December 2018

Building Deductive Program Verifiers

Lecture Notes

Peter MÜLLER [a]
[a] *Department of Computer Science, ETH Zurich, Switzerland*

Abstract. Deductive program verifiers attempt to construct a proof that a given program satisfies a given specification. Their implementations reflect the semantics of the programming language and the specification language, and often include elaborate proof search strategies to automate verification. Each of these components is intricate, which makes building a verifier from scratch complex and costly.

In these lecture notes, we will present an approach to build program verifiers as a sequence of translations from the source language and specification via intermediate languages down to a logic for which automatic solvers exist. This architecture reduces the overall complexity by dividing the verification process into simpler, well-defined tasks, and enables the reuse of essential elements of a program verifier such as parts of the proof search, specification inference, and counterexample generation. We will use the intermediate verification language Viper to demonstrate how to encode interesting verification problems.

Keywords. verification condition generation, intermediate verification language, permission logics, hyperproperties, product programs, Nagini, Viper

1. Introduction

Ensuring the correctness and security of software systems is becoming increasingly challenging. Testing has always been limited to checking just a small subset of the possible program executions. In the omnipresence of concurrency (for instance, in software that runs on multicore processors or in data centers) and event-based systems (such as applications running on mobile devices), testing is largely insufficient. It is, thus, useful to complement or replace testing with static verification techniques such as static program analysis [9], model checking [7], or deductive verification [19]. These techniques can formally prove correctness and security properties for all executions of a program, that is, for all possible inputs, thread schedules, event interactions, attacker behaviors, etc.

Static program analysis, model checking, and deductive verification strike different trade-offs between automation, expressiveness, and modularity. In these lecture notes, we focus on deductive verification, which requires more user input than the other techniques, but allows one to prove complex properties and enables modular verification [28]. Modularity is important for scalability, to reduce the re-verification effort during software maintenance, and to give guarantees for indi-

vidual program components such as libraries. Deductive program verification employs a program logic such as Hoare logic [19] or separation logic [33] to construct a mathematical proof that a given program satisfies its specification.

Program verifiers are tools that automate (parts of) the proof search, typically by reducing verification to a set of verification conditions, logical formulas whose validity implies the correctness of the program and which can be checked by automatic or interactive theorem provers. This reduction is often performed by encoding a program, its specification, and the program logic into an intermediate verification language. Programs in the intermediate language are typically not (efficiently) executable. Their correctness implies the correctness of the original program. Implementing a verifier via an intermediate verification language has two major advantages over a monolithic architecture. First, it allows one to reuse large parts of the tool infrastructure; all components that operate on the intermediate language or further downstream can be reused across multiple program verifiers just like an optimizer and a code generator for an compiler intermediate language can be reused across multiple compilers. Second, human-readable intermediate verification languages greatly simplify the prototyping of verification techniques and tools, as well as debugging.

There are several mature intermediate verification languages and corresponding tool infrastructures. Boogie [22] offers a simple procedural language and tool support for verification condition generation, bounded verification [20], and debugging of verification failures [21]. Why's language [16] has a functional flavor; its verifier targets a wide range of automatic provers. Viper [30] facilitates the verification of proofs in logics similar to separation logic, targeting especially heap-manipulating and concurrent programs. All three intermediate languages are widely used. For instance, Boogie is at the core of verifiers such as Chalice [26], Corral [20], Dafny [23], Spec# [25,2], and VCC [8]. Why powers for instance Frama-C and Krakatoa [15], and Viper is used by Nagini [11], Prusti [1], and VerCors [5].

In these lecture notes, we will use Viper. However, especially the concepts introduced in the earlier sections apply similarly to other intermediate verification languages. Fig. 1 shows the architecture of the Viper verification infrastructure. We will introduce the Viper intermediate language together with examples later, but refer to an overview of its design [30] and the tutorial [13] for details. Viper provides two backend verifiers. The symbolic execution verifier [35] reasons about heap manipulations internally and uses the SMT solver Z3 [27] for other aspects of verification, such as arithmetic. The verification condition generator is itself implemented via a translation to the intermediate language Boogie, which ultimately also targets Z3. Both tools can be tried online at `viper.ethz.ch`; the entire infrastructure is available as open-source implementation.

Outline. Sec. 2 introduces the foundations of verification condition generation. Sec. 3 shows how to verify hyperproperties (properties the relate two or more program executions) via an encoding onto an intermediate language. In Sec. 4, we explain how to encode the verification of heap-manipulating programs using a flavor of separation logic. Sec. 5 extends this encoding to recursive data structures. We briefly discuss the development of frontend verifiers in Sec. 6, and conclude in Sec. 7.

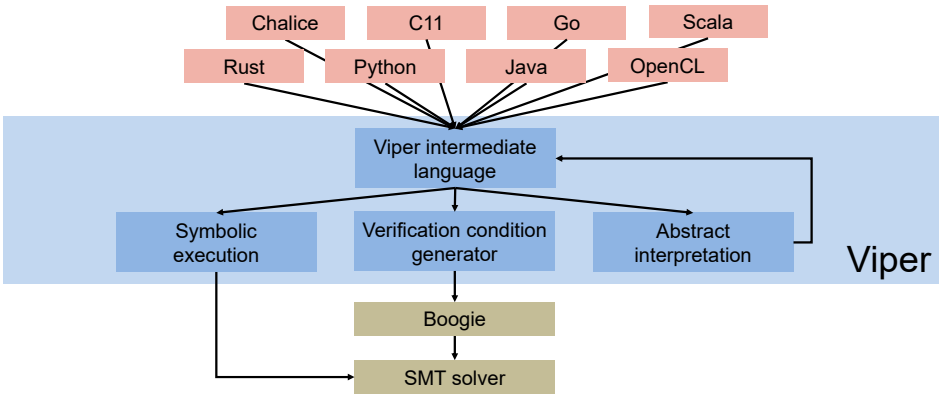

Figure 1. Architecture of the Viper verification infrastructure. The blue boxes depict the main components of the architecture: the Viper intermediate language, two verification backends (based on symbolic execution and verification condition generation, resp.), and an abstract interpreter to infer auxiliary specifications. Gray boxes are external components, and orange boxes show the languages for which existing verifiers are implemented via a translation into Viper. The verifiers in the upper row (Chalice, C11, Go, Scala) are prototypes, whereas the others (Rust, Python, Java, OpenCL) are fairly mature tools.

2. Verification Condition Generation

Throughout these lecture notes, we will encode more and more complex programs and properties into simpler intermediate representations. As foundation for this chain of translations, we employ Dijkstra's guarded-commands language [10]. In this section, we introduce the necessary background of guarded commands, explain how to encode statements into this language, and illustrate the resulting verification approach on an example.

2.1. Guarded Commands

We use a guarded-commands language with the following syntax:

$$
\begin{aligned}
S ::= {} & x{:=}e \\
\mid {} & \texttt{havoc}\ x \\
\mid {} & \texttt{assert}\ P \\
\mid {} & \texttt{assume}\ P \\
\mid {} & S;S \\
\mid {} & S \,[\!]\, S
\end{aligned}
$$

where x ranges over variables, e denotes a side-effect free expression, and P denotes an assertion (a first-order formula over program variables). Guarded commands include assignments, assignments of non-deterministic values to variables, assertions, assumptions, sequential composition, and non-deterministic choice. The execution of a guarded command from an initial state fails if the condition of an assertion evaluates to false, it is infeasible if the condition of an assumption evaluates to false, and otherwise succeeds. We say that a guarded command is *correct* if all feasible executions succeed, that is, no execution fails.

Correctness of a guarded command S can be verified by proving the validity of the verification condition $wp(S, true)$, where $wp(S, Q)$ denotes the weakest precondition of guarded command S w.r.t. assertion Q and is defined as follows ($Q[e/x]$ denotes Q with e substituted for x):

$$\begin{aligned} wp(x{:=}e, Q) &= Q[e/x] \\ wp(\texttt{havoc}\ x, Q) &= \forall x \cdot Q \\ wp(\texttt{assert}\ P, Q) &= P \wedge Q \\ wp(\texttt{assume}\ P, Q) &= P \Rightarrow Q \\ wp(S_1; S_2, Q) &= wp(S_1, wp(S_2, Q)) \\ wp(S_1 \,[\!]\, S_2, Q) &= wp(S_1, Q) \wedge wp(S_2, Q) \end{aligned}$$

Many verification problems require mathematical theories such as arithmetic and set theory. For those theories that are not natively supported by the underlying theorem prover, intermediate verification languages allow programmers to define them via sorts, uninterpreted function symbols, and axioms. The conjunction of these axioms forms a so-called *background predicate*, which may be assumed for any given verification task. So for a background predicate BP, the verification of a guarded command S means proving the validity of:

$$BP \Rightarrow wp(S, true)$$

2.2. Encoding of Statements

Guarded commands offer a simple core language, for which verification condition generation is straightforward. Other statements can be encoded conveniently into guarded commands, as we show next.

Conditional statements. A conditional statement

$$\texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{end}$$

is encoded as

$$(\texttt{assume}\ e;\ [\![S_1]\!]) \,[\!]\, (\texttt{assume}\ \neg e;\ [\![S_2]\!])$$

where $[\![S]\!]$ denotes the encoding of statement S (we omit the encoding of expressions for simplicity). Intuitively, the resulting guarded command is correct if S_1 is correct *if* e holds (otherwise the left-hand side of the non-deterministic choice is infeasible) and if S_2 is correct *if* e does not hold (otherwise the right-hand side is infeasible). This is exactly the verification condition required for a conditional statement, and formalized by the verification condition:

$$(e \Rightarrow wp([\![S_1]\!], Q)) \wedge (\neg e \Rightarrow wp([\![S_2]\!], Q))$$

Loops. The verification of loops requires a suitable loop invariant, that is, an assertion that holds before the loop and after each loop iteration. A loop invariant represents the inductive argument needed to reason about an unknown number of loop iterations. We require programmers to provide loop invariants manually

through suitable annotations in the code. However, techniques for the automatic inference of invariants, typically via fixpoint iteration, exist [9].

A loop of the form

```
while(e) invariant P begin S end
```

is encoded as

```
assert P;
havoc x_i; assume P;
(assume e; [[S]]; assert P; assume false)
[]
assume ¬e
```

This encoding first asserts the loop invariant before the loop. Instead of performing an iteration (which would require a fixpoint computation in the *wp* computation), it simulates an arbitrary loop execution. For this purpose, we assign non-deterministic values to all variables x_i that are assigned in the loop body S (the so-called loop targets) and assume the loop invariant. This step removes all previous knowledge about these variables, except that they satisfy the loop invariant. The subsequent non-deterministic choice models the two possible behaviors of the loop. If the loop condition holds, we execute the loop body S and prove that it preserves the invariant P. The subsequent `assume false` ensures that any subsequent code verifies trivially; this is needed because we encode the termination of the loop separately, in the second branch of the non-deterministic choice. Here, we assume that the loop condition e does not hold.

Verifying the encoding of a loop ensures that the loop is correct if it terminates. Proof obligations that enforce termination need to be encoded explicitly, as we discuss later.

Procedures. Modular verification techniques require specifications for procedures and verify calls using the specification of the callee instead of its implementation. This approach is compatible with information hiding, allows one to verify calls where the callee implementation is unknown (for instance, abstract methods, dynamically-bound methods, or library methods), and avoids re-verification of callers when the implementation of a callee changes.

Procedures are typically specified using pre- and postconditions. Callers need to establish the precondition and may assume the postcondition after the call. In turn, procedure bodies may assume that their precondition holds upon entry and must establish the postcondition upon termination. Consider a procedure declaration

```
procedure p(x) returns r
  requires P
  ensures Q
begin S end
```

where x is the (only) formal parameter, r is the result variable, P is the precondition, and Q is the postcondition. In languages without global state such as global

variables or a heap memory, correctness of this procedure can be encoded into guarded commands as follows:

```
assume P
⟦S⟧
assert Q
```

A a call y `:=` $p(e)$ is encoded as (where z is a fresh variable):

```
z := e
assert P[z/x]
havoc y
assume Q[z/x, y/r]
```

The substitutions replace the parameter and result variable in the pre- and postcondition by the actual argument and right-hand side variable, resp. The temporary variable z is needed since e may refer to y. The havoc operation reflects that the call updates variable y and, thus, all prior information about its value is no longer valid. Properties of the new value of y are conveyed via p's postcondition. We will discuss in Sec. 4 how to encode procedures and calls in the presence of a heap memory, where we need to reflect the potential side effects of a call on heap locations.

2.3. Example

The example in Fig. 2 illustrates the concepts used so far. This Viper procedure (called *method* in Viper) implements the first challenge of the VerifyThis 2011 verification competition (see `www.pm.inf.ethz.ch/research/verifythis/Archive/2011.html`). Method `maxSeq` computes the index of the maximum of a non-empty sequence of integers. The postcondition states that the result `x` is a valid index and that the value at position `x` is at least as large as all other values in the sequence. The loop invariant states that values to the left of `x` and to the right of `y` are less than or equal to the value at position `x`. Consequently, when the loop terminates, we have `x==y` and, thus, `x` is the index of the maximum.

`Seq` is a built-in generic datatype. It is encoded via uninterpreted function symbols and axioms, which are part of the background predicate used to verify any Viper method.

Viper does not require termination of methods and loops. However, it is possible to encode termination arguments explicitly via additional assertions in the code. In this example, we use `y - x` as ranking function. To ensure that its value ranges over a well-founded set, we prove in line 18 that it is non-negative. Termination is then guaranteed by the fact that each loop iteration decreases the value of the ranking function, which we assert in line 25.

3. Verification of Hyperproperties

With the technique introduced so far, we can prove properties for individual executions of a program such as functional correctness and termination. Other im-

```
1  method maxSeq(s: Seq[Int]) returns (x: Int)
2    requires 0 < |s|
3    ensures 0 <= x && x < |s|
4    ensures forall i: Int :: 0 <= i && i < |s| ==> s[i] <= s[x]
5  {
6    x := 0
7    var y: Int := |s| - 1
8
9    while(x != y)
10     invariant 0 <= x
11     invariant 0 <= y && y < |s|
12     invariant x <= y
13     invariant forall i: Int ::
14                     (0 <= i && i < x || y < i && i < |s|)
15                     ==> (s[i] <= s[x] || s[i] <= s[y])
16   {
17     var measure: Int  := y - x   // termination
18     assert 0 <= measure          // termination
19     if(s[x] <= s[y])
20     {
21       x := x + 1
22     } else {
23       y := y - 1
24     }
25     assert y - x < measure       // termination
26   }
27 }
```

Figure 2. A Viper method that computes the index of the maximum of a non-empty sequence of integers. The pre- and postcondition express the functional behavior of the method; termination is encoded manually through local assertions.

portant properties relate multiple executions. For instance, determinism requires that two executions starting from the same initial state terminate in the same final state. Properties of multiple program executions are called *hyperproperties* and include for instance monotonicity, non-interference [17] (which is used to prove secure information flow [34]), or read effects [24].

Hyperproperties can be verified via relational program logics [4,39]. However, these logics are difficult to automate and require dedicated tool support. An alternative is to construct a so-called *product program* [3,12] that encodes two or more executions of the original program into a single execution of the product. This product program can be expressed in an intermediate verification language and verified using the approach introduced above.

We consider *modular product programs* [12] here, which enable the modular verification of hyperproperties, and focus on the encoding of two executions. A generalization to arbitrary numbers is trivial. The basic construction is simple. To encode the state space of two program executions, we introduce two variables x_1 and x_2 for each variable x of the original program. Since the control flow of the two executions of the original program may differ, we introduce two boolean *activation*

variables p_1 and p_2 that reflect whether each of the executions is currently active. Both variables are initially true.

An assignment x`:=`e in the original program is then encoded as two conditional assignments, which update the variables of the product program *if* the corresponding execution is active:

`if(`p_1`) { `x_1`:=`e_1` }`
`if(`p_2`) { `x_2`:=`e_2` }`

where e_i is the expression e with all occurrences of a variable y replaced by y_i. A conditional statement `if(`e`) { `S_1` } else { `S_2` }` is encoded by introducing fresh activation variables that reflect which execution enters the then- and the else-branch, resp.:

`var` p_1^{true} `:=` $p_1 \wedge e_1$
`var` p_2^{true} `:=` $p_2 \wedge e_2$
`var` p_1^{false} `:=` $p_1 \wedge \neg e_1$
`var` p_2^{false} `:=` $p_2 \wedge \neg e_2$
$[\![S_1]\!](p_1^{true}, p_2^{true})$
$[\![S_2]\!](p_1^{false}, p_2^{false})$

where $[\![S]\!](p_1, p_2)$ denotes the product construction for statement S with activation variables p_1 and p_2.

A key feature of modular product programs is that their construction *does not* duplicate loops and method calls. This feature allows us to use loop invariants and method specifications that relate both executions of the original program and, thereby, enables modular verification. A loop `while(`e`) invariant` P `{` S `}` is encoded as follows:

```
while(p1 ∧ e1 ∨ p2 ∧ e2)
  invariant [[P]](p1, p2)
{
  var p1^true := p1 ∧ e1
  var p2^true := p2 ∧ e2
  [[S]](p1^true, p2^true)
}
```

The product loop iterates as long as one of the two executions is active and its loop condition is satisfied. However, the loop body is executed only for active executions.

Modular product programs support both classical and relational assertions. A classical assertion P is encoded as $(p_1 \Rightarrow P_1) \wedge (p_2 \Rightarrow P_2)$, that is, it must hold for each active execution. Relational specifications may relate both executions of the program. A relational assertion R is encoded as $p_1 \wedge p_2 \Rightarrow R$, that is, it must hold if both executions are active (the variables of an inactive execution do not have meaningful values).

Methods are encoded by duplicating parameters and results, and adding two extra parameters for the activation variables. A call then passes the values of the

```
1  method maxDet(s1: Seq[Int], s2: Seq[Int], p1: Bool, p2: Bool)
2                                              returns (x1: Int, x2: Int)
3    requires p1 ==> 0 < |s1|
4    requires p2 ==> 0 < |s2|
5    ensures p1 && p2 ==> (s1 == s2 ==> x1 == x2)
6  {
7    var y1: Int
8    var y2: Int
9    if(p1) { x1 := 0 }
10   if(p2) { x2 := 0 }
11   if(p1) { y1 := |s1| - 1 }
12   if(p2) { y2 := |s2| - 1 }
13
14   while(p1 && x1 != y1 || p2 && x2 != y2)
15     invariant p1 ==> 0 <= x1 && 0 <= y1 && y1 < |s1| && x1 <= y1
16     invariant p2 ==> 0 <= x2 && 0 <= y2 && y2 < |s2| && x2 <= y2
17     invariant p1 && p2 ==> (s1 == s2 ==> x1 == x2 && y1 == y2)
18   {
19     var pw1: Bool := p1 && x1 != y1
20     var pw2: Bool := p2 && x2 != y2
21
22     var pt1: Bool := pw1 && s1[x1] <= s1[y1]
23     var pt2: Bool := pw2 && s2[x2] <= s2[y2]
24     var pf1: Bool := pw1 && !(s1[x1] <= s1[y1])
25     var pf2: Bool := pw2 && !(s2[x2] <= s2[y2])
26
27     if(pt1) { x1 := x1 + 1 }
28     if(pt2) { x2 := x2 + 1 }
29     if(pf1) { y1 := y1 - 1 }
30     if(pf2) { y2 := y2 - 1 }
31   }
32 }
```

Figure 3. Modular product program for the method from Fig. 2. The classical preconditions ensure that sequence accesses are within bounds. The relational postcondition expresses determinism: for equal parameter values, we will get equal results. We omit the termination checks for simplicity.

caller's activation variables to the callee to ensure that the body of the callee method is executed only if the corresponding execution is active.

Fig. 3 shows the product program for the example in Fig. 2. The classical preconditions (and the corresponding loop invariants) ensure that sequence accesses are within bounds. The relational postcondition and loop invariant express determinism: for equal parameter values, we will get equal results in both method executions.

Modular product programs allow one to use off-the-shelf verifiers to verify hyperproperties. They can be used to verify even advanced non-interference properties including declassification and the absence of termination leaks [12].

4. Verification of Heap-Manipulating Programs

In this section, we present a verification technique for heap-manipulating programs and show how it can be encoded into the language introduced so far.

4.1. Access Permissions

The main verification challenge for heap-manipulating programs is *framing*: how to preserve information about heap data structures across heap changes, in particular, across method calls. For this purpose, we introduce the notion of an *access permission* (or permission for short), which facilitates static verification, but is not present during program execution. We associate an access permission with each heap location. This permission is created when the heap location is allocated. Permissions are held by method executions; a method execution may access a heap location only if it holds the corresponding permission. Permissions may be transferred between method executions, but cannot be duplicated or forged. Consequently, there is at most one permission available for each location. While one method holds the permission, no other method can have it to modify the location, which enables framing.

To distinguish read and write accesses, it is useful to support *fractional permissions* [6], where a permission can be split into several fractions, and the fractions can be re-combined to obtain a full permission. Writing to a memory location requires full permission, whereas any non-zero fraction permits reading.

Let us assume a heap that consists of objects with fields. We can encode heaps and permissions as two mathematical maps (defined as part of the background predicate) that map reference-field pairs to values and rational numbers in $[0;1]$, resp. We call the permission map a *mask*. Using this encoding, a field read y `:=` $x.f$ is encoded as:

```
assert x ≠ null
assert Mask[x, f] > 0
y := Heap[x, f]
```

Analogously, we can encode a field update $x.f$ `:=` e as:

```
assert x ≠ null
assert Mask[x, f] = 1
Heap := Heap[x, f → e]
```

Permissions are transferred between methods upon calls and when a call returns. Which permissions to transfer is specified in the method specification via accessibility predicates of the form `acc(`$x.f$`,` p`)`, where p is the required fraction. The transfer is encoded via two auxiliary operations on assertions. *Exhaling* an assertion P is done in three steps: (1) It asserts that all permissions required by P are available in the current mask and that all logical constraints in P hold; if not, verification fails. (2) It removes the transferred permissions from the mask. (3) It havocs all memory locations to which no permission is held, to reflect that other methods may use the permission to update those locations and, thereby, invalidate any knowledge about them. Conversely, *inhaling* an assertion P requires

```
field val: Int
define read(a,i)
  slot(a,i).val

domain IArray {
  function slot(a: IArray, i: Int): Ref
  function len(a: IArray): Int
  function first(r: Ref): IArray
  function second(r: Ref): Int

  axiom all_diff {
    forall a: IArray, i: Int :: { slot(a,i) }
      first(slot(a,i)) == a && second(slot(a,i)) == i
  }

  axiom len_nonneg {
    forall a: IArray :: { len(a) }
      len(a) >= 0
  }
}
```

Figure 4. A Viper field, macro, and background predicate to encode mutable integer arrays. The term `{ slot(a,i) }` is a matching pattern used by the SMT solver to instantiate the universal quantifier.

two steps: (1) It adds the transferred permissions to the mask. (2) It assumes that all logical constraints in P hold. Both operations are defined inductively over the syntax of assertions; we omit a formal encoding here for simplicity.

With these auxiliary operations, we can adapt the encoding of procedure declarations and calls presented earlier. Instead of asserting and assuming pre- and postconditions, they are exhaled and inhaled, resp. For instance, upon a call, the caller exhales the precondition (to transfer permissions to the callee) and then inhales the postcondition (to transfer permissions back). The havoc that happens in step 3 of the exhale reflects the potential side effects of the callee method.

Permissions are the basis behind modern program logics such as separation logic [33] and implicit dynamic frames [36]. They are applicable to a wide range of verification problems. In a concurrent setting, they ensure data race freedom [31]. If one thread holds full permission to write to a memory location, other threads hold no permission and can, thus, neither read nor write. Nevertheless, fractional permissions enable concurrent reading. Permissions have also been used to verify fine-grained concurrency, even on weak memory models [38,37].

4.2. Example

To illustrate the use of permissions, we discuss a variation of the example from Fig. 2 that operates on a mutable array instead of a mathematical sequence.

Viper does not support arrays natively, but they can be encoded easily as part of the background predicate, as shown in Fig. 4. We model each array location as

```
1  method maxArray(a: IArray) returns (x: Int)
2    requires 0 < len(a)
3    requires forall i: Int :: 0<=i && i<len(a) ==> acc(read(a,i), 1/2)
4    ensures 0 <= x && x < len(a)
5    ensures forall i: Int :: 0<=i && i<len(a) ==> acc(read(a,i), 1/2)
6    ensures forall i: Int :: 0<=i && i<len(a)
7            ==> read(a,i) <= read(a,x)
8  {
9    x := 0
10   var y: Int := len(a) - 1
11
12   while(x != y)
13     invariant 0 <= x && x <= y && y < len(a)
14     invariant forall i: Int :: 0<=i && i<len(a)
15               ==> acc(read(a,i), 1/2)
16     invariant forall i: Int :: (0<=i && i<x || y<i && i<len(a))
17               ==> (read(a,i) <= read(a,x) || read(a,i) <= read(a,y))
18   {
19     var measure: Int := y - x    // termination
20     assert 0 <= measure          // termination
21     if(read(a,x) <= read(a,y))
22     {
23       x := x + 1
24     } else {
25       y := y - 1
26     }
27     assert y - x < measure       // termination
28   }
29 }
```

Figure 5. A variation of the example from Fig. 2 that operates on a mutable array instead of a mathematical sequence. The fractional permissions in the method specification and loop invariant allow the method to read the array elements, but prevent modifications. They allow callers to conclude that the array is not changed by the method.

a separate reference, which is yielded by function `slot`. The value of this location can then be accessed via a predefined `val` field of that reference. To simplify the notation, we define a macro `read` that is parametric in the array and index, and expands into the access expression. The first axiom states that `slot` is injective. It is expressed via two inverse functions, which yield better performance in the SMT solver than a naive formulation of injectivity.

Fig. 5 shows the Viper encoding of the example. The fractional permissions in the method specification and loop invariant allow the method to read the array elements, but prevent modifications. Therefore, they enable framing: callers may conclude that the array is not changed by the method. The assertions use universal quantification over permissions, a feature called *iterated separating conjunction* [33], which is supported by Viper [29].

```
1  field left: Ref
2  field right: Ref
3  field val: Int
4
5  predicate tree(this: Ref) {
6    acc(this.left) && acc(this.right) && acc(this.val) &&
7    (this.left != null ==> tree(this.left)) &&
8    (this.right != null ==> tree(this.right))
9  }
10
11 function elems(this: Ref): Multiset[Int]
12   requires tree(this)
13 {
14   unfolding tree(this) in
15     Multiset(this.val) union
16     (this.left != null ? elems(this.left) : Multiset[Int]()) union
17     (this.right != null ? elems(this.right) : Multiset[Int]())
18 }
```

Figure 6. A recursive tree predicate and a heap-dependent function to obtain the multiset of integers stored in a tree.

5. Verification of Recursive Data Structures

Iterated separating conjunction lets us specify permissions to a statically-unknown number of memory locations. It is especially useful for random-access data structures such as arrays and maps. For other data structures, in particular recursive ones, we can specify permissions using (possibly recursive) predicates [32,18].

The predicate `tree` in Fig. 6 provides permission to the fields of the argument reference. If either of the two subtrees is non-null, it includes a recursive predicate instance for this tree. Consequently, the predicate represents the permissions to all locations in the entire tree. A predicate may also constrain the values of heap locations whose permissions it contains, for instance, to express invariants of data structures.

Just like permissions, predicate instances are held by method executions and transferred through exhale and inhale operations. Predicates with one parameter such as the `tree` predicate can be stored in the mask. For instance, $Mask[x, P] = 1$ expresses that the current state contains the predicate $P(x)$. Predicates with more arguments require higher-dimensional masks.

Recursive definitions are generally tricky for automatic provers because provers need to be prevented from unfolding them indefinitely, leading to non-termination in the proof search. To avoid this problem, many tools require programmers to unfold and fold predicates manually through annotations in the code. For this purpose, Viper provides a statement `unfold` $P(x)$, which exhales the predicate instance $P(x)$ and inhales the body of the predicate. Conversely, `fold` $P(x)$ exhales the body and inhales the predicate instance. an expression `unfolding` $P(x)$ `in` e temporarily unfolds $P(x)$, evaluates expression e, and then re-folds $P(x)$.

```
1  method maxTree(t: Ref) returns (m: Int)
2    requires tree(t)
3    ensures tree(t)
4    ensures 0 < (m in elems(t))
5    ensures forall e: Int :: 0 < (e in elems(t)) ==> e <= m
6    ensures elems(t) == old(elems(t))
7  {
8    var tmp: Int
9    var measure: Int := |elems(t)|      // termination
10   unfold tree(t)
11   m := t.val
12   if(t.left != null) {
13     assert |elems(t.left)| < measure  // termination
14     tmp := maxTree(t.left)
15     if(m < tmp) { m := tmp }
16   }
17   if(t.right != null) {
18     tmp := maxTree(t.right)
19     if(m < tmp) { m := tmp }
20   }
21   fold tree(t)
22 }
```

Figure 7. A Viper method to compute the maximum in a tree of integers. Permissions to the fields of the tree nodes are represented via the recursive predicate `tree`. The functional behavior is specified in terms of the heap-dependent function `elems`. Both are defined in Fig. 6.

The example in Fig. 7 illustrates these features. It solves the second challenge of the VerifyThis 2011 verification competition. Method `maxTree` computes the maximum in a tree of integers. It takes an instance of the `tree` predicate from its caller and returns it after the call. In order to get access to the fields of the tree node, the method unfolds the predicate in line 10. Before terminating, it re-folds the predicate in line 21.

The functional behavior of `maxTree` is specified in terms of the heap-dependent function `elems`, which is defined in Fig. 6. Heap-dependent functions have a precondition that requires permission to the heap locations accessed by the function. They automatically return all permissions to their caller; a postcondition that provides permissions is neither necessary nor allowed.

Heap-dependent functions are encoded via an uninterpreted function symbol and two axioms. A *definitional axiom* relates the uninterpreted function symbol to the definition of the heap-dependent function. A *framing axiom* expresses that changing heap locations whose permission is not mentioned in the function precondition cannot affect the function value.

The `elems` function traverses the tree recursively and yields the multiset of values stored in the tree nodes. The postcondition of method `maxTree` uses this function to say that (1) the returned value is in the multiset of values stored in the tree, (2) it is no smaller than all other tree elements, and (3) the method does not affect the values stored in the tree. The latter could also be achieved by

taking only a fraction of the predicate instance `tree(t)`, allowing callers to hold on to another fraction. The size of the multiset of values is also used as ranking function to prove termination of the recursive method.

6. Building Frontend Verifiers

In the previous sections, we have introduced various features of the Viper language and explained how to encode them via translations into simpler intermediate languages down to guarded commands and then to an SMT solver. Since Viper is itself an intermediate verification language, frontend tools use it to encode complex verification problems. In this section, we will discuss a Python version of the tree example form the previous section and its verification in Nagini [11]. The code and specification are shown in Fig. 8.

Nagini requires programs to be statically typed using the mypy type system. It encodes predicate and function definitions as Python methods with annotations `@Predicate` and `@Pure`, resp. Pure methods must be side-effect free. Specifications are expressed as calls to predefined Python methods, which do nothing at run time, but are interpreted by Nagini. This design is adopted from .NET Code Contracts [14] and allows programmers to add annotations without extending the Python syntax.

One such annotation is the `MustTerminate` precondition, which expresses that the method must terminate, and which provides a ranking function that is then encoded via assertions on the Viper level as shown in Fig. 7. Tanslating the Python example from Fig. 8 results in essentially the Viper program from Fig. 7. However, the actual encoding produced by the Nagini verifier is much more complex and includes, for instance, a comprehensive formalization of Python's type system.

7. Conclusion

Many modern program verifiers are implemented as a sequence of translations into simpler intermediate verification languages. In these lecture notes, we showed how to encode a range of verification techniques into a simpler guarded-commands language, for which verification condition generation is straightforward. In particular, we showed how to encode access permissions, which allow one to verify heap-manipulating and concurrent programs.

A downside of building verifiers through translations is that error messages for verification failures need to be translated back from the lowest abstraction level to the frontend tool to provide meaningful feedback to programmers. Another drawback is that all translations are part of the trusted codebase, that is, errors in those translations may compromise soundness of the verification. Extracting foundational proofs from translation-based verifiers is an interesting direction for future work.

Acknowledgement. We thank Marco Eilers for his help with the Nagini example.

```
1  from nagini_contracts.contracts import *
2  from nagini_contracts.obligations import MustTerminate
3  from typing import Optional, List
4
5  class Tree:
6    def __init__(self, left: Optional['Tree'], right: Optional['Tree'],
7                 val: int) -> None:
8      self.left = left
9      self.right = right
10     self.val = val
11
12 @Predicate
13 def tree(self: Tree) -> bool:
14   return (Acc(self.left) and Acc(self.right) and Acc(self.val) and
15           Implies(self.left is not None, tree(self.left)) and
16           Implies(self.right is not None, tree(self.right)))
17
18 @Pure
19 def elems(self: Tree) -> MSet[int]:
20   Requires(tree(self))
21   Ensures(len(Result()) > 0)
22   empty = MSet()  # type: MSet[int]
23   return Unfolding(tree(self), MSet(self.val) +
24          (elems(self.left) if self.left is not None else empty) +
25          (elems(self.right) if self.right is not None else empty))
26
27 def maxTree(t: Tree) -> int:
28   Requires(tree(t))
29   Requires(MustTerminate(len(elems(t))))
30   Ensures(tree(t))
31   Ensures(0 < elems(t).num(Result()))
32   Ensures(Forall(int, lambda e: Implies(0 < elems(t).num(e),
33                                         e <= Result())))
34   Ensures(elems(t) == Old(elems(t)))
35   Unfold(tree(t))
36   res = t.val
37   if t.left is not None:
38     tmp = maxTree(t.left)
39     if res < tmp:
40       res = tmp
41
42   if t.right is not None:
43     tmp = maxTree(t.right)
44     if res < tmp:
45       res = tmp
46   Fold(tree(t))
47   return res
```

Figure 8. A Nagini version of the tree example from Figs. 6 and 7. Predicate and function definitions are encoded as Python methods with annotations `@Predicate` and `@Pure`, resp. Method specifications and loop invariants are expressed using calls to designated Python methods.

References

[1] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. Leveraging Rust types for modular specification and verification. Technical report, ETH Zurich, 2018. `https://doi.org/10.3929/ethz-b-000311092`.

[2] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: The Spec# experience. *Communications of the ACM*, 54(6):81–91, June 2011.

[3] G. Barthe, J. M. Crespo, and C. Kunz. Relational verification using product programs. In M. J. Butler and W. Schulte, editors, *Formal Methods (FM)*, volume 6664 of *LNCS*, pages 200–214. Springer, 2011.

[4] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *Principles of Programming Languages (POPL)*, pages 14–25. ACM, 2004.

[5] S. Blom and M. Huisman. The VerCors tool for verification of concurrent programs. In C. B. Jones, P. Pihlajasaari, and J. Sun, editors, *Formal Methods (FM)*, volume 8442 of *LNCS*, pages 127–131. Springer, 2014.

[6] J. Boyland. Checking interference with fractional permissions. In *Static Analysis Symposium (SAS)*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.

[7] E. M. Clarke, O. Grumberg, D. Kroening, D. Peled, and H. Veith. *Model checking*. MIT Press, 2018.

[8] E. Cohen, M. Moskal, W. Schulte, and S. Tobies. Local verification of global invariants in concurrent programs. In T. Touili, B. Cook, and P. Jackson, editors, *Computer Aided Verification (CAV)*, volume 6174 of *LNCS*, pages 480–494. Springer, 2010.

[9] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL)*, pages 238–252. ACM, 1977.

[10] E. W. Dijkstra. Guarded commands, non-determinancy and a calculus for the derivation of programs. In F. L. Bauer and K. Samelson, editors, *Language Hierarchies and Interfaces*, volume 46 of *LNCS*, pages 111–124. Springer, 1975.

[11] M. Eilers and P. Müller. Nagini: A static verifier for Python. In H. Chockler and G. Weissenbacher, editors, *Computer Aided Verification (CAV)*, volume 10982 of *LNCS*, pages 596–603. Springer, 2018.

[12] M. Eilers, P. Müller, and S. Hitz. Modular product programs. In A. Ahmed, editor, *European Symposium on Programming (ESOP)*, volume 10801 of *LNCS*, pages 502–529. Springer, 2018.

[13] ETH Zurich. *Viper Tutorial*, 2018. `viper.ethz.ch/tutorial/`.

[14] M. Fähndrich, M. Barnett, and F. Logozzo. Code contracts. `http://research.microsoft.com/contracts`, 2008.

[15] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *Computer Aided Verification (CAV)*, volume 4590 of *LNCS*, pages 173–177. Springer, 2007.

[16] J.-C. Filliâtre and A. Paskevich. Why3—where programs meet provers. In M. Felleisen and P. Gardner, editors, *European Symposium on Programming (ESOP)*, volume 7792 of *LNCS*, pages 125–128. Springer, 2013.

[17] J. A. Goguen and J. Meseguer. Security policies and security models. In *Security and Privacy*, pages 11–20. IEEE Computer Society, 1982.

[18] S. Heule, I. T. Kassios, P. Müller, and A. J. Summers. Verification condition generation for permission logics with abstract predicates and abstraction functions. In G. Castagna, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 7920 of *LNCS*, pages 451–476. Springer, 2013.

[19] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the CACM*, 12(10):576–580,583, 1969.

[20] A. Lal, S. Qadeer, and S. K. Lahiri. A solver for reachability modulo theories. In P. Madhusudan and S. A. Seshia, editors, *Computer Aided Verification (CAV)*, volume 7358 of *LNCS*, pages 427–443. Springer, 2012.

[21] C. Le Goues, K. R. M. Leino, and M. Moskal. The Boogie verification debugger (tool paper). In G. Barthe, A. Pardo, and G. Schneider, editors, *Software Engineering and*

Formal Methods (SEFM), volume 7041 of *LNCS*, pages 407–414. Springer, 2011.

[22] K. R. M. Leino. This is Boogie 2. `www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/`, 2008.

[23] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.

[24] K. R. M. Leino and P. Müller. Verification of equivalent-results methods. In S. Drossopoulou, editor, *European Symposium on Programming (ESOP)*, volume 4960 of *LNCS*, pages 307–321. Springer, 2008.

[25] K. R. M. Leino and P. Müller. Using the Spec# language, methodology, and tools to write bug-free programs. In P. Müller, editor, *Advanced Lectures on Software Engineering—LASER Summer School 2007/2008*, volume 6029 of *LNCS*, pages 91–139. Springer, 2010.

[26] K. R. M. Leino, P. Müller, and J. Smans. Verification of concurrent programs with Chalice. In A. Aldini, G. Barthe, and R. Gorrieri, editors, *Foundations of Security Analysis and Design V*, volume 5705 of *LNCS*, pages 195–222. Springer, 2009.

[27] L. Moura and N. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[28] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer, 2002.

[29] P. Müller, M. Schwerhoff, and A. J. Summers. Automatic verification of iterated separating conjunctions using symbolic execution. In S. Chaudhuri and A. Farzan, editors, *Computer Aided Verification (CAV)*, volume 9779 of *LNCS*, pages 405–425. Springer, 2016.

[30] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer, 2016.

[31] P. W. O'Hearn. Resources, concurrency and local reasoning. In P. Gardner and N. Yoshida, editors, *Concurrency Theory (CONCUR)*, volume 3170 of *LNCS*, pages 49–67. Springer, 2004.

[32] M. Parkinson and G. Bierman. Separation logic and abstraction. In J. Palsberg and M. Abadi, editors, *Principles of Programming Languages (POPL)*, pages 247–258. ACM, 2005.

[33] J. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS)*, pages 55–74. IEEE Computer Society, 2002.

[34] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[35] M. Schwerhoff. *Advancing Automated, Permission-Based Program Verification Using Symbolic Execution*. PhD thesis, ETH Zurich, 2016.

[36] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 5653 of *LNCS*, pages 148–172. Springer, 2009.

[37] A. J. Summers and P. Müller. Automating deductive verification for weak-memory programs. In D. Beyer and M. Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 10805 of *LNCS*, pages 190–209. Springer, 2018.

[38] V. Vafeiadis and C. Narayan. Relaxed separation logic: a program logic for C11 concurrency. In A. L. Hosking, P. T. Eugster, and C. V. Lopes, editors, *Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 867–884. ACM, 2013.

[39] H. Yang. Relational separation logic. *Theor. Comput. Sci.*, 375(1-3):308–334, 2007.