[0];
  }
}
```

Figure 3.6: Example of explicit assumptions about “purity” (**P**). We add `assumed` statements (shown in the gray boxes) to reflect that Clousot assumes that the call to the pure method `Random` deterministically returns the same value and that the return value of the call to the pure method `First` does not depend on the array `a`.

tion stating that the method still yields the value stored in the heap-graph. This assumption precisely captures the case that Clousot may assume a too small read effect, as for method `First`. We label these explicit assumptions as “purity” (**P**).

3.1.2 Method-Local Properties

We now present the sources of unsoundness in Clousot that are related to properties local to a method. We divide them into two categories, integral-type arithmetic and exceptional control flow.

```
int a = ...;
assumed (long)(a + 1) == (long)a + (long)1;
a = a + 1;
assert int.MinValue < a;
```

Figure 3.7: Example of explicit assumptions about "overflows" (**O**). We insert an `assumed` statement (shown in the gray box) to capture the fact that Clousot ignores that the addition may lead to an arithmetic overflow.

Integral-type arithmetic

To reduce the number of false positives, *Clousot ignores overflow in integral-type arithmetic operations and conversions*. That is, Clousot treats bounded integral-type expressions as unbounded (except for `checked` expressions, which raise an exception when an overflow occurs).

The code in Figure 3.7 illustrates the unsoundness for operations. Although the assertion fails when an overflow occurs, no warning is emitted.

We precisely capture this unsoundness by introducing an `assumed` statement before each bounded arithmetic operation that might overflow (and is not `checked`) stating that the operation returns the same value as its unbounded counterpart. We encode this unbounded counterpart by performing the operation on operands with types for which no overflow will occur, for instance, `long` instead of `int` as in the example above, or arbitrarily large integers (`BigInteger`) instead of `long`. We label this kind of explicit assumption as "overflows" (**O**).

The code in Figure 3.8 illustrates the unsoundness for conversions. Even though the assertion fails due to an overflow that occurs when converting `a` to a `short` integer, Clousot does not emit any warnings.

We precisely capture this unsoundness by introducing an `assumed` statement for each conversion of an integral type to a type with smaller value range stating that the value before

```
int a = int.MaxValue;
assumed a == (short)a;
short b = (short)a;
assert (int)b == int.MaxValue;
```

Figure 3.8: Example of explicit assumptions about "conversions" (**CO**). We insert an `assumed` statement (shown in the gray box) to account for the fact that Clousot ignores that the conversion from a value of type `int` to a value of type `short` by lead to an overflow.

```
try {
  throw new Exception();
}
catch (Exception) {
  assumed false;
  assert false;
}
```

Figure 3.9: Example of explicit assumptions about "catch blocks" (**C**). We insert an `assumed` statement (shown in the gray box) to express the fact that Clousot ignores the `catch` block during the analysis.

the conversion is equal to the value after the conversion, as shown above. We label this kind of explicit assumption as "conversions" (**CO**).

Exceptional control flow

Exceptions add a large number of control-flow transitions and, thus, complicate static analysis. To avoid losing efficiency and precision, many static analyzers ignore exceptional control flow. *Clousot ignores* `catch` *blocks and assumes that the code in a* `finally` *block is executed only after a non-exceptional exit point of the corresponding* `try` *block has been reached.*

The code in Figure 3.9 illustrates the unsoundness for `catch` blocks. Since Clousot ignores the `catch` block, no warning is emitted about the assertion.

We precisely capture this unsoundness by introducing an `assumed` statement at the beginning of each `catch` block stating that the block is unreachable, as shown in the code above. We label this kind of explicit assumption as "catch blocks" (**C**).

The code in Figure 3.10 illustrates the unsoundness for `finally` blocks. Since Clousot assumes that the `finally` block is entered only when the `try` block executes normally, no warning is emitted about the assertion. (We use * to denote an arbitrary Boolean condition.)

We precisely capture this unsoundness by introducing an `assumed` statement at the beginning of each `finally` block stating that the block is entered only when the `try` block terminates normally. This is expressed by introducing a fresh Boolean variable for each `try` block, which is initially false and set to true at all non-exceptional exit points of the `try` block, as shown in the code. The `assumed` statement then states that this variable is true. We label this kind of explicit assumption as "finally blocks" (**F**).

```
bool b = false;
bool $noException$ = false;
try {
  if (*) {
    throw new Exception();
  }
  b = true;
  $noException$ = true;
} finally {
  assumed $noException$;
  assert b;
}
```

Figure 3.10: Example of explicit assumptions about "finally blocks" (**F**). We insert additional instrumentation and an `assumed` statement (shown in the gray boxes) to capture that Clousot assumes that the `finally` block is executed only after a non-exceptional exit from the corresponding `try` block.

3.1.3 Static Class Members

Here, we describe the sources of unsoundness for static fields and main methods.

Static fields

To avoid the complications of class initialization [24, 23] and to reduce the annotation overhead and the number of false positives, *Clousot assumes that static fields of reference types contain non-null values*.

As an example of this unsoundness, consider the code in Figure 3.11, for which no warnings are emitted.

We precisely capture this unsoundness by introducing an `assumed` statement for each read

```
static int[] a;

void M()
{
  assumed a != null;
  assert a != null;
}
```

Figure 3.11: Example of explicit assumptions about "static fields" (**S**). We insert an `assumed` statement (shown in the gray box) to account for the fact that Clousot assumes that the static field `a` stores a non-null reference.

```
void M()
{
  Main(null);
}

public static void Main(string[] args)
{
  assumed args != null && (forall arg in args • arg != null);
  assert args != null;
  assert args.Length == 0 || args[0] != null;
}
```

Figure 3.12: Example of explicit assumptions about "main methods" (**M**). We insert an `assumed` statement (shown in the gray box) to express that Clousot assumes that both the string array `args` of a main method and its elements are non-null.

access to a static field of reference type stating that the field is non-null, as shown in the code. We label this kind of explicit assumption as "static fields" (**S**).

Main methods

When a main method is invoked by the runtime system, the array of strings that is passed to the method and the array elements are never null. To relieve its users from providing preconditions for main methods, *Clousot assumes that the string array passed to a main method and its elements are non-null for all invocations of the method.*

As an example, consider the code in Figure 3.12. Although method `M` calls `Main` with a null argument, no warning is emitted about the assertions in `Main`.

We precisely capture this unsoundness by introducing an `assumed` statement at the beginning of each main method stating that the parameter array and its elements are non-null, as shown in the code above. (We use the **`forall`** keyword to denote Code Contracts' universal quantifiers.) We label this kind of explicit assumption as "main methods" (**M**).

3.1.4 Uninstrumented Unsoundness

In the rest of this section, we give an overview of the remaining sources of unsoundness in Clousot, which we do not instrument:

- *Concurrency*: Clousot does not reason about concurrency and assumes that the analyzed code runs without interference from other threads.

- *Reflection*: Clousot assumes that the analyzed method does not use reflection.
- *Unmanaged code*: Clousot checks memory safety for unmanaged code, but does not consider its effects on the analyzed method.
- *Static initialization*: Clousot assumes that the analyzed code runs without interference from a static initializer.
- *Iterators*: Clousot does not analyze iterator methods (C#'s `yield` statements).
- *Library contracts*: Clousot assumes that the contracts provided for libraries, such as the .NET API, are correct.
- *Floating-point numbers*: Under certain circumstances, Clousot assumes that operations on floating-point numbers are associative and distributive.

A very coarse way of capturing the first five sources of unsoundness would be to introduce an `assumed false` statement at each program point that starts a thread, invokes reflection, or contains unmanaged code, as well as in each static initializer and for each `yield` statement. Such an instrumentation would grossly over-approximate Clousot's unsound assumptions (for instance, many static initializers do not interfere with the execution of the analyzed method). However, a more precise instrumentation is complicated and would require explicit assumptions for most statements, for instance, to detect data races.

Incorrect library contracts could be detected by introducing an explicit assumption for the postcondition of each call into the library. We omit these assumptions because they are orthogonal to the design of the static analyzer. Finally, we do not instrument the unsoundness about floating-point numbers because we were not able to precisely determine where the assumptions occur.

Note that we do not consider Clousot's inference of method contracts and object invariants here. In the presence of inference, an unsound assumption in a method m might affect not only the analysis of m but also of methods whose analysis assumes properties inferred from m, in particular, m's postcondition and the object invariant of the class containing m.

One solution is to introduce an explicit assumption whenever Clousot assumes a postcondition or invariant that was inferred unsoundly; one can then determine easily which methods have been analyzed soundly by inspecting the instrumented method body. Another solution is to rely on the existing instrumentation, which is sufficient to reveal unsound inference during the execution of the program. If the postcondition of a method or constructor m was inferred unsoundly, we detect an assumption violation when executing a call to m, and analogously if m violates an inferred invariant.

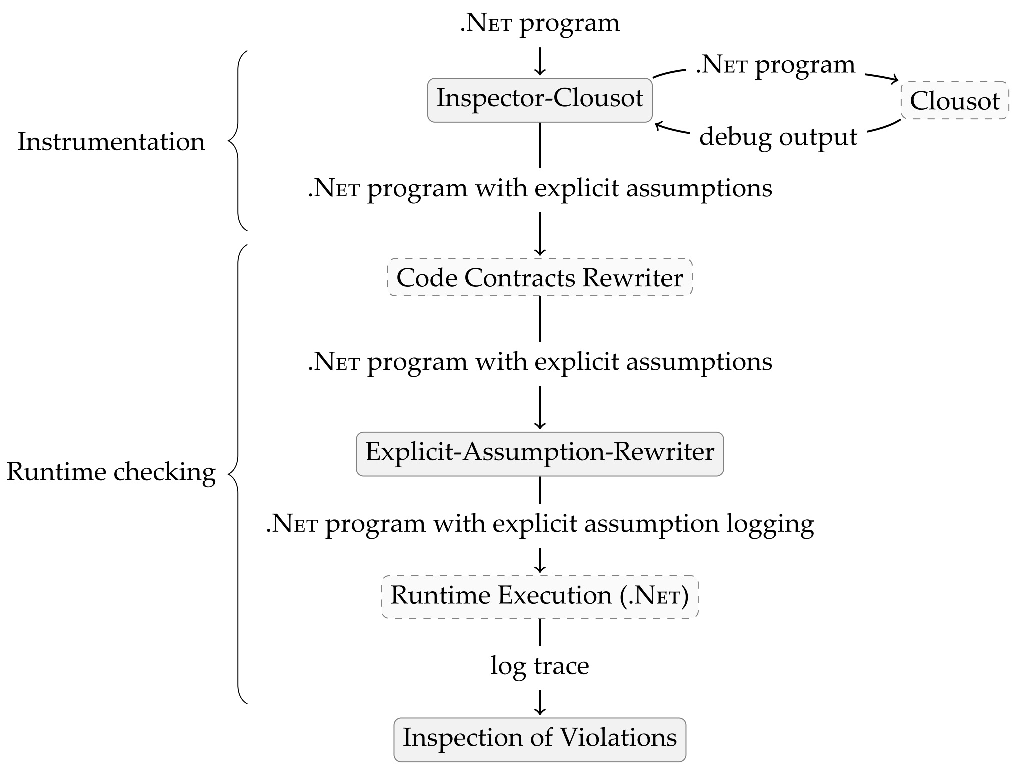


Figure 3.13: Overview of the workflow for evaluating deliberate unsoundness in Clousot. Components and activities are depicted by boxes and edges represent information that is exchanged between them. Off-the-shelf components are depicted as lighter gray boxes with a dashed border. We first invoke Inspector-Clousot, which calls Clousot, and uses its output to produces a new .NET program with explicit assumptions. We use the Explicit-Assumption-Rewriter to set up the explicit assumption logging and run the program to produce a log trace, which can be inspected for violations of explicit assumptions.

3.2 Implementation

To evaluate whether Clousot's sources of unsoundness are violated in practice, we have implemented a tool chain that instruments code with explicit assumptions and checks them at runtime. Figure 3.13 provides an overview of this tool chain and the overall workflow for evaluating unsoundness in Clousot.

Instrumentation

The instrumentation stage runs Clousot on a given .Net program, which contains code and optionally specifications expressed in Code Contracts, and instruments the sources of unsoundness of the tool as described in the previous section. For this purpose, we have implemented *Inspector-Clousot*, a wrapper around Clousot that uses the debug output emitted during the analysis to instrument the program (at the binary level).

Runtime checking

In the runtime checking stage, we first run the existing Code Contracts binary rewriter to transform Code Contracts specifications into runtime checks. For example, method postconditions, which are specified at the beginning of a method body, are transformed into runtime checks occurring at every return point of the method.

We subsequently run a second rewriter, called *Explicit-Assumption-Rewriter*, that transforms all `assumed` statements of the instrumented program into logging operations. More specifically, this rewriter replaces each explicit assumption `assumed` P by an operation that logs the program point of the `assumed` statement, which kind of unsoundness it expresses, and whether the assumed property P is violated. If P contains method calls, we do not further log assumed properties in the callees.

The Explicit-Assumption-Rewriter also instruments each method to compute its set of written objects by keeping track of all object allocations and updates to instance fields and array elements. The set of written objects of a method consists of the objects that have been modified but are not newly allocated by the method. The set of written objects for a call to an uninstrumented (library) method is always empty, that is, our instrumentation underapproximates the objects actually modified by such a method.

3.3 Evaluation

In this section, we present our experiments for evaluating whether Clousot's unsound assumptions are violated in practice and whether these violations cause Clousot to miss errors.

Application	Description	CC	Analyzed methods	Methods with violations
BCrypt.Net[1]	Password-hashing library	no	21	1 / 12 (8.3 %)
Boogie[2]	Verification language and engine	yes	299	2 / 119 (1.7 %)
ClueBuddy[3]	GUI application for board game	yes	139	16 / 67 (23.9 %)
Codekicker.BBCode[4]	BBCode-to-HTML translator	no	179	2 / 58 (3.4 %)
DSA[5]	Data structures and algorithms library	no	213	26 / 99 (26.3 %)
Scrabble (for WPF)[6]	GUI application for Scrabble	yes	127	8 / 41 (19.5 %)

Table 3.1: Applications selected for our evaluation of deliberate unsoundness in Clousot. The first two columns describe the C# applications. The third column indicates whether the applications contain Code Contracts. The fourth column shows the number of analyzed methods per project. The fifth column shows how many of the methods with explicit assumptions that were hit at runtime contained assumption violations.

For our experiments, we used code from six open-source C# projects (see Table 3.1) from different application domains. We selected only applications that come with a test suite so that the experiments achieve good code coverage. We chose three applications to contain Code Contracts specifications to evaluate the explicit assumptions about object invariants. We ran our tool chain on at least one substantial DLL from these applications to perform the instrumentation described in the previous sections.

For invoking Clousot, we enabled all checks, set the warning level to the maximum, and disabled all inference options. We subsequently ran tests from the test suite of each application and logged which explicit assumptions were hit at runtime and which of those were violated.

Finally, we manually inspected a large number of methods including all methods that produced assumption violations to determine whether Clousot misses any errors because of its unsound assumptions.

[1] `http://bcrypt.codeplex.com`, rev: `d05159e21ce0`
[2] `http://boogie.codeplex.com`, rev: `8da19707fbf9`
[3] `https://github.com/AArnott/ClueBuddy`, rev: `c1b64ae97c01fec249b2212018f589c2d8119b59`
[4] `http://bbcode.codeplex.com`, rev: `80132`
[5] `http://dsa.codeplex.com`, rev: `96133`
[6] `http://wpfscrabble.codeplex.com`, rev: `20226`

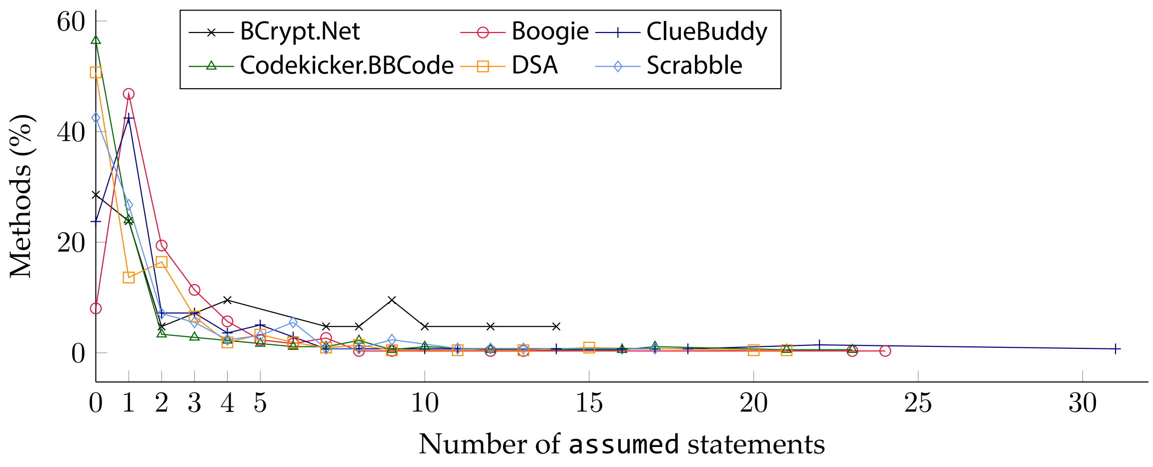


Figure 3.14: The percentage of analyzed methods from each project versus the number of `assumed` statements in the methods.

3.3.1 Experimental Results: Instrumentation

Figure 3.14 presents the percentage of analyzed methods from each project versus the number of `assumed` statements in the methods. An *analyzed* method is checked by Clousot but not necessarily hit at runtime by the test suite of a project. We analyzed a total of 978 methods with Clousot. As shown in the figure, the majority of these methods (860) contain less than 5 `assumed` statements, and a large number of those (326) are soundly checked, that is, do not contain any explicit assumptions. There are only 20 methods with more than 10 `assumed` statements. In these methods, the prevailing sources of unsoundness are "invariants at call sites" (**IC**), "write effects" (**W**), "purity" (**P**), and "overflows" (**O**).

Figure 3.15 shows the average number of bytecode instructions in the analyzed methods versus the number of `assumed` statements in the methods. Notice that most soundly checked methods contain only a small number of instructions. A manual inspection of these methods showed that many of them are setters, getters, or (default) constructors. Our results indicate that methods with more instructions contain a larger number of `assumed` statements.

Figure 3.16 shows Clousot's sources of unsoundness versus the number of `assumed` statements that are introduced in the analyzed methods of each project. The results are dominated by the assumptions that are introduced for each method (**IE**) or for common statements (**IC**, **W**, **P**). The unsound treatment of aliasing (**A**) affects relatively few methods, even though it could be introduced for each field, property, or array update. Assumptions about "main methods" (**M**) were not introduced because there are either no main methods at all (for instance, in libraries) or not in the portions of the code that we analyzed and instrumented.

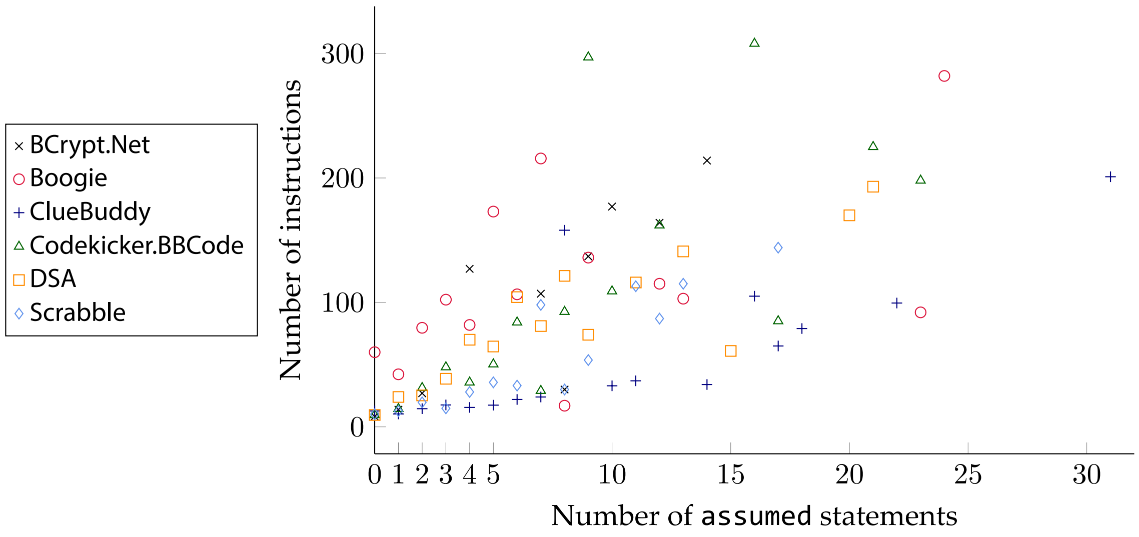


Figure 3.15: The average number of bytecode instructions in the analyzed methods from each project versus the number of `assumed` statements in the methods.

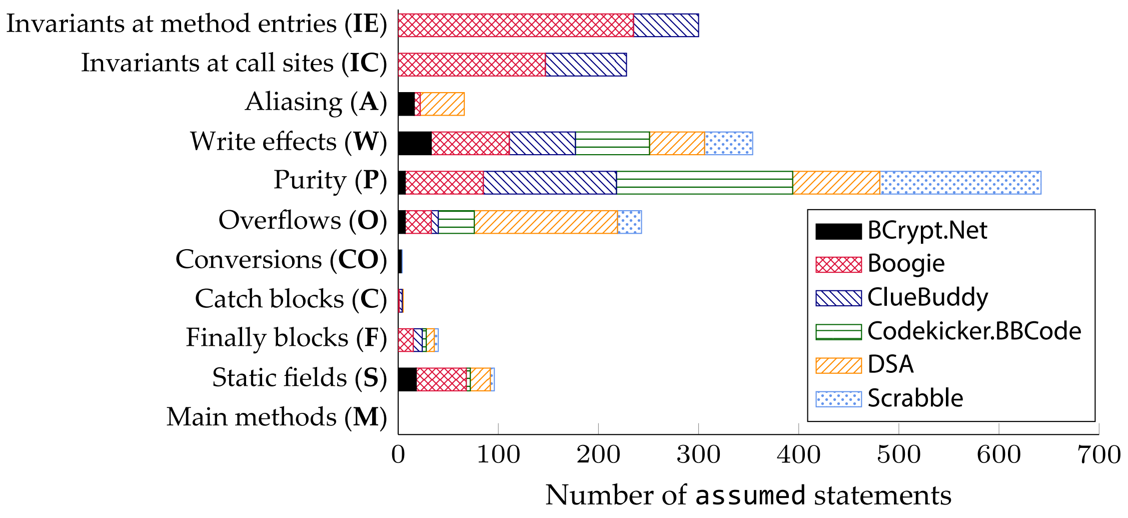


Figure 3.16: Clousot's sources of unsoundness versus the number of `assumed` statements that are introduced in the analyzed methods of each project.

	BCrypt.Net	Boogie	ClueBuddy	Codekicker.BBCode	DSA	Scrabble
IE	-	0/1694124 (0%)	275/27318 (1.01%)	-	-	-
IC	-	0/628448 (0%)	0/9759 (0%)	-	-	-
A	0/25844436 (0%)	0/24771 (0%)	-	-	131/992 (13.21%)	-
W	0/6419169 (0%)	0/372851 (0%)	0/3589 (0%)	82/11577 (0.71%)	0/613 (0%)	25/5011 (0.50%)
P	0/6405279 (0%)	27/108506 (0.02%)	12198/241385 (5.05%)	0/10311 (0%)	0/1008 (0%)	425/21580 (1.97%)
O	102488804/326722626 (31.37%)	0/569258 (0%)	0/547 (0%)	0/1196 (0%)	0/6053 (0%)	0/909 (0%)
CO	0/6633876 (0%)	-	-	-	-	0/2 (0%)
C	-	-	-	-	1/1 (100%)	-
F	-	0/53246 (0%)	0/325 (0%)	0/114 (0%)	0/43 (0%)	0/65 (0%)
S	0/708 (0%)	1/155080 (0%)	-	0/7 (0%)	129/640 (20.16%)	0/15 (0%)
M	-	-	-	-	-	-

IE	:	invariants at method entries	**P**	:	purity	**F**	:	finally blocks
IC	:	invariants at call sites	**O**	:	overflows	**S**	:	static fields
A	:	aliasing	**CO**	:	conversions	**M**	:	main methods
W	:	write effects	**C**	:	catch blocks			

Table 3.2: The number and percentage (rounded to two decimal places) of violated explicit assumptions per application and kind of assumption. These numbers include all executions of a single `assumed` statement. Cells with non-zero values are highlighted; the "-" indicates that no explicit assumptions are hit at runtime.

	BCrypt.Net	Boogie	ClueBuddy	Codekicker.BBCode	DSA	Scrabble
IE	-	0/108 (0%)	7/44 (15.91%)	-	-	-
IC	-	0/60 (0%)	0/59 (0%)	-	-	-
A	0/16 (0%)	0/1 (0%)	-	-	16/46 (34.78%)	-
W	0/30 (0%)	0/32 (0%)	0/43 (0%)	2/61 (3.28%)	0/51 (0%)	1/25 (4.00%)
P	0/7 (0%)	1/40 (2.50%)	10/81 (12.35%)	0/130 (0%)	0/86 (0%)	11/85 (12.94%)
O	4/11 (36.36%)	0/11 (0%)	0/5 (0%)	0/25 (0%)	0/134 (0%)	0/13 (0%)
CO	0/3 (0%)	-	-	-	-	0/1 (0%)
C	-	-	-	-	1/1 (100%)	-
F	-	0/3 (0%)	0/5 (0%)	0/3 (0%)	0/8 (0%)	0/2 (0%)
S	0/18 (0%)	1/31 (3.23%)	-	0/2 (0%)	16/18 (88.88%)	0/2 (0%)
M	-	-	-	-	-	-

IE	:	invariants at method entries	**P**	:	purity	**F**	:	finally blocks
IC	:	invariants at call sites	**O**	:	overflows	**S**	:	static fields
A	:	aliasing	**CO**	:	conversions	**M**	:	main methods
W	:	write effects	**C**	:	catch blocks			

Table 3.3: The number and percentage (rounded to two decimal places) of violated explicit assumptions per application and kind of assumption. These numbers are per occurrence of a single `assumed` statement. Cells with non-zero values are highlighted; the "-" indicates that no explicit assumptions are hit at runtime.

3.3.2 Experimental Results: Runtime Checking

The experimental results for the instrumentation alone provide very limited insight into the impact of Clousot's unsoundness. For instance, while some explicit assumptions reflect details of the analysis (such as **A** and **W**, which are based on Clousot's heap-graph), others merely indicate the existence of a syntactic element (for instance, we generate one assumption of kind **C** per catch-block). Moreover, some explicit assumptions are not violated in any concrete program execution; for instance, the assumptions of kind **M** always hold if a program does not call a main method. To better understand the impact of Clousot's unsound assumptions, we measure how often the generated explicit assumptions are violated during concrete program executions.

Table 3.2 shows the number and percentage of violated explicit assumptions per application and kind of assumption. These numbers include *all executions* of a single `assumed` statement. That is, different executions of the same `assumed` statement in different method invocations or loop iterations are counted separately. Table 3.3 shows the corresponding numbers when counting only per *occurrence* of an `assumed` statement rather than per execution. For example, in BCrypt.Net, the assumption violations shown in Table 3.2 occur in only 4 `assumed` statements (see Table 3.3), which are all in the body of the same loop.

3.3.3 Manual Inspection

We manually inspected a large number of explicit assumptions, including all violated assumptions, and made the following observations:

- "Invariants at method entries" (**IE**): Only Boogie and ClueBuddy contain invariant specifications, and all violations are found in ClueBuddy. These violations are all caused by constructors that call property setters in their body. The object invariants are, therefore, violated on entry to the setters since the constructors have not yet established the invariants. Objects that escape from their constructors are a well-known problem; a possible solution is to annotate methods that may operate on partially-initialized objects and, thus, must not assume their invariants [108].
- "Invariants at call sites" (**IC**): These assumptions are never violated because in all of our applications, sub-classes do not strengthen the object invariants of their super-classes such that calls to inherited methods could violate them.
- "Aliasing" (**A**): These assumptions are violated only in DSA. All violations occur in nine methods of two classes implementing singly and doubly-linked lists. For example, one violation occurs in method `AddAfter` when expressions `this.Tail`, `this.Head`, and the node to be added are aliased. The small number of these violations suggests that there is only a limited practical need for performing a sound, but expensive heap analysis. However, an analyzer could optionally allow users to run a sound heap analysis, for instance, for methods with violations of "aliasing" assumptions.
- "Write effects" (**W**): Table 3.3 shows that these assumptions are hardly ever violated. By inspecting assumptions of this kind that are not violated, we confirmed that the write effects assumed by Clousot are usually conservative.
- "Purity" (**P**): Most of these assumptions are violated for pure methods that return newly-allocated objects, that is, for non-deterministic methods. In applications without Code Contracts, these assumptions are introduced only in property getters, but are never violated.
- "Overflows" (**O**): These assumptions are violated only in BCrypt.Net. All violations occur in an `unchecked` block, which suppresses overflow exceptions. This indicates that, in this application, overflows are actually expected to occur or even intended.
- "Conversions" (**CO**): These assumptions are never violated. Our manual inspection showed that the value ranges of the converted expressions are sufficiently small such that no overflow may occur.

- "Catch blocks" (**C**): Only one assumption of this kind was introduced in a method that removes a value from an AVL tree in application `DSA`. An auxiliary method throws an exception when the AVL tree is empty. Catching this exception violates the assumption. This violation could be avoided by using an out-parameter instead of an exception to signal that the tree was empty.
- "Finally blocks" (**F**): Our instrumentation introduced only 39 assumptions about "finally blocks". The majority of these `finally` blocks are added by the compiler to desugar `foreach` statements. If the body of the `foreach` statement does not throw an exception, these assumptions are not violated.
- "Static fields" (**S**): The violations of these assumptions are, in some cases, due to static fields being lazily initialized, that is, being assigned non-null values after having first been read. Supporting lazy initialization via a language construct, such as Scala's "lazy val" declarations, could help avoid such violations. In other cases, the values of static fields are passed as arguments to library methods, which are designed to handle null arguments.

To investigate how Code Contracts specifications influence the number of explicit assumptions and their violations, we also ran the test suites of the three applications with Code Contracts after having removed these specifications. As expected, there were no assumptions about "invariants at methods entries" (**IE**) and "invariants at call sites" (**IC**). Moreover, the number of assumptions about "purity" (**P**) and their violations were slightly reduced, as the user-specified `Pure` attributes were no longer taken into account.

Missed errors

The violation of an explicit assumption does not necessarily mean that Clousot misses errors since the resulting unsoundness may be irrelevant for the subsequent checks. To determine whether the assumption violations detected in our experiments might lead to missed errors, we manually inspected the containing methods of all 70 violations (computed from Table 3.3).

We did not find any runtime errors or assertion violations that Clousot missed due to its unsound assumptions. With the exception of a few cases, it was fairly straightforward to determine whether an assumption violation could conceal an error. For instance, violations of explicit assumptions about "purity" (**P**) are harmless when there is only a single call to the pure method. The same holds for explicit assumptions about "aliasing" (**A**) when the updated field, property, or array element is not accessed after the update.

The fact that we did not find any missed errors due to assumption violations possibly indicates that providing slightly weaker soundness guarantees in certain situations in favor of performance, precision, and low annotation overhead does not compromise Clousot's effectiveness; its unsound assumptions are not problematic in the code and executions we investigated.

3.3.4 Threats to Validity

We identified the following threats to the validity of our experiments:

- *Instrumentation*: It is possible that we missed some of Clousot's unsound assumptions. Since we elicited the assumptions very diligently, it seems unlikely that we overlooked any major sources of unsoundness. There are several sources of unsoundness that we identified, but do not capture (see Section 3.1.4). For most of these sources, a syntactic check suffices to determine whether a program might be affected. Moreover, even though our instrumentation captures most of Clousot's unsound assumptions precisely, it under-approximates the unsound treatment of write effects for calls to uninstrumented (library) methods and of non-deterministic pure methods. As a result, it is possible that Clousot's analysis of a method is unsound even though all runtime checks for explicit assumptions pass (this is very unlikely for non-deterministic pure methods).

- *Runtime checking*: We measured assumption violations in executions of the projects' test suites. There were no failing tests, that is, any errors detected by the test suites have been fixed. This explains in part why we did not find any errors missed by Clousot. However, in our manual inspection of the violated assumptions, we checked the entire method, that is, all execution paths of the method for all its input states, not just the code covered by the test suite. Thus, we could have detected errors that the tests missed.

- *Project selection and sample size*: The projects in our experiments were chosen from different application domains. All projects were required to include a test suite. We selected projects with and without Code Contracts. Since Clousot analyzes each method modularly, we were able to pick those DLLs that have the most comprehensive test suites. We ran Clousot on 978 methods; `assumed` statements were added in 652 methods, 396 out of which were hit during the execution of the projects' test suites. Therefore, we believe that our projects are representative for a large class of C# code bases.

3.4 Related Work

To the best of our knowledge, there is no existing work on experimentally evaluating sources of deliberate unsoundness in static analyzers.

There are, however, several approaches for ensuring soundness of static analyzers and verifiers, ranging from manual proofs [93], over interactive and automatic proofs [14, 9], to less formal techniques, such as "smoke checking" in the Boogie verification engine [4].

Many static analyzers compromise soundness to improve on other qualities such as precision or efficiency (see Cousot and Cousot [39] for an overview), and there is existing work on evaluating these other qualities of analyzers in practice. For instance, Sridharan and Fink [105] evaluate the efficiency of Andersen's pointer analysis, and Liang et al. [85] evaluate the precision of different heap abstractions. Recently, a proof system has been proposed for identifying a class of programs for which an given abstract domain is complete [59] and, therefore, does not report spurious errors. We show that such evaluations are also possible for the unsoundness in static analyzers, and propose a practical approach for doing so.

Our explicit assumptions could be used to express semantic environment conditions inferred from a base program, as in VMV [89]; a new version of the program could then be instrumented with these inferred conditions (in the form of assumptions) to reduce the number of warnings reported by Clousot. Moreover, our technique could be applied in "probabilistic static analyzers" [86] to determine the probabilities of their judgments about analyzed code. Specifically, one could estimate the probability that an unsound assumption holds (or is violated) based on its value along a number of concrete executions.

Finally, we refer the reader to `http://soundiness.org` for the "soundiness" movement in static program analysis, which brings forward the ubiquity of unsoundness in static analyzers, draws a distinction between analyzers with specific, well-defined soundness trade-offs and tools that are not concerned with soundness at all, and issues a call to the research community to clearly identify the nature and extent of unsoundness in static analyzers [87].

3.5 Summary

In this chapter, we have presented the first systematic effort to document and evaluate the sources of deliberate unsoundness in a widely-used, commercial static analyzer. Our technique is general and applicable to any analyzer whose unsoundness is expressible using a code instrumentation. In particular, we have explained how to derive the instrumentation by concretizing relevant portions of the abstract state (in our case, the heap-graph). We believe that this approach generalizes to a large class of assumptions made by static analyzers.

Our work can help designers of static analyzers in finding good trade-offs. We encourage them to document all compromises of soundness and to motivate them empirically. Such a documentation facilitates tool integration since other static analyzers or test case generators could be applied to compensate for the explicit assumptions. Information about violated assumptions (for instance, collected during testing) could also be valuable in identifying methods that require special attention during testing and code reviews. Finally, our results could be used to derive programming guidelines and language designs that mitigate the unsoundness of a static analyzer.

Chapter 4

The Dafny Integrated Development Environment

Our technique for expressing partial verification results provides a flexible and expressive basis for integrating program analysis tools by exchanging annotated programs. However, eventually the verification results need to be presented *and* explained to the user. Nowadays, this task is often orchestrated by an integrated development environment (IDE) that incorporates one or more program analysis tools. In recent years, this task has become more important and more challenging due to the fact that program verifiers and interactive theorem provers have become more powerful and, thus, more suitable for verifying large programs or proofs. As a consequence, a wider audience of non-experts are interested in using such tools for improving software quality and correctness. This has demonstrated the need for improving the user experience of these tools to increase productivity and to make them more accessible to non-experts.

Such tools usually integrate three major subsystems. At the foundation of the tool lies the logic it uses, for example a Hoare-style program logic or a logic centered around type theory. On top of the logic sits some mechanism for automation, such as a set of cooperating decision procedures or some proof search strategies (e.g., programmable tactics). The logic and automation subsystems affect how a user interacts with the verification system, as is directly evident in the tool's input language. The third subsystem is the tool's IDE, which in a variety of ways tries to reduce the effort required by the user to understand and make use of the proof system and its partial verification results.

In this chapter, we present the IDE for Dafny [76, 73]—a programming language, verifier, and proof assistant. The IDE is an extension of Microsoft Visual Studio (VS) and incorporates both new and existing techniques to improve the overall user experience. We thereby push the state-of-the-art closer towards a verification environment that can provide verification feedback as the user types and can present more helpful information about the program or failed verification attempts in a demand-driven and unobtrusive way. This allows the user to quickly gain insights about the program and to understand the cause of partial verification results. Such insight and understanding is crucial for supporting users in developing fully-verified programs. In the following, we present several important aspects of the IDE that each contribute the overall goal of providing a user experience that goes beyond what has been done in previous IDEs (for Dafny and other verification systems).

continuous processing The IDE runs the program verifier in the background, thus providing *design-time feedback* with every keystroke. The user does not need to reach for a "Verify now" button.

Design-time feedback is common in many tools. For example, the spell checker in Microsoft Word is always on in this way. Anyone who remembers from the 1980s having to invoke the spell checker explicitly knows what a difference this can make in how we think about the interaction with the tool; the burden of having to go through separate spelling sessions was transformed into the interaction process that is hardly noticeable. Parsing and type checking in many programming-language IDEs is done this way, enabling completion and other kinds of IntelliSense context-sensitive editing and documentation assistance. The Spec# verifier was the first to integrate design-time feedback for a verifier [4]. The jEdit editor for Isabelle [113] also provides continuous processing in the background by running both a proof search and the Nitpick [13] checker which searches for counterexamples to the proof goal.

non-linear editing The text buffer can be edited anywhere, just like in usual programming-language editors. Any change in the buffer will cause the verifier to reconsider proof obligations anywhere in the buffer. (Since the Dafny language is insensitive to the order of declarations, the proof obligations that have to be reconsidered can occur both earlier and later in the buffer.)

Although such non-linear editing seems obvious, it is worth noting that it is in stark contrast to common theorem prover IDEs like ProofGeneral[1] and CoqIde[2], where the user manually moves a *high water mark* in the buffer—anything preceding this mark in the buffer has been processed by the system and is locked down to prevent editing, and anything following the mark has not been processed and can be freely edited.

[1] http://proofgeneral.inf.ed.ac.uk
[2] http://coq.inria.fr

dependency analysis and caching The Dafny IDE caches verification results as well as computed dependencies of what is being verified. Before starting a new verification task, the system first consults the cache. This feature makes the tool more responsive and reduces the user's wait times.

Our users have found this to be the most useful of our features for making the interaction between user and system more effective. It is also what makes continuous processing desirable for large files. When a user gets stuck during a verification attempt, a typical response is to try many little input variations that might explain or remove the obstacle at hand. It is during these times that the user needs the tool the most, so supporting fluid interactions at this time is of utmost importance.

There has been a lot of work on caching, modifying, and replaying proofs for interactive proof assistants. For proofs performed by SMT solvers, Grigore and Moskal have worked on these things in the context of ESC/Java [65]. The static analyzer Clousot [52] makes use of caching to retrieve the results of previous runs of its cloud-based analysis service [3]. The SPARK tool set was also extended with light-weight support for caching of verification results [18].

multi-threading The Dafny IDE makes more aggressive and informed use of the support for concurrency in todays hardware. The number of concurrent threads used is adjusted dynamically, depending on what the verification tasks at hand are able to saturate.

Although conceptually an obvious thing to do, the Dafny tool chain previously lacked the features to run separate verification tasks in parallel. The use of multiple threads is especially noticeable when a file is just opened in the editor, since caches are cold at that time and everything needs to be verified.

The Isabelle/jEdit editor [113, 106] comes with support for multi-threading, which is motivated by the fact that it also supports non-linear editing and therefore offers more opportunities to parallelize verification tasks. The SPARK 2014 toolset [48] also supports multi-threading, both in its translation from SPARK into the intermediate verification language Why3 and in the Why3 processing itself.

showing information Commonly, a verification system can supply various associated declarations automatically. For example, common induction schemes may be constructed by default, some types and loop invariants may be inferred, and syntactic shorthands can reduce clutter in the program text. Sometimes, a user may find it necessary to inspect this information. The Dafny IDE attempts to make this information available via *hover text*—when the user hovers the mouse cursor over a part of the program text, say, an identifier, any additional information about that identifier is displayed. This makes the information easily accessible to users, but is at the same time not cluttering up the view of the program text.

Note that in console-based interactive tools, such as ACL2 [71], the unobtrusive nature of information in hover text is difficult to achieve. Such a tool has to either provide a set of commands that can be used to query information gathered by the tool or optimistically spill out a stream of information to the console window in the off-chance that a user wants to see some part of that information.

An important consequence of making additional information easily accessible to the user is that it gives the verification system greater freedom in what can be computed automatically. Users no longer need to fully understand the creative and elaborate schemes employed to compute this information, because whatever is computed can be viewed by the user, if needed.

This feature is also common in programming-language IDEs, where inferred types or fully qualified identifier names are displayed as hover text. The Dafny IDE takes this a step further, by showing information such as default *termination measures*, specifications of implicit methods (such as those generated for *iterators*), which calls are classified as *co-recursive*, and code inherited by Dafny's " ... " construct from a refined module.

integrated debugging Verification error messages can have a lot of associated information, some of which can be useful to users. Previously, the Dafny IDE would highlight, directly in the IDE editor, the error trace leading to a reported error. The same holds for other tools, such as SPARK 2014. To get information about the possible values of variables for the reported error, a Dafny user can use the Boogie Verification Debugger (BVD) [74], which presents this information in a format akin to that provided in modern source-level debuggers. We have done a deep integration of BVD into the Dafny IDE.

Previously, BVD was accessible for Dafny only as a standalone tool, which meant the user manually had to correlate the source lines reported by BVD with the text buffer containing the program in the IDE. The program verifier VCC [30] integrates BVD into its Visual Studio IDE. The Dafny IDE now goes further, for example, by letting the user select which program state to inspect by clicking in the program text itself. This allows the user to focus its attention on one particular verification error and it allows the IDE to provide much more targeted and relevant information to the user. It also uses hover text to present values of variables in the selected state. OpenJML [31, 32] also presents error information in this way, letting users inspect values of any subexpression and letting the source code location of the expressions hovered over determine which execution state is used to look up the value to be displayed.

A alternative approach for debugging failed verification attempt has been developed for Spec# [95]. Unlike BVD, it uses information from counterexamples to create *unit tests* which simulate the verification semantics and can be executed in a off-the-shelf debugger. By actually executing those unit tests genuine errors can be confirmed and spurious errors can be detected. Boogaloo [100] pursues a similar goal. The main difference is that it does not require any output (e.g., a counterexample) from the verifier and, instead, uses symbolic execution to produce tests for the Boogie program.

An orthogonal approach for diagnosing errors [46] makes use of abductive inference to compute queries that are answered by the user. The user's answers can help the verifier in deciding if an error is spurious or genuine.

diagnosing timeouts As most other verifiers, Dafny may fail to terminate within a given time limit. In this case, the user is usually left with very little information about what might have led to the timeout. Most often, the user can avoid such timeouts by making minor changes to the program, such as providing additional intermediate assertions to help the verifier. However, coming up with such changes to the program is difficult without useful information from the verifier. To provide such information, the Dafny IDE uses a technique for diagnosing timeouts. In particular, it can tell a user if the timeout can be avoided by increasing the default time limit slightly or which assertions the verifier is struggling with.

As an alternative to running Dafny in Visual Studio, Dafny can also be run from within a web browser[3] and from the command line. However, the bulk of the features we mention in this chapter are available only in the Visual Studio IDE extension. Dafny, including its IDE, is available as open source[4].

This chapter is based on a paper that was presented at the *Workshop on Formal Integrated Development Environment (F-IDE)* in 2014 [82].

[3]http://rise4fun.com/dafny
[4]http://dafny.codeplex.com

Outline

Section 4.1 provides an overview of the architecture of the Dafny IDE and its underlying components. We give an overview of a caching technique for avoiding unnecessary re-verification work as programs are edited by users of the IDE in Section 4.2. In Section 4.3, we demonstrate a novel feature for reducing the amount of information about the program that is displayed to the user and for showing more relevant information on-demand. In Section 4.4, we present another novel feature that makes the error reporting more focused. We demonstrate a technique for diagnosing timeouts that happen during verification in Section 4.5. In Section 4.6 we evaluate the effect of parallelizing the verification effort using multiple solver instances and our technique for diagnosing timeouts. We have presented related work above and we summarize our results in Section 4.7.

4.1 Tool Architecture with Multiple Solver Instances

Before presenting the new tool architecture, we will give an overview of the underlying components and the tool architecture that was used in the past (see Figure 4.1); it is similar to the architecture of other verification tools that are built on top of the Boogie verification engine [4], such as Spec# [5] and VCC [30]. As the user is editing the program, the VS extension continuously sends snapshots of the program to the underlying Dafny verifier, which encodes the correctness proof obligations as a translation into Boogie. Boogie is an intermediate language [81] for program verification (similar to Why3 [54]). Boogie programs typically consist of several primitive constructs (e.g., axioms, variables, procedures) that are used to formalize programs in a higher-level language, such as Dafny.

For example, a Dafny *method* is translated to several Boogie constructs: (1) a *procedure (declaration)* that captures the specification of the method, (2) a *procedure implementation* that captures the method body and checks that it adheres to the method specification, and (3) a second procedure implementation that captures the well-formedness conditions for the method specification [75]. As another example, a Dafny *function* is translated to a corresponding Boogie *function* and a procedure implementation that captures the function's well-formedness conditions. Boogie functions are given meaning by *axioms*, but to simplify our presentation, we omit some details of the translation of Dafny functions.

This architecture gives rise to a pleasant and highly responsive user interaction for small programs, but does not scale well to larger programs that consist of many methods and functions. Since the requests to the underlying solver can easily be parallelized, we have

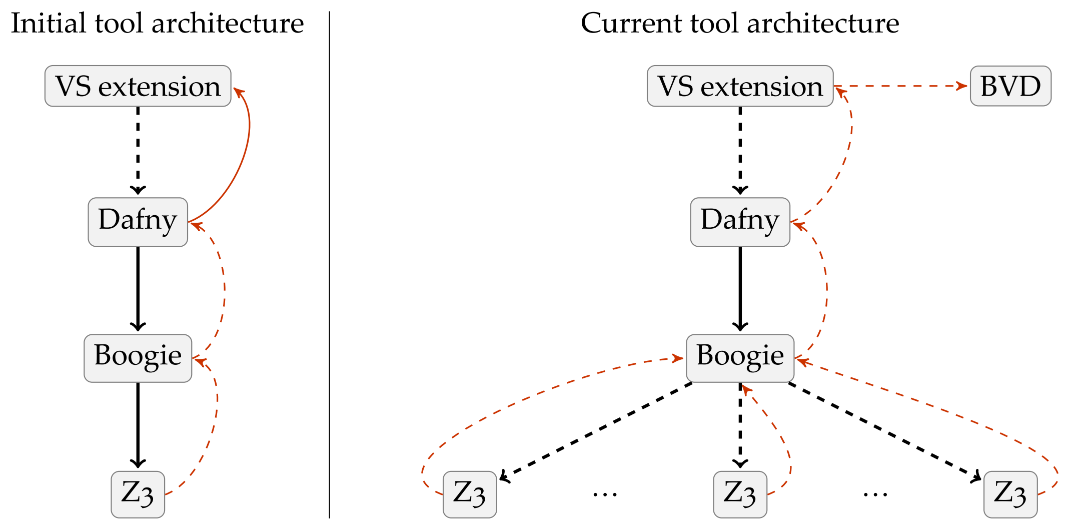
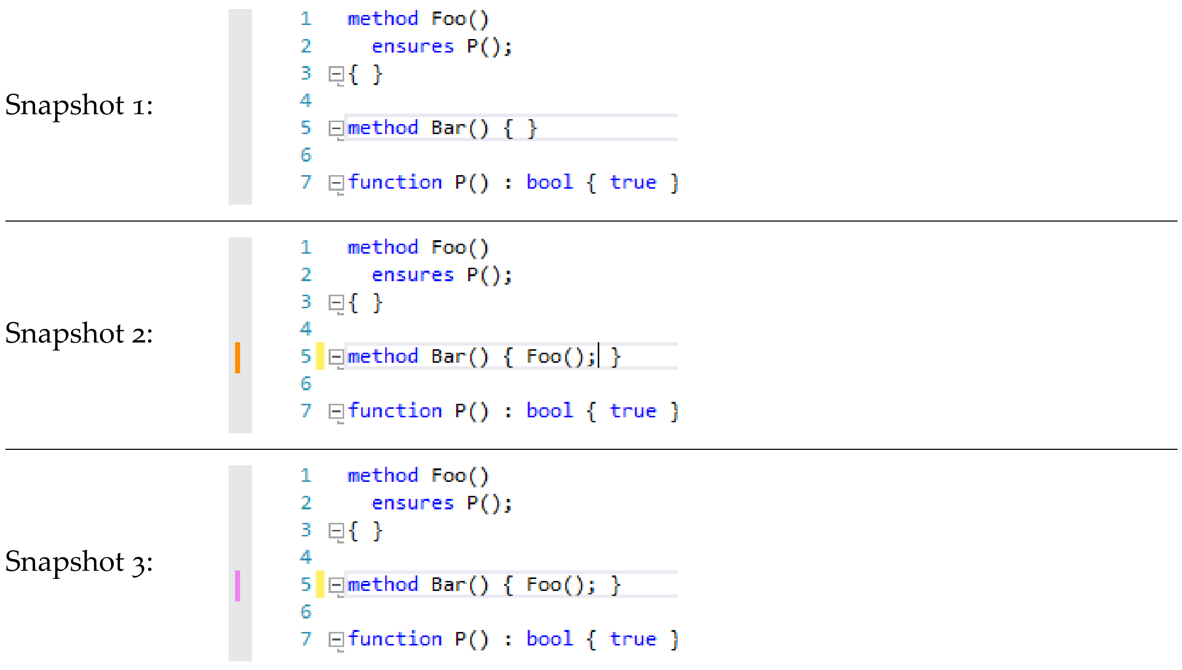


Figure 4.1: Comparison of initial and current tool architecture. Arrows indicate data that is passed from one component to another, where dashed arrows indicate that data is transferred asynchronously. Less thick, red arrows indicate error information (including counterexamples for BVD in the current architecture) that is returned.

extended the Boogie verification engine to make use of separate tasks for verifying Boogie implementations in parallel. Each task may discharge its verification conditions using one or more solver instances that are managed in a dynamically allocated pool of solvers. To take full advantage of this architectural change, we made the propagation of verification errors to the user fully asynchronous (see dashed arrows in Figure 4.1). This lets error messages show up as soon as the corresponding verification condition has been processed by the solver. (Previously, Boogie only made use of multi-threading in one place, namely in its mode for verification-condition splitting [79]. We have preserved that functionality and integrated it into the new task-based architecture.)

The Visual Studio extension for Dafny gets notified anytime there is a new snapshot, that is, anytime the text buffer changes. Upon each such change, the extension recomputes syntax highlighting, which is done through a simple lexical scan (that is, the parser is not invoked and no abstract syntax tree is built). After 0.5 seconds of inactivity, the Dafny IDE invokes the Dafny parser, resolver, and type checker on the current buffer snapshot. If the snapshot passes these phases without error, the additional information computed during these phases (e.g., which calls are co-recursive) is made available to the user in hover text. Also, the snapshot is then asynchronously sent to the Dafny verifier, unless the verifier is already running on a previous snapshot. As verification errors are reported by the asynchronously running verifier, they are displayed in the IDE. Once a snapshot has been fully processed by the ver-



Snapshot 1:

```
method Foo()
  ensures P();
{ }

method Bar() { }

function P() : bool { true }
```

Snapshot 2:

```
method Foo()
  ensures P();
{ }

method Bar() { Foo(); }

function P() : bool { true }
```

Snapshot 3:

```
method Foo()
  ensures P();
{ }

method Bar() { Foo(); }

function P() : bool { true }
```

Figure 4.2: Progress indication via colors in the margins. The three program snapshots of the buffer are shown in chronological order (from top to bottom). The dark-orange margin in Snapshot 2 indicates that changes have not yet been sent to the prover, while the purple margin in Snapshot 3 indicates that the verifier has started processing this snapshot.

ifier, a new verification task is started for the current snapshot, unless that is the snapshot that was just verified. This guarantees that the IDE immediately starts a new verification task in case the user made any changes to the program while the previous verification task was still running.

A constant question that users would have about Dafny's previous IDE was, "Is the verifier done yet?". To give the user a sense of the processing that is taking place in the background, the new Dafny IDE uses colors in the margin (see Figure 4.2). A dark-orange color in the margin shows a line that has been edited in a snapshot that has not yet been sent to the verifier, and a violet color in the margin shows a line that has been edited in a snapshot that is currently being processed by the verifier.

On top of this, we adapted the tool architecture to integrate the Boogie Verification Debugger (BVD) [74] directly. Under this change, which is independent of the parallelization, the solver is asked to include the counterexample information needed by BVD with each verification error.

4.2 On-demand Re-verification

Caching is a popular technique for improving the responsiveness of systems that would need to repeatedly perform expensive computations whose output is a function of the given input. Since in a modular verification approach different entities of a program (e.g., modules, classes, or—as in Dafny—methods and functions) are verified in isolation, changes to one program entity usually invalidate only a small fraction of the verification results previously obtained for other program entities. More specifically, one can safely avoid re-verification of an entity by caching previously computed verification results, except when the user has changed some other program entity on which it depends. This optimization is crucial in providing rapid feedback when the program is larger than just a handful of entities.

On a high level, one can see a caching mechanism for partial verification results as a verifier that *swiftly* returns a valid subset of the partial verification results that were stored for the cached program snapshot (i.e., verification results that can be more partial than the ones that were cached) when it is asked to verify the current program snapshot.

Our technique for avoiding re-verification of methods and functions in Dafny deals with two core issues: (1) detecting changes to program entities and (2) tracking dependencies between different program entities to determine what needs to be re-verified. To solve the first issue, we extended Dafny to compute an *entity checksum* for each function, each method, and the specification (e.g., pre- and postconditions) of each method. This checksum is insensitive to various minor syntactic changes of the specific program text, because it is computed based on the Dafny abstract syntax tree. For instance, the checksum of a method does not change if a comment is edited by the user.

To deal with the second issue, these entity checksums are used to track dependencies by computing *dependency checksums* for each program entity (function, method, or method specification) based on its own entity checksum and the dependency checksums of other entities on which it depends directly (e.g., methods it calls). This lets us compare the dependency checksum of a given entity for the current program snapshot with the one stored in our verification result cache to determine if it needs to be re-verified.

In our implementation, we keep the cache in memory and we chose to compute the dependency checksums at the level of Boogie entities, thus making this feature available to other verifiers that target Boogie. The corresponding mechanism is explained in full detail in Chapter 5.

Figure 4.3 illustrates how our technique works on a concrete verification session that consists of three program snapshot, which are sent to the prover in chronological order (i.e., snapshot 1 is the initial program and Snapshot 3 is the final program). All entities of the initial program snapshot need to be verified, since nothing has been cached yet. For Snapshot 2, only method `Bar` needs to be re-verified: the corresponding Boogie implementations (for checking the correctness and well-definedness of the method body) are tagged with an entity checksum that is different from the one in the cache, but the entity checksum of the corresponding Boogie procedure (for capturing the method specification) stays the same. For Snapshot 3, all entities need to be re-verified: the entity checksum of the Boogie function that corresponds to the Dafny function `P` changes with respect to the previous snapshot, which affects the dependency checksums of all remaining Boogie implementations.

Besides this coarse-grained technique for caching verification results for each top-level program entity, we also make use of a more fine-grained caching technique that makes use of cached verification results for parts (e.g., one branch of a conditional statement) of a top-level entity. This technique is more low-level and is explained in full detail in Chapter 5.

One interesting application of our coarse-grained caching technique has to do with prioritizing the program entities that are being verified. Ideally, we want to prioritize entities that are more directly affected by the latest change to the program text, because that is where the user is likely to want to see the effect of the re-verification first. To do that, we assign different levels of priority to an entity based on its current checksums and the ones stored in the verification result cache (see Section 5.2.3 for more details).

Other verification systems have also used forms of checksums and dependencies in order to reduce the need for constructing new proofs. In the heterogeneous Why3 system, both the construction and verification of proof obligations can be parameterized by different transformations and different solvers. To maintain such *proof sessions* as much as possible when any subsystem changes, or if the program under scrutiny changes, Why3 uses checksums and *goal shapes*, a heuristics similarity measure, for matching goals from the existing proof session with new proof goals [15]. For Dafny, we have focused on reducing turnaround time for the user, rather than trying to be robust against changes in components of Dafny itself.

Change management is also important in interactive proof assistants where large parts of proofs are authored by users. Work on such change management has been done, for example, in the context of KIV [101] and KeY [72].

Snapshot 1:

```
method Foo()
  ensures P();
{ }

method Bar() { }

function P(): bool { true }
```

Snapshot 2:

```
method Foo()
  ensures P();
{ }

method Bar() { Foo(); }

function P(): bool { true }
```

Snapshot 3:

```
method Foo()
  ensures P();
{ }

method Bar() { Foo(); }

function P(): bool { false }
```

Figure 4.3: Example of on-demand re-verification. The three program snapshots are ordered chronologically (i.e., Snapshot 1 is the initial program and Snapshot 3 is the final program) and changes between snapshots are highlighted in gray. All entities in Snapshot 1 need to be verified, while for Snapshot 2 only method `Bar` needs to be re-verified. Finally, for Snapshot 3 all entities need to be re-verified since all of them depend directly or indirectly on the modified function `P`.

4.3 Showing Relevant Information On-demand

A verification system typically computes various properties that determine how verification conditions are formulated. For example, Dafny uses heuristics to determine automatically generated induction hypotheses [77]. Sometimes, it can be unclear to the user which properties were computed. For instance, Dafny uses some rules that determine if a function self-call is recursive or co-recursive; a user who does not know the precise rules may want to find out which calls have been determined to be co-recursive.

We devised a simple mechanism by which the Dafny resolver and type checker can associate any information with any AST node. When Dafny is running in the IDE, this information then gets displayed as hover text for the region in the text buffer that corresponds to the respective AST node. We use this mechanism to display the type and kinds of variables (e.g., "(ghost local variable) `x: List⟨int⟩`" or "(destructor) `List.head: T`"), the default **`decreases`** clauses for methods and functions [76], the automatically generated conclusions of **`forall`** statements, which methods are tail recursive, which function calls are co-recursive, the expansion of the syntactic sugar for calls to prefix predicates and prefix methods [78], the class expansion of iterators, and code inherited from a refined module through Dafny's " ... " construct. Currently, such additional information is computed during Dafny's resolution and type checking phases. However, in principle, one could imagine showing information that is computed during the verification phase as well.

This mechanism allows us to make a lot more information accessible to users without overwhelming them with details and without cluttering up the view of the program text. Consequently, it becomes much easier to show relevant information to the user in a demand-driven fashion.

4.4 Focused Error Reporting

When a verification attempt is not going through, a user has to debug the cause. Usually, there are several options: (1) the code itself may be wrong, (2) the specifications may be wrong, (3) more information may be needed to make the proof go through, or (4) the problem could be caused by some incompleteness of the SMT solver or of the program's logical encoding as verification conditions.

One way to debug such a situation is to ask the verifier questions like "does the following condition hold here?" (which is done by adding an **`assert`** statement in the program text) and "can the proof goal be met under this additional assumption?" (which is done by temporarily adding an **`assume`** statement in the program text). This kind of interactive dialog with the verifier is supported well in the Dafny IDE, because the caching (and sometimes parallelization) makes the interaction swift and fluid.

It is also possible to obtain more information about the failing situation. This is done by exploring the counterexample produced by the solver. The Boogie Verification Debugger (BVD), via a Dafny plug-in, makes this counterexample intelligible at the source context [74]. BVD was previously available for Dafny only as a standalone tool, but we have now integrated it directly in the IDE.

Let us describe our interface to BVD. When an attempted verification fails, like the postcondition violation shown in Figure 4.4, a red dot (and a red squiggly line) indicate the path along which the error is reported. The error pane at the bottom of the screen shows the error message, which also appears as hover text for the squiggly line. The error pane also lists source locations related to the error, in this case showing the particular postcondition that could not be verified.

By clicking on a red dot, the Dafny IDE will display more information related to that error, resulting in the screen shown in Figure 4.5. The blue dots that now appear in the program text trace the control path from the start of the enclosing routine and leading to the error. There is state information associated with each blue dot, and the user can click on a blue dot to select a particular state (by default, the last state is selected, which is the state in which the error was detected).

In addition to the blue dots, BVD is brought up in a pane to the right. BVD shows the variables in scope, in a familiar debugger-like fashion, but with two conspicuous differences: some of the values shown are underspecified (the names of these values begin with an apostrophe, like `'7` and `'8`; distinct names refer to distinct values), and some values are not shown at all, because they are not relevant to the counterexample (like all of the array elements of `a`, except the one at index `2804`). Note that the Dafny plug-in for BVD currently does not display values of functions in the counterexample, but we are hoping to add that functionality.

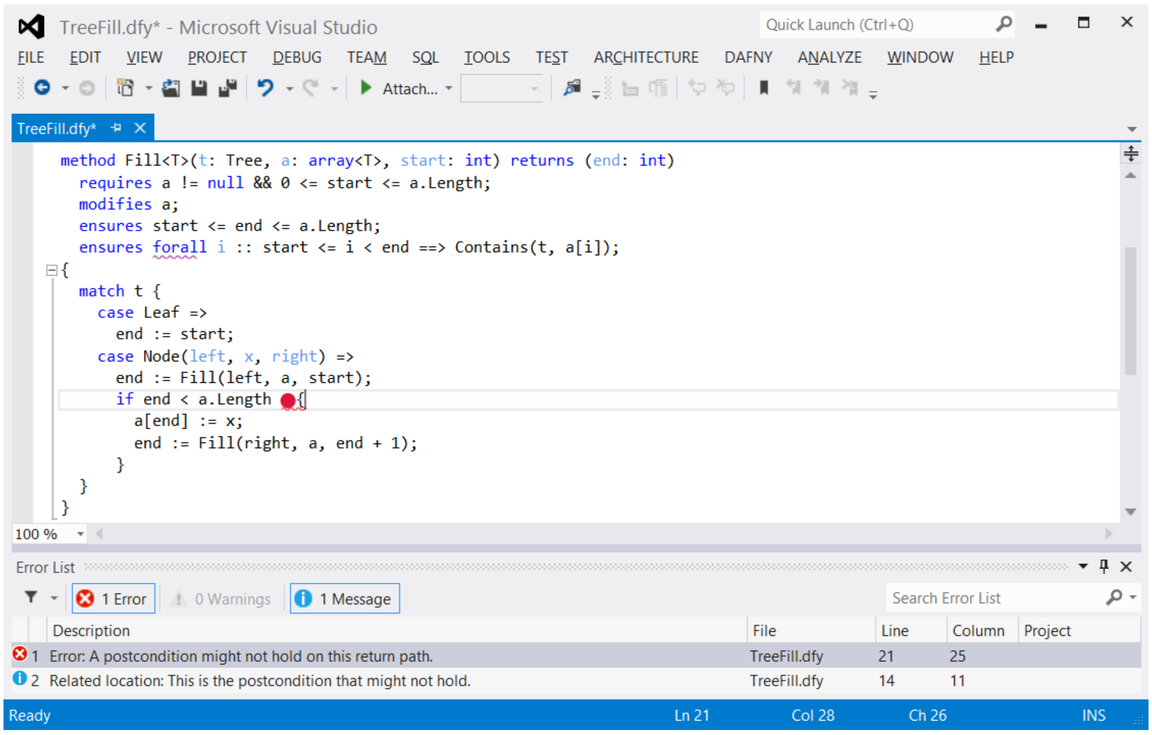
The "Value" column in the BVD pane shows values in the currently selected state, whereas the "Previous" column shows the values in the previously selected state. This gives a simple way to compare the values in two states. In the example in the figure, we had first had the error state selected and then selected the state one line earlier.

Finally, the figure illustrates how values for variables of primitive types (in the currently selected state) are also displayed as hover text.

What all of this tells us for the example is that the postcondition cannot be verified when the bound variable `i` in the postcondition is `2804`. That index of the array is set by the assignment to `a[end]`, but is then changed (from `'7` to `'8`) in the next line where a recursive call to `Fill` is made. Some thinking then reveals that the cause of the verification error is that the postcondition of `Fill` is too weak. We can fix the problem by adding a postcondition about the array indices between `0` and `start`, in particular by saying that `Fill` leaves those array elements unchanged:

```
    ensures ∀ i • 0 ≤ i < start ⟹ a[i] = old(a[i]);
```

By simply adding this postcondition and then waiting a split second, the error goes away.



TreeFill.dfy* - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
FILE EDIT VIEW PROJECT DEBUG TEAM SQL TOOLS TEST ARCHITECTURE DAFNY ANALYZE WINDOW HELP
Attach...
TreeFill.dfy*
100 %
Error List
1 Error
0 Warnings
1 Message
Search Error List
Ready
Ln 21
Col 28
Ch 26
INS

```
method Fill<T>(t: Tree, a: array<T>, start: int) returns (end: int)
  requires a != null && 0 <= start <= a.Length;
  modifies a;
  ensures start <= end <= a.Length;
  ensures forall i :: start <= i < end ==> Contains(t, a[i]);
{
  match t {
    case Leaf =>
      end := start;
    case Node(left, x, right) =>
      end := Fill(left, a, start);
      if end < a.Length ●{
        a[end] := x;
        end := Fill(right, a, end + 1);
      }
  }
}
```

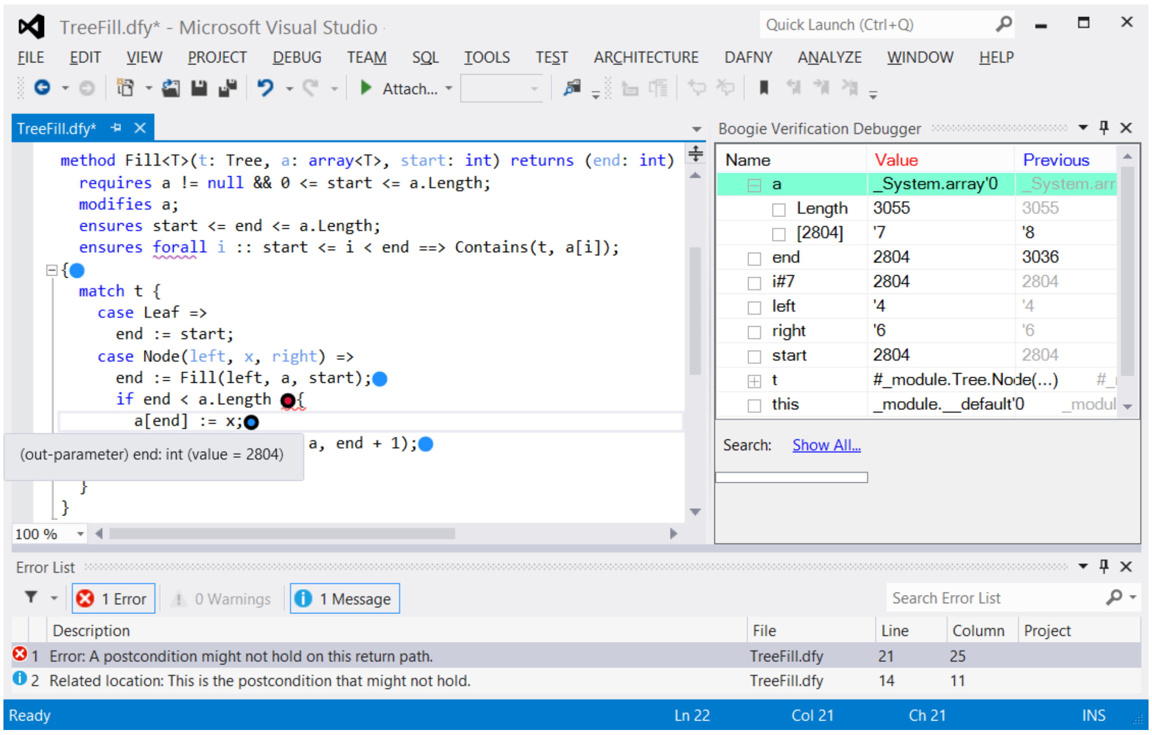
	Description	File	Line	Column	Project
1	Error: A postcondition might not hold on this return path.	TreeFill.dfy	21	25	
2	Related location: This is the postcondition that might not hold.	TreeFill.dfy	14	11	

Figure 4.4: A screenshot of the Dafny IDE. The verification error is displayed in the text buffer as a red dot, which can be selected to obtain more information.

4.5 Diagnosing Timeouts

An important consideration for ensuring a responsive user interaction is what to do with verification tasks that require a long time. At the moment, our IDE performs all verification on a per-method (or per-function) basis. When a method is long and difficult, we often wish for breaking up the verification task into smaller pieces. Boogie has some facilities for *verification-condition (VC) splitting* [79] and *selective checking* of parts of procedure implementations, but our Dafny IDE is currently not taking advantage of these. In fact, since we already cache information from the previous program snapshot, one could even consider to adjust the parameters of VC splitting and selective checking dynamically based on previous verification attempts. Generally, as long as the task terminates in the end, our fine-grained caching helps in reducing that time during the next run.

However, this does not help if the verifier actually runs out of time. Subjectively, we find that timeouts occur in some part of any larger proof attempt, especially those that involve large recursive functions or non-linear arithmetic, *while* the user is working on getting the



TreeFill.dfy* - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
FILE EDIT VIEW PROJECT DEBUG TEAM SQL TOOLS TEST ARCHITECTURE DAFNY ANALYZE WINDOW HELP
Attach...
TreeFill.dfy*
Boogie Verification Debugger
100 %
Error List
Search Error List
Ready
Ln 22
Col 21
Ch 21
INS

```
method Fill<T>(t: Tree, a: array<T>, start: int) returns (end: int)
  requires a != null && 0 <= start <= a.Length;
  modifies a;
  ensures start <= end <= a.Length;
  ensures forall i :: start <= i < end ==> Contains(t, a[i]);
{
  match t {
    case Leaf =>
      end := start;
    case Node(left, x, right) =>
      end := Fill(left, a, start);
      if end < a.Length {
        a[end] := x;
                              a, end + 1);
      }
  }
}
```

(out-parameter) end: int (value = 2804)

Name	Value	Previous
a	_System.array'0	_System.arr
Length	3055	3055
[2804]	'7	'8
end	2804	3036
i#7	2804	2804
left	'4	'4
right	'6	'6
start	2804	2804
t	#_module.Tree.Node(...)	#_
this	_module.__default'0	_modul

Search: Show All...

1 Error | 0 Warnings | 1 Message

	Description	File	Line	Column	Project
1	Error: A postcondition might not hold on this return path.	TreeFill.dfy	21	25	
2	Related location: This is the postcondition that might not hold.	TreeFill.dfy	14	11	

Figure 4.5: A screenshot showing additional information obtained by selecting an error (red dot). The blue dots show the program states along the control path leading to the error, and the BVD pane to the right shows values of variables in the selected state of the selected error.

verification through. That is, timeouts are often a symptom of missing proof ingredients, and good performance tends to be restored once the necessary ingredients have been supplied by the user. Timeouts during this time are bad, since they are on the user's time. Most verifiers struggle with the issue of timeouts. For instance, the Why3 system guards against timeouts by being able to run several solvers at the same time [54].

By default, we set the solver time limit to 10 seconds. While we do allow this default to be overridden through Dafny custom attributes, it rarely seems to help in situations where the verification attempt is actually missing information. For a user to figure out what information is missing (let alone which proof obligations are taking a long time), the solver must end its proof search. However, since in this case the solver is neither able to produce a proper counterexample nor to prove that no counterexample exists, it will return an incomplete counterexample or none at all. Consequently, the verifier usually does not produce as much information for verification attempts that time out as it does for attempts that fail. To provide better feedback to the user in such cases, we have developed a feature for diagnosing timeouts. In particular, it might turn out that the time limit is simply too low or that very few assertions seem responsible for a time out. This allows the user to either adjust the time limit or focus its attention on those particular assertions.

When the verifier runs into a timeout, this is reported much like errors and the user may now decide to re-run the verification in a diagnostic mode. This mode is different from the usual mode in two ways: what verification conditions are generated and what happens during the interaction with the solver after a timeout. The basic idea is to split up the verification condition into smaller fragments after a timeout to decompose the verification task and, thereby, narrow down the number of assertions that seem responsible for the timeout.

To make this possible, we instruct Boogie to generate slightly different verification conditions by, conceptually, marking every assertion A_k as verified under $U(k)$, where U is an uninterpreted function that maps an *assertion identifier* of type integer to a Boolean value. Since nothing is known about this function initially, the program will be verified as usual. However, once a timeout occurs, we can feed additional constraints about applications of that function to the solver. In particular, we can *temporarily* mark some assertions as fully verified (i.e., disable the corresponding checks) to simplify the verification task.

Figure 4.6 shows our algorithm for decomposing the verification task once a timeout has been encountered. It takes four arguments: (1) the current verification condition VC, (2) the set of unverified assertion identifiers U (initially contains all assertion identifiers in the verification condition), (3) the integer F to determine what fraction of those assertions to check next (initially 2), and (4) the set of timed-out assertion identifiers T (initially empty).

If the set U is empty, we are done and return the result TimeOut, in case there are any assertion identifiers in T; otherwise we return Verified. If set U is non-empty, we choose a subset of the unverified assertion identifiers S (on line 9) and check only those assertions for a fixed time TIME_LIMIT_PER_ASSERTION. If we find a failing assertion, we terminate immediately. If the check successfully verified the assertions in S, we recursively diagnose the remaining assertions. Otherwise, we try to check a smaller set of identifiers by invoking procedure diagnose while doubling the argument F. If the set cannot get smaller, we found an assertion that seems to be responsible for the timeout and we try to check the remaining assertions.

The procedure check_some checks the verification condition after temporarily marking some assertions as fully verified. To do so, it makes use of scopes in the solver that push and later pop additional constraints about applications of function U for assertion identifiers that are not in set S.

This algorithm allows us to determine the set of assertions that cannot be verified individu-

```
procedure diagnose(VC, U, F, T) {
  if (|U| = 0) {
    if (0 < |T|) {
      report the timed-out assertions in T;
      return TimeOut;
    }
    return Verified;
  }
  choose S, such that S ⊆ U ∧ |S| = max(|U| / F, 1);
  var R := check_some(VC, S, TIME_LIMIT_PER_ASSERTION);
  if (R = Error) {
    return R;
  } else if (R = Verified) {
    return diagnose(VC, U \ S, 1, T);
  } else {
    if (2 ≤ (|U| / F)) {
      return diagnose(VC, U, 2 * F, T);
    } else {
      return diagnose(VC, U \ S, 1, T ∪ S);
    }
  }
}
```

Figure 4.6: Algorithm for diagnosing timeouts by decomposing the verification task. It takes the following arguments: the current verification condition VC, the set of unverified assertion identifiers U (initially all assertion identifiers in the verification condition), an integer F to specify what fraction of those assertions to check next (initially 2), and a set of timed-out assertion identifiers (initially none). The procedure check_some checks only some assertions in a verification condition and terminates when the specified time limit is reached.

ally within the fixed time limit TIME_LIMIT_PER_ASSERTION. These provide a clear indication to the user about what to focus on to reduce the running time of the verifier. In practice, we set this time limit to 10% of the time limit for the entire method or function. However, one could also imagine that a user might choose this value manually; for instance, to narrow down the set of assertions that seem responsible even further.

Note that what we presented here is essentially a black-box approach for diagnosing timeouts; i.e., it does not require any internal information from the solver. However, this might be an interesting alternative to explore. For instance, one could try analyzing the solver logs that the Z3 Axiom Profiler gives access to.

Program	Tasks	Time (in seconds)		
		1 Solver	**2 Solvers**	**4 Solvers**
1 (synthetic)	202	221	119 (× 1.86)	73 (× 3.02)
2 (synthetic)	800	178	82 (× 2.17)	49 (× 3.63)
3	244	137	72 (× 1.90)	51 (× 2.67)
4	59	12	7 (× 1.71)	7 (× 1.71)
5	104	27	15 (× 1.80)	13 (× 2.08)
6	10	18	14 (× 1.29)	12 (× 1.50)
7	66	19	13 (× 1.46)	12 (× 1.58)

Table 4.1: Performance with parallelization. The three right-most columns show the running times for three different configurations: 1 solver instance (i.e., no parallelization), 2 solver instances, and 4 solver instances. Speedup factors over no parallelization are shown in parentheses. The second column shows the number of individual verification tasks (i.e., Boogie procedure implementations) that were executed.

4.6 Evaluation

The Dafny IDE has been under active development for several years. During this period we have continuously designed new features and techniques to improve the user experience. Most of them were developed in response to feedback from users, including students and professional developers. In this section, we focus on evaluating two aspects: parallelizing verification tasks to increase responsiveness and diagnosing timeouts to provide more helpful feedback to users. We provide an evaluation of another important aspect, namely on-demand re-verification, in Section 5.4.

4.6.1 Parallelizing Verification Tasks

Table 4.1 demonstrates how parallelization with multiple solver instances affects the performance of the verifier for several long-running verification tasks. Programs 1 and 2 are solutions to two challenges from a verification competition [17] that each contain a single Dafny method. To measure the performance, we have created 100 copies of that method for program 1 and 400 copies of that method for program 2. We can see that the running times are reduced significantly when using more solver instances.

The remaining programs are large Dafny programs of up to 1300 lines of code. Programs 3,

	Time limit per assertion	
	Low	**High**
TimeOut (in %)	57.89	50.00
Error (in %)	17.11	20.69
Verified (in %)	25.00	29.31
Average number of prover queries	65.67	51.00
Average time (relative to time limit per method/function)	6.24	9.25
Average number of assertions that seem responsible	2.67 (0.15%)	1.84 (0.11%)

Table 4.2: Comparison between two configurations for diagnosing timeouts. The configurations only differ by the parameter TIME_LIMIT_PER_ASSERTION from the algorithm in Figure 4.6. The first three rows show how often the algorithm returns which output. The fourth row shows the average number of additional prover queries for diagnosing a timeout. The fifth row shows the average running time of the algorithm relative to the time limit per method/function. The sixth row shows for verification conditions that still result in a timeout the average number and percentage (out of all assertions in the timed-out verification condition) of timed-out assertions that seem responsible.

4, and 5 are taken from a formalization of a build language using Dafny. Program 6 is an implementation [76] of the Schorr-Waite algorithm [102] for marking all nodes that are reachable from a given node in a graph, and program 7 is an implementation of the C5 library's snapshotable tree data structure. For these the running times are also reduced significantly when using multiple solver instances.

However, we can see that the running times do not always scale linearly with the number of solver instances. This is due to the fact that some methods require much more time than others. For instance, for program 6 almost all the time is spent on a single method, which makes it hard to balance the work load between solvers. In principle, these are cases where verification-condition splitting [79] could help in balancing the work load.

On one hand, this demonstrates that multiple solver instance can help to increase the responsiveness of the IDE. On the other hand, it illustrates why our technique for caching of verification results turned out to be even more crucial in practice. This is due to the fact that it can reduce both the number of verification tasks *and* the time for verifying each task. A detailed evaluation of our caching technique can be found in Section 5.4.

4.6.2 Diagnosing Timeouts

To evaluate our technique for diagnosing timeouts we have run it on 39 programs from the sessions that are described in Section 5.4. We have used two different configurations that only differ by the parameter TIME_LIMIT_PER_ASSERTION from the algorithm in Figure 4.6:

- **Low** (10% of the time limit per method/function), and
- **High** (20% of the time limit per method/function).

Table 4.2 demonstrates the different trade-offs clearly. While configuration **Low** is significantly faster by using a larger number of short prover queries, it results in timeouts more often and is able to narrow down the set of timed-out assertions that seem responsible less aggressively. For verification conditions that still result in a timeout, configuration **Low** reports on average 0.15% (at most 10 assertions) of all assertions in that method/function as responsible. For configuration **High** these numbers are significantly lower (0.11% on average, at most 4 assertions).

Independently, both configurations are able to avoid a large number of timeouts by decomposing the verification tasks (as shown by the first three rows in Table 4.2). For instance, with configuration **High** the algorithm from Figure 4.6 returns the result Verified or Error for 50% of the timed-out verification conditions. This means that for those verification conditions none of the assertions required more time than the limit per assertion. This suggests that the user could avoid the timeout by increasing the time limit for the corresponding method or function. We believe that configuration **Low** is more useful in practice since it provides feedback to users more quickly.

4.7 Summary

The Dafny IDE represents a new generation of interaction between user and verification system. We have built dependency analysis, caching, and concurrent verification into the design-time feedback loop to make re-verification responsive with minimal user effort.

Besides this, we have developed several techniques to provide useful information to the user in a demand-driven and unobtrusive way. The IDE provides a deep integration of the Boogie Verification Debugger which displays information about failed verification attempts in the program text and can be controlled directly from within the program text. To provide better feedback to the user in cases where the verifier times out, we have developed a technique for diagnosing timeouts that is able to narrow down the set of assertions that seem responsible for a timeout. Finally, using hover text, we have given easy access to computed information without cluttering up the user display with irrelevant information.

CHAPTER 5

Fine-grained Caching of Verification Results

Making formal program verification useful in practice requires not only automated logical theories and formal programming-language semantics, but also—inescapably—a human understanding of why the program under verification might actually be correct. This understanding is often gained by trial and error, debugging verification attempts to discover and correct errors in programs and specifications and to figure out crucial inductive invariants. To support this important trial and error process, it is essential that the integrated development environment (IDE) provides rapid feedback to the user.

In this chapter, we describe the caching of verification results in the Dafny IDE (see Chapter 4) in more detail. In particular, we focus on how our technique for fine-grained caching makes use of cached results from earlier runs of the verifier. As mentioned in the high-level description from Section 4.2, one can think of a cache for partial verification results as a verifier that *swiftly* returns a valid subset of the partial verification results that were stored for the cached program snapshot (i.e., verification results that can be more partial than the ones that were cached) when it is asked to verify the current program snapshot.

The effect of this caching is to reduce the time from user keystrokes in the editor to the reporting of verification errors that are gathered in the background. When editing programs with more than just a few methods, this lag time is now often around a second where it previously took tens of seconds for the verifier to repeat the checking of proof obligations that were not affected by the latest change.

These improvements rely on a basic caching technique that tracks dependencies using the program's call graph to avoid re-verification of methods that were not affected by the most recent change to the program (see Section 4.2 for a high-level overview from the perspective of the user of the IDE).

Our fine-grained caching takes this a step further. It is motivated by the fact that when a proof obligation is not automatically verified, a user tends to spend human focus and editing in one small area of the program. Often, this area can be in one branch of a method, so if the tool can rapidly re-verify just what has changed, the user can make progress more quickly. Our fine-grained caching thus makes use of the program's control-flow graph.

Like other verifiers, the Dafny verifier generates proof obligations by translating Dafny to an intermediate verification language (IVL), namely Boogie [4, 81]. We designed our fine-grained caching to operate at the level of the IVL, which makes it possible for other Boogie front ends to make use of the new functionality. Our novel caching approach compares the current *snapshot* of a Boogie program with a previously verified snapshot. It then instruments the current snapshot using our technique of partial verification results from Chapter 2 to adjust the proof obligations accordingly. Finally, it passes the instrumented Boogie program to the underlying satisfiability-modulo-theories (SMT) solver in the usual way. Our implementation is available as part of the Boogie [1] and Dafny [2] open source projects.

This chapter is based on a paper that was presented at the *International Conference on Computer Aided Verification (CAV)* in 2015 [83].

Outline

In Section 5.1, we explain a motivating example in more detail. Section 5.2 gives background on the architecture of the Dafny verifier and describes the basic, coarse-grained caching based on the program's call graph. We describe our fine-grained caching in Section 5.3 and evaluate how both techniques improve the performance of the verifier in Section 5.4. We review related work in Section 5.5 and summarize our results in Section 5.6.

5.1 Motivating Example

Let us consider some typical steps in the interactive process of developing a verifiably correct program, indicating where our caching improvements play a role. Figure 5.1 shows an

[1] https://github.com/boogie-org/boogie
[2] http://dafny.codeplex.com/

```
datatype Color = Red | White | Blue

predicate Ordered(c: Color, d: Color) { c = Red ∨ d = Blue }

method Sort(a: array<Color>)
  requires a ≠ null
  modifies a
  ensures forall i,j • 0 ≤ i < j < a.Length ⟹ Ordered(a[i], a[j])
{
  var r, w, b := 0, 0, a.Length;
  while w ≠ b
    invariant 0 ≤ r ≤ w ≤ b ≤ a.Length
    invariant forall i • 0 ≤ i < r ⟹ a[i] = Red
    invariant forall i • r ≤ i < w ⟹ a[i] = White
    invariant forall i • b ≤ i < a.Length ⟹ a[i] = Blue
  {
    match a[w]
      case Red ⇒
        a[r], a[w] := a[w], a[r]; r := r + 1;
      case White ⇒
        w := w + 1;
      case Blue ⇒
        b := b - 1;
  }
}
```

Figure 5.1: Incomplete attempt at implementing the Dutch Flag algorithm. As written, the program contains a specification omission, a specification error, and two coding errors. As the program is edited, our fine-grained caching of verification results enables a more responsive user experience by avoiding re-verification of unaffected proof obligations.

incomplete attempt at specifying and implementing the Dutch Flag algorithm, which sorts an array of colors.

The program gives rise to several proof obligations, following the rules of Hoare logic. The loop invariants are checked when control flow first reaches the loop. The loop body with its three branches is checked to decrease a termination metric (here provided by the tool: the absolute difference between `w` and `b`) and to maintain the loop invariants. The postcondition of the method is checked to follow from the loop invariants and the negation of the guard (without further inspection of the loop body). For every call to method `Sort` in the rest of the program, the method's precondition is checked and its postcondition is assumed.

In addition, all statements and expressions, including those in specifications, are verified to be well-formed. For example, for the assignment that swaps two array elements in the loop body (line 19), the well-formedness checks ensure that the array is not **`null`**, that the indices are within bounds of the array, that the method is allowed to modify the heap at these locations, and that the parallel assignment does not attempt to assign different values to the same heap location.

To provide *design-time* feedback to the user, the Dafny IDE automatically runs the verifier in the background as the program is being edited. This allows the verifier to assist the user in ways that more closely resemble those of a background spell checker. Given the program in Figure 5.1, the Dafny verifier will report three errors:

1. **DutchFlag.dfy(11,5):** A postcondition might not hold on this return path.
   – **DutchFlag.dfy(8,13):** This is the postcondition that might not hold.
2. **DutchFlag.dfy(11,5):** Cannot prove termination; try supplying a decreases clause for the loop.
3. **DutchFlag.dfy(15,17):** This loop invariant might not be maintained by the loop.

The first error message points out that the method body may not establish the postcondition. Selecting this error in the Dafny IDE brings up the verification debugger [74], which readily points out the possibility that the array contains two `White` values. To fix the error, we add a disjunct `c = d` to the definition of predicate `Ordered`. Instead of expecting the user to re-run the verifier manually, the Dafny IDE will do so automatically.

To speed up this process, the basic caching technique will already avoid some unnecessary work by using the call graph: only methods that depend on the predicate `Ordered` will be re-verified, which includes the body of `Sort` and, since the postcondition of `Sort` mentions the predicate, all callers of `Sort`. Caller dependencies get lower scheduling priority, since they are likely to be further away from the user's current focus of attention. However, we can hope for something even better: the maintenance of the loop invariant in `Sort` need not be re-verified, but only the fact that the loop invariant and the negation of the guard establish the postcondition. Our fine-grained caching technique makes this possible.

The second error message points out that the loop may fail to terminate. Selecting the error shows a trace through the `Red` branch of the **`match`** statement, and we realize that this branch also needs to increment `w`. As we make that change, the tool re-verifies only the loop body, whereas it would have re-verified the entire method with just the basic caching technique.

The third error message points out that the last loop invariant is not maintained by the `Blue` branch. It is fixed by swapping `a[w]` and `a[b]` after the update to `b`. After doing so, the re-verification proceeds as for the second error.

Finally, it may become necessary to strengthen `Sort`'s postcondition while verifying some caller—it omits the fact that the final array's elements are a permutation of the initial array's. If only the basic caching was used, the addition of such a postcondition would cause both `Sort` and all of its callers to be re-verified. By using the fine-grained caching, the body of `Sort` is re-verified to check only the new postcondition (which in this case will require adding the postcondition also as a loop invariant). For callers, the situation is even better: since the change of `Sort`'s specification only strengthens the postcondition, proof obligations in callers that succeeded before the change are not re-verified.

The performance improvements that we just gave a taste of have the effect of focusing the verifier's attention on those parts of the program that the user is currently, perhaps by trial and error, editing. The result is a user experience with significantly improved response times. In our simple example program, the time to re-verify the entire program is about 0.25 seconds, so caching is not crucial. However, when programs have more methods, contain more control paths, and involve more complicated predicates, verification times can easily reach tens of seconds. In such cases, our fine-grained caching can let the user gain insight from the verification tool instead of just becoming increasingly frustrated and eventually giving up all hopes of ever applying formal verification techniques.

5.2 Verification Architecture and Basic Caching

In this section, we describe the role of the intermediate verification language Boogie and the basic caching technique that the fine-grained caching builds on. We have presented a high-level overview of the basic caching technique in Section 4.2 with a focus on the user experience within the Dafny IDE [82]. Here, we describe the technique in more detail and we focus on the underlying caching system we developed for the Boogie verification engine.

5.2.1 Architecture

We have described the overall architecture behind verifiers—such as Dafny—that make use of the Boogie intermediate verification language to express verification conditions in Section 4.1. Boogie supports a modular verification approach by verifying procedure implementations individually. More precisely, calls in procedure implementations are reasoned about only in terms of their specification (i.e., the corresponding procedure declaration). Consequently, a change to a program often does not invalidate verification results obtained for independent program entities. In particular, a change in a given procedure implementation does not invalidate verification results of other procedure implementations, and a change in a procedure's specification may invalidate verification results only of its callees and of the corresponding procedure implementation.

5.2.2 Basic Caching

While the Boogie pipeline accepts a single program, obtains verification results, and then reports them, the basic caching mechanism turns Boogie into more of a verification service: it accepts a stream of programs, each of which we refer to as a *snapshot*. The basic caching approach exploits the modular structure of Boogie programs by determining which program entities have been changed *directly* in the latest program snapshot and which other program entities are *indirectly* affected by those changes.

To determine direct changes, Boogie relies on the client front end (Dafny in our case) to provide an *entity checksum* for each function, procedure, and procedure implementation. For example, the Boogie program in Figure 5.2 shows entity checksums provided by a front end to Boogie via the `:checksum` custom attribute. In our implementation, Dafny computes them as a hash of those parts of the Dafny abstract syntax tree that are used to generate the corresponding Boogie program entities. This makes checksums insensitive to certain textual changes, such as ones that concern comments or whitespace.

To determine indirect changes, Boogie computes *dependency checksums* for all functions, procedures, and procedure implementations based on their own entity checksum and the dependency checksums of entities they depend on directly (e.g., callees). These checksums allow the basic caching to reuse verification results for an entity if its dependency checksum is unchanged in the latest snapshot.

For example, when computing dependency checksums from entity checksums in Figure 5.2, Boogie takes into account that both implementations depend on the procedure declaration of `abs` (implementation `abs` needs to adhere to its procedure declaration and `main` contains a call to `abs`). Consequently, a change that only affects the entity checksum of *procedure* `abs` (e.g., to strengthen the postcondition) will prevent Boogie from returning cached verification results for both implementations. However, a change that only affects the entity checksum of *implementation* `abs` (e.g., to return the *actual* absolute value) will allow Boogie to return cached verification results for implementation `main`.

```
procedure {:checksum "727"} abs(a: int) returns (r: int)
  ensures 0 ≤ r;

implementation {:checksum "733"} abs(a: int) returns (r: int)
{
  r := 0;
}

implementation {:checksum "739"} main()
{
  var x: int;
  call x := abs(-585);
  assert x = 585;
}
```

Figure 5.2: Boogie program illustrating how front ends can use custom attributes on declarations to assign entity checksums. Their computation may be front-end specific.

Figure 5.3 gives an architectural overview of the caching system. In terms of it, the basic caching works as follows. First, Boogie computes dependency checksums for all entities in a given program snapshot. Then, for each procedure implementation P, the cache is consulted. If the cache contains the dependency checksum for P, branch (1) is taken and the cached verification results are reported immediately. Otherwise, branch (2) is taken and the procedure implementation is verified as usual by the Boogie pipeline. Our fine-grained caching may also choose branch (3), as we explain in Section 5.3.

5.2.3 Prioritizing Verification Tasks using Checksums

Besides using them for determining which procedure implementations do not need to be re-verified, we use the checksums for determining the order in which the others should be verified. Ideally, procedure implementations that are more directly related to the user's latest changes are given higher priority, since these most likely correspond to the ones the user cares about most and wants feedback on most quickly. Note that this is mainly important if the IDE reports errors to the user asynchronously, which is the case for the Dafny IDE (see Section 4.1).

The checksums provide a metric for achieving this by defining four priority levels for procedure implementations:

– low (unlike the entity checksum, the dependency checksum in the cache is different from the current one): Only dependencies of the implementation changed.

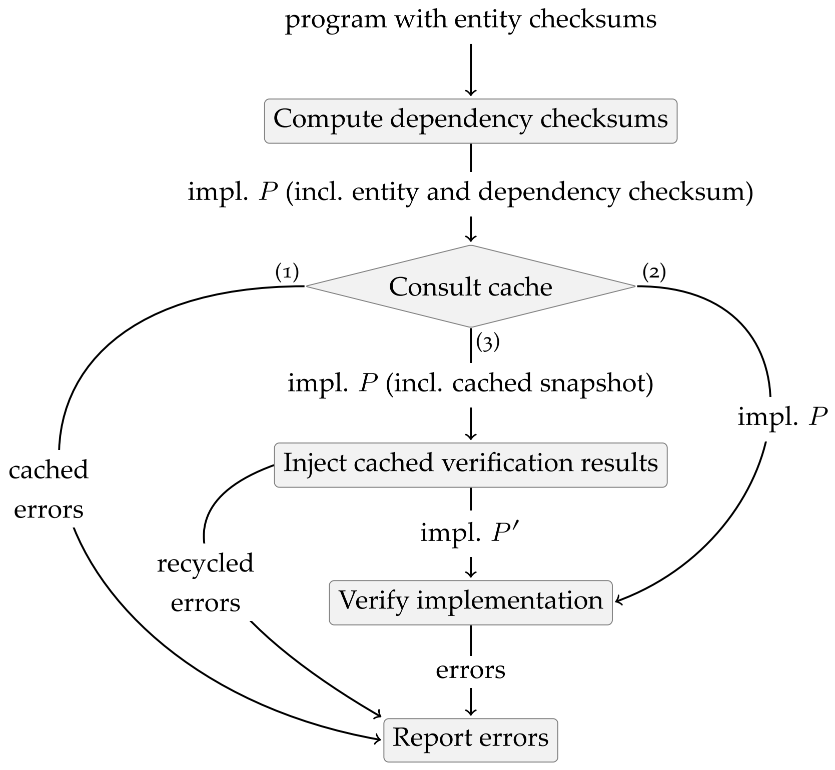

program with entity checksums
Compute dependency checksums
impl. P (incl. entity and dependency checksum)
Consult cache
(1)
(2)
(3)
impl. P (incl. cached snapshot)
impl. P
cached errors
Inject cached verification results
impl. P′
recycled errors
Verify implementation
errors
Report errors

Figure 5.3: Overview of the verification process for procedure implementations. Boxes correspond to components and arrows illustrate data flow. The caching component produces three possible outputs: (1) cached errors in case the entity and dependency checksums are unchanged, (2) the implementation P in case it is not contained in the cache, or (3) the implementation P and the cached snapshot in case either the entity or the dependency checksum have changed. Cached snapshots are used to inject verification results into the implementation and to identify errors that can be recycled.

– medium (entity checksum in the cache is different from the current one): The implementation itself changed.
– high (no cache entry was found): The implementation was added recently.
– highest (both the entity and the dependency checksum is the same as the one in the cache): The implementation was not affected by the change and a cache lookup is sufficient for reporting verification results to the user *immediately*, instead of waiting for other implementations to be verified.

```
 1 procedure gcd(x, y: int) returns (r: int)
 2   requires 0 < x ∧ 0 < y;
 3   ensures 0 ≤ r;
 4
 5 implementation gcd(x, y: int) returns (r: int)
 6 {
 7   if (x < y) {
 8     call r := gcd(x, y - x);
 9     assert 1 ≤ r;
10   } else if (y < x) {
11     call r := gcd(x - y, y);
12   }
13   assert 0 < x + y;
14 }
```

Figure 5.4: Incomplete attempt at implementing a Boogie procedure for computing the greatest common denominator. Boogie reports a postcondition violation for the implementation and an assertion violation on line 9.

5.3 Fine-grained Caching

Basic caching can determine which procedure implementations in a new snapshot do not need to be re-verified at all, but it does not track enough information to allow us to reuse verification results for parts of an implementation. In this section, we present an extension of the basic caching that reuses verification results in fine-grained ways. In particular, our extension avoids re-verification of checks that were not affected by the most recent change and it recycles errors that are still present in the current snapshot.

Before giving our full algorithm, we sketch how it works in two common scenarios we want to address: when an isolated part of a procedure implementation (e.g., one of two branches or a loop body) has been changed, and when the specification of a procedure has been changed.

We proceed by example, starting from the program in Figure 5.4. Running Boogie on this program results in two errors: a failure to establish the postcondition on line 3 and an assertion violation on line 9. To fix the postcondition error in the program in Figure 5.4, the user might add an explicit else branch on line 12 and insert statement `r := x`. This is an instance of the common change-in-isolated-part scenario. In particular, the change has no effect on the assertion on line 9, and thus we would hope to be able to cache and recycle the error.

5.3.1 Fine-grained Dependency Tracking using Statement Checksums

To cache and reuse verification results at this fine granularity, we need to know what each statement depends on. To determine this, we compute a *statement checksum* for every statement from a hash of its pretty-printed representation and—to keep the overhead small—the statement checksums of *all* statements that precede it in the control flow (as opposed to ones that actually affect it). If a statement contains a function call in some subexpression, then the statement depends on the callee's definition and we include the callee's dependency checksum when computing the statement checksum.

The computation of statement checksums occurs after the Boogie program has undergone some simplifying transformations. For example, loops have been transformed using loop invariants and back-edges of loops have been cut [6]; thus, the computation of statement checksums does not involve any fixpoint computation. As another example, the checks for postconditions have been made explicit as **`assert`** statements at the end of the implementation body and the preconditions of procedure implementations have been transformed into **`assume`** statements at the beginning of the implementation body; thus, these statements are taken into account for computing the statement checksums.

After the simplifications from above, there are only two kinds of statements that lead to checks: assertions and calls (precondition of callee). We will refer to them as *checked statements*. We introduce a cache that associates statement checksums of such statements in a given implementation with verification results. Before verifying a new snapshot, we compute statement checksums for the new snapshot and then instrument the snapshot by consulting this cache.

Let us describe this in more detail using our example. We will refer to the program in Figure 5.4 as Snapshot 1 and the program resulting from adding the else branch and assignment on line 12 as Snapshot 2.

After verifying Snapshot 1, the cache will have entries for the statement checksums of the following checked statements: the failing assertion on line 9, the succeeding precondition checks for the calls on lines 8 and 11, the succeeding assertion on line 13, and the failing check of the postcondition from line 3. The statement checksums for the first three checked statements (on lines 8, 9, and 11) in Snapshot 2 are the same as in Snapshot 1. Since the cache tells us the verification results for these, we report the cached error immediately and we add **`assume`** statements for the checked condition before these checked statements in Snapshot 2. Note that, in the case of assertions, this is equivalent to marking them as fully verified.

The statement checksums of the fourth and fifth checked statement are different in Snapshot 2, since they are affected by the modification of line 12. Since the new checksums are not found in the cache, the statements are not rewritten. As a result, Boogie needs to only verify those checks. Indeed, Boogie is now able to prove both and it updates the cache accordingly. With reference to Figure 5.3, what we have just described takes place along branch (3) after the basic cache has been consulted.

5.3.2 Injecting Partial Verification Results

To fix the failing assertion on line 9 in Figure 5.4, the user might now decide to strengthen the postcondition of the procedure by changing it to `1` $\leq$ `r`. This is an instance of the common change-in-specification scenario. In this case, the change involves a strengthened postcondition, and we would therefore hope to avoid re-verifying any previously succeeding checks downstream of call sites. We will refer to the program resulting from the user's change as Snapshot 3.

After Boogie computes the statement checksums, only the statement checksum for the assertion of the postcondition will be different from the ones in the cached snapshot. However, since the dependency checksums of the callee changed for both calls, we introduce an explicit assumption after each call to capture the condition assumed at this point in the cached snapshot.

Note that, unlike in Chapter 2, we will not explain our instrumentation in terms of `assumed` statements and partially-justified `assume` statements, but directly in terms of assignments and regular `assume` statements. Since Boogie is an intermediate verification language, we decided not to introduce new syntactic constructs to add support for expressing partial verification results. Instead, Boolean variables can simply be declared as assumption variables by using an `:assumption` custom attribute. This will make sure that such variables are initialized to **`true`** and that they are only assigned to once using a statement of the form `a` $:=$ `a` $\wedge$ P, where `a` is the assumption variable and P is a Boolean condition.

To capture the condition C that was assumed after the call in the cached snapshot, we introduce a unique assumption variable `a` for each such call and assign `a` $\wedge$ C to it after the corresponding call. The variable allows us to later refer to an assumption that was made at a specific program point. For instance, by using a regular `assume` statement to mark a check that was not failing in the corresponding cached snapshot as partially verified under a conjunction of assumption variables.

To illustrate, consider the rewrite of Snapshot 3 in Figure 5.5. At this stage, the precondition is assumed explicitly on line 3 and the postcondition is asserted explicitly on line 20 as described earlier. On line 1, we introduce one assumption variable for each call to a procedure with a different dependency checksum. The call on line 7 gets to assume the new postcondition of `gcd`. If that call happens to return in a state that was allowed by the previous postcondition (`0` $\leq$ `r`), then assumption variable `a0` will remain true after the update on line 8. But if the call returns in a state that does not satisfy the previously assumed postcondition, then `a0` will be set to **`false`**.

In our example, since the postcondition of the callee is strengthened, the explicit assumption `0` $\leq$ `r` will always evaluate to true. Indeed, this works particularly well when postconditions are not weakened, but, depending on the calling context, it may also simplify the verification otherwise. For instance, it would work for a call where the state is constrained such that for this particular call site the previous postcondition holds after the call, even though the new postcondition is indeed weaker.

Next, we inject assumptions into the program about checked statements that are found to be non-failing in the cached snapshot based on their statement checksum. More precisely, for each statement with checked condition P whose statement checksum is in the cache and that was non-failing in the cached snapshot, we inject an assumption $A \implies P$, where A is the conjunction of all assumption variables. Intuitively, this tells the verifier to skip this check if all assumption variables are true. Otherwise, the verifier will perform the check since a state was reached for which it has not already been verified in the cached snapshot. In other words, the check has been marked as *partially verified* by adding the equivalent of a partially-justified `assume` statement. As an optimization, we include in A only those assumption variables whose update statement definition can reach this use; we refer to these as *relevant* assumption variables.

Figure 5.5 shows the assumptions being introduced on lines 6, 11, and 17, preceding the precondition checks and the assert statement, thus marking these checks as partially verified. Note that the assertion on line 9 is not marked as partially verified, since it is a failing assertion in Snapshot 2. Since the assumption variables remain true, the partially verified checks in effect become fully verified in this example.

In general, the verifier may discover that only some partially verified checks are in effect fully verified depending on the state at those checks. For instance, this may happen if the state after some call was not always allowed by the callee's previous postcondition, but some partially verified checks after that call are in a conditional branch where the branching condition constrains the state such that all states are allowed by the previous postcondition *there*.

In general, one could imagine making the caching even more fine-grained in cases where

```
 1 var {:assumption} a0, a1: bool;
 2
 3 assume 0 < x ∧ 0 < y;  // precondition
 4
 5 if (x < y) {
 6   assume (true) ⟹ (0 < x ∧ 0 < y - x);
 7   call r := gcd(x, y - x);
 8   a0 := a0 ∧ (0 ≤ r);
 9   assert 1 ≤ r;
10 } else if (y < x) {
11   assume (true) ⟹ (0 < x - y ∧ 0 < y);
12   call r := gcd(x - y, y);
13   a1 := a1 ∧ (0 ≤ r);
14 } else {
15   r := x;
16 }
17 assume (a0 ∧ a1) ⟹ (0 < x + y);
18 assert 0 < x + y;
19
20 assert 1 ≤ r;  // postcondition
```

Figure 5.5: Body of the procedure implementation for Snapshot 3 after injecting cached verification results (shown in the gray boxes). The instrumented program contains two explicit assumptions [25] on lines 8 and 13 derived from the postcondition of the cached callee procedure. Also, all checks that did not result in errors in the cached snapshot have been marked as partially verified by introducing **`assume`** statements on lines 6, 11, and 17.

other assumptions in the program (e.g., resulting from user-provided `assume` statements, preconditions, and user-provided or inferred loop invariants) are affected by a change. We believe that—much like for procedure calls—we can use explicit assumptions to capture assumptions that were made in the cached snapshot, and thereby mark more checks as partially verified.

5.3.3 Algorithm for Injecting Cached Verification Results

In this subsection, we present our algorithm for injecting cached verification results in procedure implementations of medium or low priority, for which no limit on the number of reported errors was hit when verifying the cached implementation. At this point, most existing Boogie transformations have been applied to the implementation as described earlier (e.g., eliminating loops using loop invariants and adding explicit checks for postconditions).

As a first step, we compute statement checksums for all statements in an implementation as defined earlier.

As a second step, we insert explicit assumptions for calls if the dependency checksum of the callee has changed in the current snapshot. More precisely, for each call, we distinguish between three different cases, in order:

1. Dependency checksum of callee is the same as in the cached snapshot: We do not need to do anything since the asserted precondition and the assumed postcondition are the same as in the cached snapshot.
2. All functions that the callee transitively depended on in the cached snapshot are still defined and unchanged in the current snapshot: Before the call, we add the statement **assume** ? $\implies P$, where ? is a placeholder that will be filled in during the final step of the algorithm and P is the precondition of the callee in the cached snapshot. This may allow us to reuse the fact that the precondition of a call has been verified in the cached snapshot. To simplify the presentation, we will only later determine if the precondition has indeed been verified and under which condition. Since the dependency checksum of the callee is different from the one in the cached snapshot, we additionally introduce an explicit assumption to capture the condition that was assumed after the call in the cached snapshot. This condition depends on the callee's *modifies clause* (which lists the global variables that the callee is allowed to modify) and its postcondition. To capture the former, let V be the set of global variables that were added to the callee's modifies clause since the cached snapshot. We now add `ov` $:=$ `v` for each global variable `v` in this set V before the call, where `ov` is a fresh, local variable. This allows us to express the explicit assumption by adding the statement `a` $:=$ `a` $\wedge$ `(`$Q \wedge M$`)` after the call, where `a` is a fresh assumption variable, Q is the postcondition of the callee in the cached snapshot and M contains a conjunct `ov` $=$ `v` for each global variable `v` in the set V. Note that M does not depend on global variables that were removed from the callee's modifies clause since the cached snapshot; the statements after the call have already been verified for all possible values of such variables.
3. Otherwise: Since we cannot easily express the pre- and postcondition of the callee in the cached snapshot, we need to be conservative. We therefore do not add any assumption about the precondition and we add the statement `a` $:=$ `a` $\wedge$ **false** after the call, where `a` is a fresh assumption variable.

As a third step, we transform each checked statement with the checked condition P to express cached verification results. We distinguish four cases, in order:

1. Some relevant assumption variable is definitely false when performing constant prop-

agation: We do not do anything, since we cannot determine under which condition the check may have been verified.
2. There was an error for this check in the cached implementation and there are no relevant assumption variables: Since it has previously resulted in an error under identical conditions, we add the statement **`assume`** P before and report the error immediately to avoid unnecessary work.
3. There was no error for this check in the cached implementation: Since it has been verified previously, we add the statement **`assume`** $A \implies P$ before, where A is the conjunction of all relevant assumption variables. If there are any such assumption variables, we say that the check has been marked as *partially* verified; otherwise, we say that it has been marked as *fully* verified.
4. Otherwise: We do not do anything. For instance, this may happen if we cannot determine that we have seen the same check in the cached snapshot.

As a last step, we replace the placeholder `?` in each statement **`assume`** `?` $\implies P$ with the conjunction of all relevant assumption variables, if none of the relevant assumption variables are definitely false and there was no error for the corresponding call in the cached implementation. Otherwise, we drop the statement.

Optimization for explicit assumptions within loops.

By default, loop bodies are verified modularly in Boogie. That is, on entry to a loop body, all variables that are modified within the body are "havocked" by assigning a non-deterministic value and the invariant is assumed. After the loop body, only the invariant remains to be checked.

For this reason, an assumption (e.g., as a result of a procedure call) that was made in the loop body when verifying the cached snapshot was neither used for verifying statements after the loop (provided there is no **`break`** statement in the loop) nor for verifying statements within the loop that precede the assignment to the corresponding assumption variable. To reproduce this behavior for the current snapshot, it is safe not to havoc assumption variables that would usually be havocked in this case. By doing so, such assumption variables usually remain true at that point unless the corresponding loop has previously been unrolled a number of times.

5.4 Evaluation

To evaluate the effectiveness of our caching techniques *in practice*, we recorded eight verification sessions during expert use of the Dafny IDE for regular development tasks. Those sessions were not scripted and therefore cover real workloads that such a tool faces when it is being used by a user to develop provably correct software. The sessions span a wide range of activities (including extension, maintenance, and refactoring) that are encountered when developing programs of several hundred lines. Sessions consist of up to 255 individual program snapshots (see Table 5.1) since the Dafny IDE automatically verifies the program as the user is editing it. To make this a pleasant experience for the user, the responsiveness of the tool is of paramount importance.

Table 5.1 clearly shows that this user experience could not be achieved without caching. The basic caching alone decreases the running times of the verifier tremendously (more than an order of magnitude for many sessions) and complementing it with fine-grained caching decreases them even more. This confirms the positive feedback that we received from users of the Dafny IDE, including members of the Ironclad project at Microsoft Research, whose codebase includes more than 30'000 lines of Dafny code [67]. Interestingly, caching turned out to have a more significant effect on the responsiveness of the tool than parallelization of verification tasks in Boogie using multiple SMT solver instances (see Sections 4.1 and 4.6).

Figure 5.6 sheds more light on why the basic caching is so effective by showing the priorities of the procedure implementations that are sent to the verifier for each snapshot in session 6: most of the procedure implementations do not need to be re-verified at all and only two implementations (originating from a single Dafny method) need to be verified for most snapshots. This data looks very similar for the other sessions and demonstrates that the basic caching benefits significantly from the modular verification approach in Dafny.

Generally, we can see occasional spikes with procedure implementations of low priority. For example, snapshot 3 consists of a change to a function that may affect all callers. In fact, due to the way that functions are handled, all *transitive* callers are affected, which is not the case for procedures. While in this case the basic caching needs to re-verify 11 procedure implementations from scratch, the fine-grained caching is able to mark 400 out of 971 checked statements in Boogie as fully verified. This reduces the running time from 28 seconds to 14 seconds and at the same time avoids a timeout (by default, 10 seconds per procedure implementation) for one of those procedure implementations.

Overall, Table 5.1 shows that the fine-grained caching performs even better than the basic

Session	Snapshots	Time (in Seconds)			Number of Timeouts		
		NC	**BC**	**FGC**	**NC**	**BC**	**FGC**
1	70	4'395.3	315.3	277.5	58	3	1
2	13	758.5	88.2	74.8	11	4	2
3	59	3'648.0	220.2	206.1	83	5	4
4	254	13'977.6	1'734.7	1'008.7	2	1	0
5	255	6'698.6	533.8	499.8	16	6	5
6	27	1'956.0	785.7	519.7	0	0	0
7	29	106.9	33.3	27.3	0	0	0
8	7	765.5	20.5	20.0	0	0	0

Table 5.1: Comparison of three configurations for verifying eight recorded IDE sessions: no caching (**NC**), basic caching (**BC**) and fine-grained caching (**FGC**). The second column shows the number of program snapshots per session. The next three columns show the running times for each configuration and the rightmost three columns show the number of timed-out procedure implementations for each configuration.

caching for all sessions (42 % faster for session 4 and on average 17 % faster compared to the basic caching). For session 8, there is no significant speedup even though the fine-grained caching is able to mark a large number of checks as verified. It seems that, in this case, most of the time is spent on verifying a single check (e.g., the postcondition of the edited method) that could not be marked as verified. Such cases can come up occasionally since the times that are needed for verifying different checks are usually not distributed uniformly.

Besides increasing responsiveness, caching helps in reducing the number of procedure implementations that fail to verify due to timeouts (see Table 5.1). Again, the basic caching avoids the majority of timeouts and the fine-grained caching avoids even more of them (between 17 % and 100 % less), which is not obvious given our program transformations. This additional reduction over the basic caching is due to the fact that Boogie is able to focus on fewer unverified or partially verified checks.

To provide a better indication of how much the fine-grained caching is able to reduce the verification effort, Figure 5.7 shows the number of checked statements for each snapshot in session 6 that were transformed when injecting cached verification results. This demonstrates that for many snapshots, more than half of the checks can be marked as fully verified or errors from the cached snapshot can be recycled (two errors each for snapshots 6 and 7 and one error each for snapshots 8 and 9).

At an early development stage, fewer checks were marked as verified since statement check-

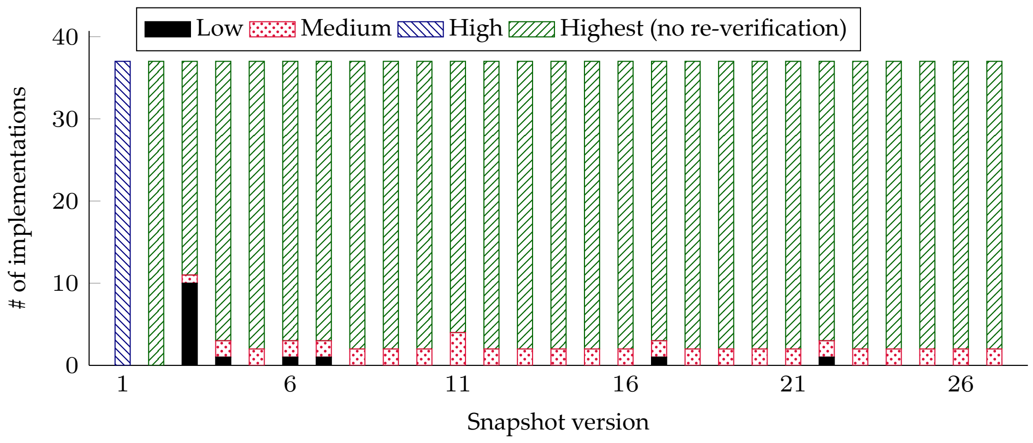


Figure 5.6: Priorities of procedure implementations for session 6. The bars show the number of procedure implementations of a given priority for each snapshot version. Most implementations are assigned the highest priority and do not need to be re-verified.

sums changed more often. It turned out that small changes in a Dafny program could result in significant changes to the corresponding Boogie program due to the way in which names (e.g., of auxiliary variables) were generated. After taking this into account during the translation of Dafny into Boogie, performance improved significantly.

5.5 Related Work

Caching is a widely-used technique for reusing information that was computed in the past. More specifically, there are several existing approaches for reusing results from previous runs of static analyzers, model checkers, program verifiers, and automatic test-case generation tools. Clousot [52], a static analyzer for .Net, uses caching to retrieve the results of previous runs of its cloud-based analysis service [3]. Unlike our fine-grained caching, it only reuses such results if a method itself did not change *and* if the specifications of all its callees did not change. Clousot also supports "verification modulo versions" [89], which uses conditions inferred for a previous version of a program to only report new errors for the current version. The SPARK tool set was also extended with light-weight support for caching of verification results [18]. Unlike in our approach, the caching happens on the granularity of the verification conditions that are sent to the solver and may happen on a dedicated caching server. The Why3 verification platform uses checksums to maintain program proofs in the form of *proof sessions* as the platform evolves (e.g., by generating different

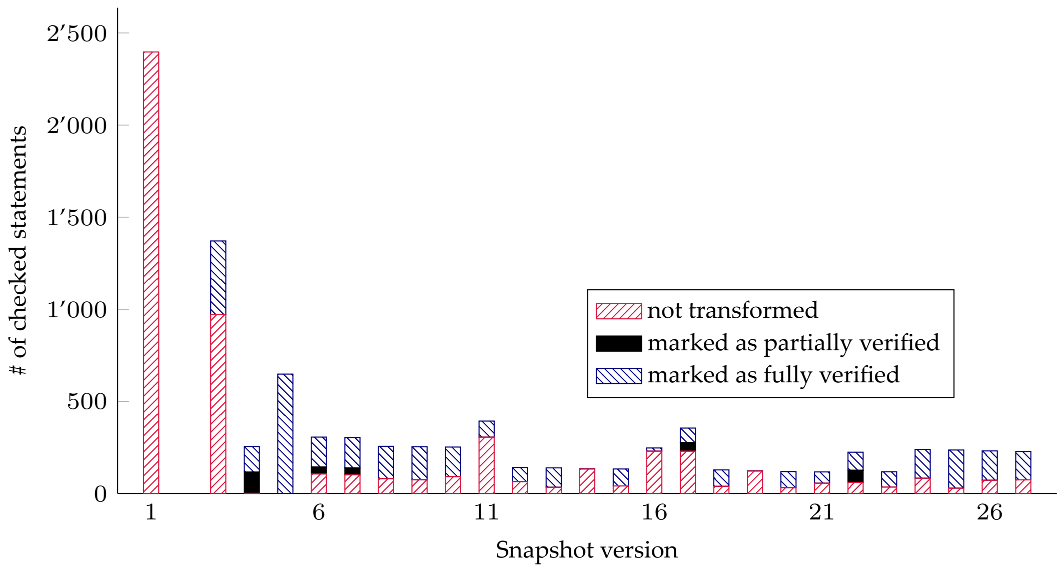


Figure 5.7: Transformed checked statements in session 6. The bars show the number of checked statements for each snapshot version that are marked as fully verified, partially verified, or not transformed at all. Additionally, a number of errors are recycled: two errors each for snapshots 6 and 7 and one error each for snapshots 8 and 9.

proof obligations) [15]. In particular, it matches goals from the existing proof with new goals using both checksums and *goal shapes*, a heuristic similarity measure.

Maintenance of proofs is particularly important for interactive proof assistants since proofs are largely constructed by users and, ideally, do not need to be changed once they have been completed. Such work has been done for the KIV [101] and KeY [72] tools. Grigore and Moskal [65] have worked on such techniques for proofs that were generated by SMT solvers to verify programs using ESC/Java.

There are several approaches for reusing information that was computed when running a non-modular tool on an earlier revision of a program. In the area of model checking, such information can consist of summaries computed using Craig interpolation [104], *derivation graphs* that record analysis progress [33], or parts of the reachable, abstract state space [68]. Even the precision of the analysis that was sufficient for analyzing an earlier program revision may be used later [12]. This can avoid the need for running a more precise analysis if the earlier version of a program could be verified using a less precise analysis. Similarly, it can avoid running an analysis that is not precise enough. Work on incremental compositional dynamic test generation [61] presents techniques for determining if function summaries that were obtained for an earlier version of a program can be safely reused when performing symbolic execution on the current version of the program.

Regression verification [107] is another area that developed techniques for reusing information that was collected during runs of a tool on earlier versions of a program. Unlike in our approach, the goal is to check if the behavior of the latest version of a program is equivalent to the one of an earlier version, much like in regression testing.

In spirit, our caching scheme is an instance of a truth maintenance system [47]. However, the mechanisms used are quite different. For example, a truth maintenance system records justifications for each fact, whereas our caching scheme tracks snapshots of the programs that give rise to proof obligations, not the proofs of the proof obligations themselves.

5.6 Summary

We have presented two effective techniques for using cached verification results to improve the responsiveness and performance of the Dafny IDE. Both techniques are crucial for providing *design-time feedback* at every keystroke to users of the IDE, much like background spell checkers.

The key novelties of our technique are its use of checksums for determining which parts of a program are affected by a change and how a program is instrumented with cached information to focus the verification effort. In particular, we use explicit assumptions to express the conditions under which we can reuse cached verification results. We have designed our technique to work on the level of an intermediate verification language. This makes it immediately usable for other verifiers that use the Boogie verification engine (e.g., AutoProof [112], Chalice [80], or VCC [30]) and should make possible to adopt by other intermediate verification languages, such as Silver [70] and Why3 [16].

Chapter 6

Conclusions and Future Work

We have presented a technique for expressing partial verification results of static analyzers by annotating programs using two new language constructs. Traditionally, a static analyzer is thought of as a tool that produces some output—usually errors or invariants—from an input program. Alternately, our approach makes it possible to see a static analyzer as a tool that produces an annotated program that expresses its results from an input program. The fact that, in this setting, both input and output are programs makes it both natural and convenient for other tools to make efficient use of that output.

We have designed both `assumed` statements and partially-justified `assume` statements such that they can be expressed in terms of assignments and regular `assume` statements. This allows most state-of-the-art program analysis tools to benefit from partial verification results *immediately* and without changes to their inner workings. The importance of this should not be underestimated since it ensures that the approach can actually be adopted *in practice* by a wide range of real-world program analysis tools—ranging from static analyzers, over deductive verifiers, to test case generation tools. To demonstrate this flexibility we have shown several novel use cases for partial verification results in test case generation tools, static analyzers, or inference tools. In particular, we have shown how to soundly express the results of static analyzers with sources of deliberate unsoundness. Until now, even though such analyzers are very common in practice, it was not possible to share their results with other tools in a practical and sound way.

We believe that integration of several program analysis tools is key to obtaining more mature and more usable tools and to making advances in program analysis and formal methods available to a wider audience more rapidly. In particular, we hope that the community will be able to build new and better tools more easily by building on and integrating existing tools. We have seen similar developments in several other research areas. For instance, many of the recent advances in SMT solvers have been facilitated by the Nelson-Oppen method [96] for combining decision procedures. The same holds for the combination of abstract domains [36] in the area of abstract interpretation.

Besides the use cases for expressing partial verification results that we outlined in Section 2.3, this work has opened up several other promising directions for future work. First, we would like to explore how our technique for expressing partial verification results can be used to capture more advanced properties. For instance, this might include temporal properties or properties about concurrent programs. The latter might allow us to run a static analyzer for sequential programs to analyze a concurrent program by capturing what implicit assumptions it makes that do not necessarily hold in a concurrent setting. The same idea could be pushed further by running a static analyzer for concurrent programs that assumes sequential consistency to analyze concurrent programs for weaker memory models.

Second, we would like to explore how a static analyzer could track its unsound assumptions more precisely. This would help in focusing the effort of any subsequent program analysis tools even further. There are two sides to this problem: tracking more precisely if an assumption (e.g., about arithmetic overflow) is actually unsound at a given program point and tracking where that assumption is actually used by the analyzer. For instance, for tools based on abstract interpretation, one could try to use taint analysis to mark values in the abstract state as tainted if they depend on some unsound assumptions. Alternatively, as suggested by Shuvendu Lahiri, one could inspect the unsatisfiable core that is produced by the SMT solver to determine if certain unsound assumptions are not relevant for verifying a program using a deductive verifier.

Third, one could use our approach from Chapter 3 to come up with new kinds of sources of deliberate unsoundness. In fact, this might make it possible to build tunable static analyzers that offer different levels of unsoundness. A user might then run the same analyzer with decreasing levels of unsoundness to catch shallow bugs very early and still look for more intricate bugs efficiently by building on previous partial verification results. Such a static analyzer would seem particularly well-suited for being used inside an IDE since an analyzer with a high level of unsoundness could provide early feedback to the user very efficiently. In such a setting, it would also be interesting to investigate how the order of different static analysis runs affects the partial verification results. This is particularly relevant if static analyzers are incomparable with respect to their level of unsoundness and this could shed light on the conditions under which the results are independent of the order in which analyzers are run.

Fourth, it would be interesting to explore ways for “fixing” programs to avoid explicit assumptions in the static analyzers that are used or to ensure that those explicit assumptions always hold. As a result, more properties could be fully verified in the “fixed” program. We have explored a specific instance of this in Section 2.3.3 by inferring sufficient preconditions that justify the explicit assumptions. However, more involved techniques could target specific sources of deliberate unsoundness. For instance, some explicit assumptions about overflow could be avoided by rewriting the program to use arbitrary-precision integers or by rewriting the expression that may overflow [88, 90]. Similarly, some explicit assumptions about aliasing could be avoided by using value types instead of reference types. Such work might inspire new programming language designs that can be more easily analyzed in a sound way.

Finally, one could explore more systematic ways for automatically identifying sources of deliberate unsoundness in static analyzers and for proving their absence. This would help designers of static analyzers in making sure that no sources of deliberate unsoundness were missed and that all of them were soundly captured. To this end, it might be interesting to phrase the problem of identifying a source of unsoundness as an application of logical abduction with the goal of inferring a condition that explains the (possibly unsound) results of a static analyzer.

Bibliography

[1] Thomas Ball, Brian Hackett, Shuvendu K. Lahiri, Shaz Qadeer, and Julien Vanegue. Towards scalable modular checking of user-defined properties. In *VSTTE*, volume 6217 of *LNCS*, pages 1–24. Springer, 2010.

[2] Thomas Ball and Sriram K. Rajamani. Boolean programs: A model and process for software analysis. Technical Report MSR–TR–2000–14, Microsoft Research, 2000.

[3] Michael Barnett, Mehdi Bouaziz, Manuel Fähndrich, and Francesco Logozzo. A case for static analyzers in the cloud. In *Workshop on Bytecode Semantics, Verification, Analysis, and Transformation (Bytecode 2013)*, 2013.

[4] 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 *FMCO*, volume 4111 of *LNCS*, pages 364–387. Springer, 2005.

[5] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and verification: The Spec# experience. *Communications of the ACM*, 54(6):81–91, 2011.

[6] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 82–87. ACM, 2005.

[7] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007.

[8] Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles-Henri Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *CACM*, 53:66–75, 2010.

[9] Frédéric Besson, Pierre-Emmanuel Cornilleau, and Thomas P. Jensen. Result certification of static program analysers with automated theorem provers. In *VSTTE*, volume 8164 of *LNCS*, pages 304–325. Springer, 2013.

[10] Dirk Beyer, Thomas A. Henzinger, M. Erkan Keremoglu, and Philipp Wendler. Conditional model checking. *CoRR*, abs/1109.6926, 2011.

[11] Dirk Beyer, Thomas A. Henzinger, M. Erkan Keremoglu, and Philipp Wendler. Conditional model checking: A technique to pass information between verifiers. In *FSE*, pages 57–67. ACM, 2012.

[12] Dirk Beyer, Stefan Löwe, Evgeny Novikov, Andreas Stahlbauer, and Philipp Wendler. Precision reuse for efficient regression verification. In *ESEC/SIGSOFT FSE*, pages 389–399. ACM, 2013.

[13] Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In *ITP*, volume 6172 of *LNCS*, pages 131–146. Springer, 2010.

[14] Sandrine Blazy, Vincent Laporte, André Maroneze, and David Pichardie. Formal verification of a C value analysis based on abstract interpretation. In *SAS*, volume 7935 of *LNCS*, pages 324–344. Springer, 2013.

[15] François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. Preserving user proofs across specification changes. In *VSTTE*, volume 8164 of *LNCS*, pages 191–201. Springer, 2013.

[16] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, 2011.

[17] Thorsten Bormer, Marc Brockschmidt, Dino Distefano, Gidon Ernst, Jean-Christophe Filliâtre, Radu Grigore, Marieke Huisman, Vladimir Klebanov, Claude Marché, Rosemary Monahan, Wojciech Mostowski, Nadia Polikarpova, Christoph Scheben, Gerhard Schellhorn, Bogdan Tofan, Julian Tschannen, and Mattias Ulbrich. The COST IC0701 verification competition 2011. In *FoVeOOS*, volume 7421 of *LNCS*, pages 3–21. Springer, 2011.

[18] Martin Brain and Florian Schanda. A lightweight technique for distributed and incremental program verification. In *VSTTE*, volume 7152 of *LNCS*, pages 114–129. Springer, 2012.

[19] Cristian Cadar and Dawson R. Engler. Execution generated test cases: How to make systems code crash itself. In *SPIN*, volume 3639 of *LNCS*, pages 2–23. Springer, 2005.

[20] Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of C programs. In *NFM*, volume 6617 of *LNCS*, pages 459–465. Springer, 2011.

[21] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, pages 289–300. ACM, 2009.

[22] Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. The SANTE tool: Value analysis, program slicing and test generation for C program debugging. In *TAP*, volume 6706 of *LNCS*, pages 78–83. Springer, 2011.

[23] Maria Christakis. *Narrowing the gap between verification and systematic testing*. PhD thesis, ETH Zurich, 2015.

[24] Maria Christakis, Patrick Emmisberger, and Peter Müller. Dynamic test generation with static fields and initializers. In *RV*, volume 8734 of *LNCS*, pages 269–284. Springer, 2014.

[25] Maria Christakis, Peter Müller, and Valentin Wüstholz. Collaborative verification and testing with explicit assumptions. In *FM*, volume 7436 of *LNCS*, pages 132–146. Springer, 2012.

[26] Maria Christakis, Peter Müller, and Valentin Wüstholz. Synthesizing parameterized unit tests to detect object invariant violations. In *SEFM*, volume 8702 of *LNCS*, pages 65–80. Springer, 2014.

[27] Maria Christakis, Peter Müller, and Valentin Wüstholz. An experimental evaluation of deliberate unsoundness in a static program analyzer. In *VMCAI*, volume 8931 of *LNCS*, pages 336–354. Springer, 2015.

[28] Maria Christakis, Peter Müller, and Valentin Wüstholz. Guiding dynamic symbolic execution toward unverified program executions. Technical report, ETH Zurich, May 2015.

[29] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.

[30] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical

system for verifying concurrent C. In *TPHOLs*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009.

[31] David R. Cok. Improved usability and performance of SMT solvers for debugging specifications. *STTT*, 12(6):467–481, November 2010.

[32] David R. Cok. OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse. In *Workshop on Formal Integrated Development Environment (F-IDE)*, volume 149 of *EPTCS*, pages 79–92, April 2014.

[33] Christopher L. Conway, Kedar S. Namjoshi, Dennis Dams, and Stephen A. Edwards. Incremental algorithms for inter-procedural analysis of safety properties. In *CAV*, volume 3576 of *LNCS*, pages 449–461. Springer, 2005.

[34] Loïc Correnson, Pascal Cuoq, Florent Kirchner, Virgile Prevosto, Armand Puccetti, Julien Signoles, and Boris Yakobowski. *Frama-C User Manual*, 2011. http://frama-c.com/.

[35] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.

[36] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282. ACM, 1979.

[37] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. Automatic inference of necessary preconditions. In *VMCAI*, volume 7737 of *LNCS*, pages 128–148. Springer, 2013.

[38] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTREÉ analyzer. In *ESOP*, volume 3444 of *LNCS*, pages 21–30. Springer, 2005.

[39] Patrick Cousot, Radhia Cousot, Jérôme Feret, Antoine Miné, Laurent Mauborgne, David Monniaux, and Xavier Rival. Varieties of static analyzers: A comparison with ASTRÉE. In *TASE*, pages 3–20. IEEE Computer Society, 2007.

[40] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96. ACM, 1978.

[41] Christoph Csallner and Yannis Smaragdakis. JCrasher: An automatic robustness tester for Java. *SPE*, 34:1025–1050, 2004.

[42] Christoph Csallner and Yannis Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *ICSE*, pages 422–431. ACM, 2005.

[43] Christoph Csallner, Yannis Smaragdakis, and Tao Xie. DSD-Crasher: A hybrid analysis tool for bug finding. *TOSEM*, 17:1–37, 2008.

[44] Mike Czech, Marie-Christine Jakobs, and Heike Wehrheim. Just test what you cannot verify! In *FASE*, volume 9033 of *LNCS*, pages 100–114. Springer, 2015.

[45] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *CACM*, 18:453–457, 1975.

[46] Isil Dillig, Thomas Dillig, and Alex Aiken. Automated error diagnosis using abductive inference. In *PLD*, pages 181–192. ACM, 2012.

[47] Jon Doyle. A truth maintenance system. *Artificial Intelligence*, 12(3):231–272, November 1979.

[48] Claire Dross, Pavlos Efstathopoulos, David Lesens, David Mentré, and Yannick Moy. Rail, space, security: Three case studies for SPARK 2014. In *ERTS*^{2}, 2014.

[49] Sophia Drossopoulou, Adrian Francalanza, Peter Müller, and Alexander J. Summers. A unified framework for verification techniques for object invariants. In *ECOOP*, volume 5142 of *LNCS*, pages 412–437. Springer, 2008.

[50] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69:35–45, 2007.

[51] Manuel Fähndrich, Michael Barnett, and Francesco Logozzo. Embedded contract languages. In *SAC*, pages 2103–2110. ACM, 2010.

[52] Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In *FoVeOOS*, volume 6528 of *LNCS*, pages 10–30, 2010.

[53] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV*, volume 4590 of *LNCS*, pages 173–177. Springer, 2007.

[54] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In *ESOP*, volume 7792 of *LNCS*, pages 125–128. Springer, 2013.

[55] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In *FME*, volume 2021 of *LNCS*, pages 500–517. Springer, 2001.

[56] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI*, pages 234–245. ACM, 2002.

[57] Xi Ge, Kunal Taneja, Tao Xie, and Nikolai Tillmann. DyTa: Dynamic symbolic execution guided with static verification results. In *ICSE*, pages 992–994. ACM, 2011.

[58] Roberto Giacobazzi. Abductive analysis of modular logic programs. *J. Log. Comput.*, 8(4):457–483, 1998.

[59] Roberto Giacobazzi, Francesco Logozzo, and Francesco Ranzato. Analyzing program analyses. In *POPL*, pages 261–273. ACM, 2015.

[60] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *PLDI*, pages 213–223. ACM, 2005.

[61] Patrice Godefroid, Shuvendu K. Lahiri, and Cindy Rubio-González. Statically validating must summaries for incremental compositional dynamic test generation. In *SAS*, volume 6887 of *LNCS*, pages 112–128. Springer, 2011.

[62] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *NDSS*. The Internet Society, 2008.

[63] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. SAGE: whitebox fuzzing for security testing. *Commun. ACM*, 55(3):40–44, 2012.

[64] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *CAV*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.

[65] Radu Grigore and Michał Moskal. Edit and verify. In *Workshop on First-Order Theorem Proving (FTP)*, 2007.

[66] Klaus Havelund and Thomas Pressburger. Model checking JAVA programs using JAVA PathFinder. *STTT*, 2:366–381, 2000.

[67] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated full-system verification. In *OSDI*, pages 165–181. USENIX Association, 2014.

[68] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Marco A. A. Sanvido. Extreme model checking. In *Verification: Theory and Practice*, volume 2772 of *LNCS*, pages 332–358. Springer, 2003.

[69] C. A. R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1:271–281, 1972.

[70] Uri Juhasz, Ioannis T. Kassios, Peter Müller, Milos Novacek, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. Technical report, ETH Zurich, 2014.

[71] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[72] Vladimir Klebanov. *Extending the Reach and Power of Deductive Program Verification*. PhD thesis, Department of Computer Science, Universität Koblenz-Landau, 2009.

[73] Jason Koenig and K. Rustan M. Leino. Getting started with Dafny: A guide. In *Software Safety and Security: Tools for Analysis and Verification*, volume 33 of *NATO Science for Peace and Security Series D: Information and Communication Security*, pages 152–181. IOS Press, 2012. Summer School Marktoberdorf 2011 lecture notes.

[74] Claire Le Goues, K. Rustan M. Leino, and Michał Moskal. The Boogie Verification Debugger (tool paper). In *SEFM*, volume 7041 of *LNCS*, pages 407–414. Springer, 2011.

[75] K. Rustan M. Leino. Specification and verification of object-oriented software. In *Engineering Methods and Tools for Software Safety and Security*, volume 22 of *NATO Science for Peace and Security Series D: Information and Communication Security*, pages 231–266. IOS Press, 2009. Summer School Marktoberdorf 2008 lecture notes.

[76] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR-16*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.

[77] K. Rustan M. Leino. Automating induction with an SMT solver. In *VMCAI*, volume 7148 of *LNCS*, pages 315–331. Springer, 2012.

[78] K. Rustan M. Leino and Michal Moskal. Co-induction simply: Automatic co-inductive proofs in a program verifier. In *FM*, volume 8442 of *LNCS*, pages 382–398. Springer, 2014.

[79] K. Rustan M. Leino, Michał Moskal, and Wolfram Schulte. Verification condition splitting. Technical report, Microsoft Research, October 2008. Manuscript KRML 192.

[80] K. Rustan M. Leino and Peter Müller. A basis for verifying multi-threaded programs. In *ESOP*, volume 5502 of *LNCS*, pages 378–393. Springer, 2009.

[81] K. Rustan M. Leino and Philipp Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In *TACAS*, volume 6015 of *LNCS*, pages 312–327. Springer, 2010.

[82] K. Rustan M. Leino and Valentin Wüstholz. The Dafny integrated development environment. In *Workshop on Formal Integrated Development Environment (F-IDE)*, volume 149 of *EPTCS*, pages 3–15, April 2014.

[83] K. Rustan M. Leino and Valentin Wüstholz. Fine-grained caching of verification results. In *CAV*, volume 9206 of *LNCS*, pages 380–397. Springer, 2015.

[84] Boyang Li, Isil Dillig, Thomas Dillig, Kenneth L. McMillan, and Mooly Sagiv. Synthesis of circular compositional program proofs via abduction. In *TACAS*, volume 7795 of *LNCS*, pages 370–384. Springer, 2013.

[85] Percy Liang, Omer Tripp, Mayur Naik, and Mooly Sagiv. A dynamic evaluation of the precision of static heap abstractions. In *OOPSLA*, pages 411–427. ACM, 2010.

[86] Benjamin Livshits and Shuvendu K. Lahiri. In defense of probabilistic static analysis. In *APPROX*, 2014.

[87] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *CACM*, 58:44–46, 2015.

[88] Francesco Logozzo, Michael Barnett, Manuel Fähndrich, Patrick Cousot, and Radhia Cousot. A semantic integrated development environment. In Gary T. Leavens, editor, *SPLASH*, pages 15–16. ACM, 2012.

[89] Francesco Logozzo, Shuvendu K. Lahiri, Manuel Fähndrich, and Sam Blackshear. Verification modulo versions: Towards usable verification. In *PLDI*, pages 294–304. ACM, 2014.

[90] Francesco Logozzo and Matthieu Martel. Automatic repair of overflowing expressions with abstract interpretation. In *Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday*, volume 129 of *EPTCS*, pages 341–357, 2013.

[91] Bertrand Meyer. *Object-Oriented Software Construction, 1st edition*. Prentice-Hall, 1988.

[92] Bertrand Meyer, Arno Fiva, Ilinca Ciupa, Andreas Leitner, Yi Wei, and Emmanuel Stapf. Programs that test themselves. *IEEE Computer*, 42(9):46–55, 2009.

[93] Jan Midtgaard, Michael D. Adams, and Matthew Might. A structural soundness proof for Shivers's escape technique: A case for Galois connections. In *SAS*, volume 7460 of *LNCS*, pages 352–369. Springer, 2012.

[94] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. *Sci. Comput. Program.*, 62:253–286, October 2006.

[95] Peter Müller and Joseph N. Ruskiewicz. Using debuggers to understand failed verification attempts. In *FM*, volume 6664 of *LNCS*, pages 73–87. Springer, 2011.

[96] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.

[97] Aditya V. Nori, Sriram K. Rajamani, Saideep Tetali, and Aditya V. Thakur. The YOGI project: Software property checking via static analysis and testing. In *TACAS*, volume 5505 of *LNCS*, pages 178–181. Springer, 2009.

[98] Corina S. Pasareanu and Willem Visser. Verification of Java programs using symbolic execution and invariant generation. In *Workshop on Model Checking Software (SPIN)*, volume 2989 of *LNCS*, pages 164–181. Springer, 2004.

[99] Charles Sanders Peirce. *Collected Papers of Charles Sanders Peirce, Volumes V and VI: Pragmatism and Pragmaticism and Scientific Metaphysics*. Belknap Press, 1935.

[100] Nadia Polikarpova, Carlo A. Furia, and Scott West. To run what no one has run before: Executing an intermediate verification language. In *RV*, volume 8174 of *LNCS*, pages 251–268. Springer, 2013.

[101] Wolfgang Reif and Kurt Stenzel. Reuse of proofs in software verification. In *FSTTCS*, volume 761 of *LNCS*, pages 284–293. Springer, 1993.

[102] Herbert Schorr and William M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Commun. ACM*, 10(8):501–506, 1967.

[103] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *ESEC*, pages 263–272. ACM, 2005.

[104] Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. Incremental upgrade checking by means of interpolation-based function summaries. In *FMCAD*, pages 114–121. IEEE, 2012.

[105] Manu Sridharan and Stephen J. Fink. The complexity of Andersen's analysis in practice. In *SAS*, volume 5673 of *LNCS*, pages 205–221. Springer, 2009.

[106] Christian Sternagel. Getting started with Isabelle/jEdit. In *Isabelle Users Workshop (IUW)*, 2012.

[107] Ofer Strichman and Benny Godlin. Regression verification - A practical way to verify programs. In *VSTTE*, volume 4171 of *LNCS*, pages 496–501. Springer, 2005.

[108] Alexander J. Summers and Peter Müller. Freedom before commitment: A lightweight type system for object initialisation. In *OOPSLA*, pages 1013–1032. ACM, 2011.

[109] Nikolai Tillmann and Jonathan de Halleux. Pex—White box test generation for .NET. In *TAP*, volume 4966 of *LNCS*, pages 134–153. Springer, 2008.

[110] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. Usable verification of object-oriented programs by combining static and dynamic techniques. In *SEFM*, volume 7041 of *LNCS*, pages 382–398. Springer, 2011.

[111] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. Verifying Eiffel programs with Boogie. In *Workshop on Intermediate Verification Languages (BOOGIE 2011)*, pages 14–26, 2011.

[112] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Nadia Polikarpova. AutoProof: Auto-active functional verification of object-oriented programs. In *TACAS*, volume 9035 of *LNCS*, pages 566–580. Springer, 2015.

[113] Makarius Wenzel. Asynchronous proof processing with Isabelle/Scala and Isabelle/jEdit. In *9th International Workshop On User Interfaces for Theorem Provers (UITP 2010)*, ENTCS. Elsevier, July 2010.