

Symbolic Execution for Implicit Dynamic Frames

Jan Smans Bart Jacobs Frank Piessens

Katholieke Universiteit Leuven, Belgium

{jans,bartj,frank}@cs.kuleuven.be

Abstract

The combination of verification condition generation and automated theorem proving is a well-known and widely used technique for checking whether a program satisfies its specification. However, for Java-like languages with shared mutable state and support for data abstraction, experience has shown that this technique has 3 important disadvantages: (1) it is slow, (2) it is unpredictable - small changes to the specification can have significant impact on verification time, and (3) it can be hard to determine why a verification condition fails.

In this paper, we propose an approach for automatically verifying whether a Java program satisfies its specification based on symbolic execution instead of verification condition generation. The specification language used is a JML-like language supporting the technique of implicit dynamic frames for dealing with data abstraction and framing. We implemented our approach in a verifier prototype. Experiments with the prototype indicate verification times are consistently low. By inspecting the symbolic state at each program point, the programmer can determine why verification failed.

Categories and Subject Descriptors D.2.4 [Software Engineering]: Program Verification

General Terms Languages, Theory, Verification

Keywords Implicit Dynamic Frames, Symbolic Execution

1. Introduction

Program verification is commonly based on Hoare-like program logics. Such program logics are formal systems that can derive correct (precondition, statement, postcondition) triples: the statement, when started in a state satisfying the precondition will end in a state satisfying the postcondition

(or never end at all if the logic does not care about termination).

To increase automation of the verification process, many state-of-the-art program verification tools [1, 2, 3, 4, 5, 6, 7] are based on Dijkstra's weakest preconditions calculus [8], an automatic calculus that computes the most general precondition for a given (statement, postcondition) pair. Given a program that is annotated by the programmer with method pre- and postconditions and loop invariants, the weakest precondition calculus makes it possible to automatically generate first-order logic sentences (the *verification conditions*) that together imply the correctness of the program. These verification conditions can be sent to an automated first-order theorem prover, typically a satisfiability-modulo-theory (SMT) solver [9, 10].

However, for Java-like languages with shared mutable state and support for data abstraction, experience has shown that this approach has 3 important disadvantages: (1) it is slow, (2) it is unpredictable - small changes to the specification can have significant impact on verification time, and (3) it can be hard to determine why a verification condition fails.

While it is clear that automatic program verification is an undecidable problem in general, and hence that one cannot expect any technique to work well in all cases, one important culprit for these disadvantages is the way in which verification condition generation handles the heap. Very convincing evidence of this fact has been presented by the separation logic community [11]. They have shown that some Java programs that cannot be handled well by classic automatic verifiers, can be verified automatically by using a special-purpose Hoare logic with built-in primitives for reasoning about the heap [12]. In particular, they can handle implementations of design patterns such as Iterator or Visitor. However, by switching to a special-purpose logic, one can no longer build on the many years of effort that have gone into building automatic first-order logic theorem provers.

The key contribution of this paper is that it shows how one can have the benefits of the separation logic approach to program verification in a classical setting. We build on the implicit dynamic frames program logic [7], a JML-like program logic that supports a variant of dynamic frames [13]. The verification algorithm for implicit dynamic frames reported on in [7] is based on verification condition gen-

eration, and hence suffers from the same three disadvantages mentioned above. This paper defines a new verification algorithm based on symbolic execution that has a significantly improved performance, increased predictability and better support for debugging specifications. The key innovation with respect to existing symbolic execution algorithms for JML-like languages [14] is the representation of the symbolic heap, and the handling of data abstraction (using heap *snapshots*). Compared to symbolic execution algorithms proposed for separation logic [12], our algorithm supports heap-dependent expressions and pure method calls in assertions.

The rest of this paper is structured as follows. First, in Section 2 we briefly recap the implicit dynamic frames approach. Section 3 describes the symbolic execution algorithm in detail, and Section 4 provides some examples. Section 5 summarizes the soundness proof of our algorithm, and Section 6 reports on our experience with a prototype implementation. Finally, Section 7 discusses related work, and Section 8 offers a conclusion.

2. Background: Implicit Dynamic Frames

The specification language we consider in this paper is a formal contract-based behavioral interface specification language similar to the Java Modeling Language (JML) [15, 16, 17] or Spec# [1]. It supports method pre- and postconditions, and inline assertions expressed in a programmer-friendly Java-like notation.

The main distinguishing feature of our specification language is its support for *implicit dynamic frames* [7], a recent technique to deal with framing and data abstraction in specifications. This section briefly summarizes the implicit dynamic frames approach. For a more detailed discussion, the reader can consult [7].

2.1 Framing

To reason modularly about a method invocation, one should not rely on the callee’s implementation, but only on its specification. For example, consider the code in Figure 1(b). To prove that the assertion at the end of the code snippet holds in every execution, one should only take into account *Cell*’s method contracts. However, the given contracts are too weak to prove the assertion. Indeed, *setX*’s implementation is allowed to change the state arbitrarily, as long as it ensures that **this.x** equals *v* on exit. In particular, the contract does not prevent *c₂.setX(10)* from modifying *c₁.x*.

To prove the assertion at the end of Figure 1(b), we must strengthen *Cell*’s method contracts. More specifically, the contracts should additionally specify an upper bound on the set of memory locations modifiable by the corresponding method. This problem is called the *frame problem*.

Various solutions to the frame problem have been proposed in the literature. The most well-known techniques use **modifies** or **assignable** clauses to specify what parts of the

```
class Cell {
  int x;

  Cell()
    ensures acc(this.x) ∧ this.x = 0;
  { this.x := 0; }

  void setX(int v)
    requires acc(this.x);
    ensures acc(this.x) ∧ this.x = v;
  { this.x := v; }
}
```

Figure 2. A revised version of the class *Cell* from Figure 1(a).

state a method is allowed to change. However, **modifies** clauses are hard to combine with data abstraction.

A more recent approach to framing was pioneered by separation logic [11, 18] and worked out in a JML-like setting by the *implicit dynamic frames* technique [7]. In these approaches, one can think of memory locations as being protected by permissions: a method may only access a memory location *o.f* if it has permission to do so. Preconditions then need to specify the permissions that a method requires. Since run time tests on permissions are not allowed, the only way in which a method implementation can modify a location is by requiring a corresponding permission in its precondition. In other words, framing information can be derived from the permissions required by the precondition. This approach to framing can be nicely combined with support for data abstraction. Subsection 2.2 will discuss how.

In the implicit dynamic frames approach, permissions are modeled by means of an *accessibility* predicate. More specifically, writing to or reading from a memory location *o.f* requires *o.f* to be accessible, denoted as **acc(o.f)**. As an example, consider the revised version of the class *Cell* of Figure 2. *setX* can only modify **this.x**, since its precondition only requires accessibility of **this.x**. Similarly, *Cell*’s constructor does not require access to any location, and can therefore only assign to fields of the new object.

The accessibility of a memory location can change over time. For example, when a new object is created, the fields of the new object become accessible to the constructor. To support scenarios where a method “captures” accessibility of a location, method invocations transfer accessibility from caller to callee: locations required to be accessible by the precondition of the callee are no longer accessible to the caller. The callee can however return the accessibility of locations in its postcondition. One can think of a method invocation as *consuming* the permissions required by its precondition, and *producing* the permissions asserted by its postcondition. For instance, the call *c₁.setX(5)* consumes the permission to access *c₁.x*, but upon return from the

```

class Cell {
  int x;

  Cell()
    ensures this.x = 0;
  { this.x := 0; }

  void setX(int v)
    ensures this.x = v;
  { this.x := v; }
}

```

(a)

```

Cell c1 := new Cell();
c1.setX(5); //A

Cell c2 := new Cell();
c2.setX(10);

assert c1.x = 5;

```

(b)

Figure 1. A class *Cell* and some client code.

call the permission is produced again, so that later calls to `setX()` are possible on c_1 .

Given the new method contracts for *Cell* of Figure 2 together with the rules for framing outlined above, we can now prove the assertion at the end of Figure 1(b). Informally, the reasoning is as follows. At program location *A*, the postcondition of $c_1.setX(5)$ holds: $c_1.x$ is accessible and its value is 5. Since c_2 's constructor does not require access to any location, it can modify neither the accessibility nor the value of any existing location. In particular, $c_1.x$ is still accessible and still holds 5. Similarly, the call $c_2.setX(10)$ only requires $c_2.x$ to be accessible, and hence $c_1.x$ is not affected. We may conclude that the assertion, $c_1.x = 5$, holds in any execution.

2.2 Data Abstraction

Data abstraction is crucial in the construction of modular programs, since it ensures that internal changes in one module do not propagate to other modules. However, the class *Cell* of Figure 2 and its specifications were not written with data abstraction in mind. More specifically, (1) client code must directly access the field x to query a *Cell* object's internal state and (2) *Cell*'s method contracts are not implementation-independent as they mention the internal field x . Any change to *Cell*'s implementation, such as renaming x to y , would break or at least oblige us to reconsider the correctness of client code.

Developers typically solve issue (1) by adding “getters” to their classes. For example, the class *Cell* of Figure 3(a) defines a method `getX` to query a *Cell*'s internal state. The method is marked **pure** to indicate it does not have side-effects. As shown in Figure 3(b), the assertion of Figure 1(b) can now be rephrased in terms of `getX`.

One of the strengths of the implicit dynamic frames approach is that it uses this same abstraction mechanism for specifications.

Two kinds of pure methods are supported in specifications: normal pure methods (annotated with **pure**) and predicates (annotated with **predicate**). Normal pure methods

can be used in implementation code and in contracts. Their body consists of a single return statement, returning an expression. They are typically used to abstract over data representation.

Predicates consists of a single return statement, returning an assertion. Since no run time tests on accessibility of locations is allowed, predicates should not be used by implementation code. They can only be called by contracts or from the bodies of other predicates. Predicates are typically used to represent invariants and to abstract over accessibility of memory locations. Predicates can not have preconditions.

A key challenge to support data abstraction in specifications is to keep track of how and when abstractions change value as the program state changes. In the implicit dynamic frames approach, this is handled as follows.

We can deduce from the precondition of a normal pure method an upper bound on the set of locations readable by that method: a pure method p can only read $o.f$ if p 's precondition requires $o.f$ to be accessible. In other words, the return value of a normal pure method only depends on locations required to be accessible by its precondition.

Predicates are required to be *self-framing*. That is, the return value of a predicate q only depends on locations that q itself requires to be accessible.

Given these properties of pure methods, we can now prove the assertion at the end of Figure 3(b). Informally, the reasoning is as follows. At program location *A*, the postcondition of $c_1.setX(5)$ holds: $c_1.valid()$ is true and $c_1.getX()$ returns 5. Because c_2 's constructor does not require access to any existing location, it can only modify fresh locations (i.e. c_2 's fields and fields of objects allocated within the constructor itself). Since $c_1.valid()$ only requires access to non-fresh locations, both its own return value and the return value of $c_1.getX()$ are not affected by c_2 's constructor. In addition, the set of memory locations required to be accessible by $c_1.valid()$ is disjoint from the set of locations required to be accessible by $c_2.valid()$, since the latter set only contains fresh locations. $c_2.setX()$ can only modify locations covered by $c_2.valid()$. The latter set of locations is disjoint

```

class Cell {
  int x;

  Cell()
    ensures valid() ∧ getX() = 0;
  { this.x := 0; }

  void setX(int v)
    requires valid();
    ensures valid() ∧ getX() = v;
  { this.x := v; }

  predicate valid()
  { return acc(this.x); }

  pure int getX()
    requires valid();
  { return this.x; }

  void swap(Cell c)
    requires c ≠ null;
    requires valid() * c.valid();
    ensures valid() * c.valid();
    ensures getX() = old(c.getX());
    ensures c.getX() = old(getX());
  { int i := x; x := c.getX(); c.setX(i); }
}

```

(a)

```

Cell c1 := new Cell();
c1.setX(5); //A

Cell c2 := new Cell();
c2.setX(10);

assert c1.getX() = 5;

```

(b)

Figure 3. A revised version of class *Cell* with data abstraction.

from $c_1.valid()$, hence the return values of $c_1.valid()$ and $c_1.getX()$ are not affected by $c_2.setX(10)$. We may conclude that the assertion, $c_1.getX() = 5$, holds in any execution.

Implicit dynamic frames specifications support separation logic’s separating conjunction ($*$). To illustrate the use of the separating conjunction, consider the method *swap* of Figure 3(a). *swap*’s precondition requires that the receiver and c are “separately” valid, i.e. that both $this.valid()$ and $c.valid()$ hold and that the set of locations required to be accessible by $this.valid()$ is disjoint from the set of locations required to be accessible by $c.valid()$. If we would have used a normal conjunction instead of a separating conjunction, we would not be able to prove $c.valid()$ holds after the assign-

ment to x . In particular, the separating conjunction ensures that $c.valid()$ does not depend on $this.x$.

3. Symbolic Execution

In this section, we show how one can automatically verify whether programs, such as the ones shown above, satisfy their specification. An object-oriented program satisfies its specification if each method body correctly implements the corresponding method contract. A method body correctly implements a contract if for all states and input parameters that satisfy the precondition, its execution results in a state which satisfies the postcondition. However, the set of initial states is infinite. How can we check for infinitely many initial states that a method body satisfies its contract?

The key idea of symbolic execution is to execute methods using symbolic instead of concrete values. Since a single symbolic state can represent an infinite number of concrete states, it suffices to show that execution of a method body always ends up in state satisfying the postcondition if it is started in an arbitrary symbolic state which satisfies the precondition.

The remainder of this section is structured as follows. We start by defining a small Java-like language with method contracts (Section 3.1). Symbolic execution is formalized with respect to this language. In Section 3.2, we describe the logic used for representing symbolic values and assumptions over such values. The notion of symbolic state is then introduced in Section 3.3. Symbolic evaluation of expressions and assertions and symbolic execution of statements in symbolic states is discussed in Section 3.4. Finally, we define what it means for a program to be valid in Section 3.5.

3.1 Language

We define the following sets. \mathcal{X} is the set of variable names with typical element x . \mathcal{C} is the set of class names with typical element C . \mathcal{F} is the set of field names with typical element f . \mathcal{M} is the set of mutator names with typical element m . \mathcal{P} is the set of pure method names with typical element p . \mathcal{Q} is the set of predicate method names with typical element q .

The syntax of programs is shown in Figure 4. Overlining indicates repetition. A program consists of a number of classes and a main routine \bar{s} . Each class declares a number of fields and methods. We distinguish three kinds of methods: mutators, pure methods and predicates. The body of a mutator consists of a list of statements, while the body of a pure method and a predicate consist of a single return statement, returning an expression, respectively an assertion. A statement s is either a local variable declaration, a variable update, a field update, a mutator invocation, an object creation, an if-then-else, assert, open, close or use statement. An expression e is either *null*, a variable, a field read, a pure method invocation, a conditional, old, opening or using expression. An assertion ϕ is **true**, **false**, an access assertion,

an equality, a separating conjunction, a predicate invocation, a conditional assertion or an untouched assertion.

<i>prog</i>	::=	$\overline{\text{class } \bar{s}}$
<i>class</i>	::=	class $C \{ \overline{\text{field } \bar{m}} \}$
<i>field</i>	::=	$C \ f;$
<i>method</i>	::=	$\text{mutator} \mid \text{pure} \mid \text{pred}$
<i>mutator</i>	::=	void $m(\overline{C \ x})$ requires ϕ ; ensures ϕ ; $\{ \bar{s} \}$
<i>pure</i>	::=	pure $C \ p(\overline{C \ x})$ requires ϕ ; $\{ \text{return } e; \}$
<i>pred</i>	::=	predicate $q(\overline{t \ x}) \{ \text{return } \phi; \}$
<i>s</i>	::=	$C \ x; \mid x := e; \mid e.f := e; \mid e.m(\bar{e}) \mid$ $x := \text{new } C; \mid \text{if}(e = e) \{ \bar{s} \} \text{ else } \{ \bar{s} \} \mid$ assert $e_1 = e_2; \mid \text{open } e.q(\bar{e}); \mid \text{close } e.q(\bar{e}); \mid$ use $e.p(\bar{e})$
<i>e</i>	::=	$\text{null} \mid x \mid e.f \mid e.p(\bar{e}) \mid e = e ? e : e \mid \text{old}(e) \mid$ opening $e.q(\bar{e})$ in $e \mid \text{using } e.p(\bar{e})$ in $e \mid$
ϕ	::=	true $\mid \text{false} \mid \text{acc}(e.f) \mid e = e \mid \phi * \phi \mid$ $e.q(\bar{e}) \mid e = e ? \phi : \phi \mid \text{untouched}(\phi)$

Figure 4. Syntax of a small Java-like language.

A separating conjunction $\phi_1 * \phi_2$ holds only if both conjuncts hold and if ϕ_1 and ϕ_2 demand access to disjoint parts of the heap. For example, $\text{acc}(\text{this}.x) * \text{acc}(\text{this}.x)$ is equivalent to **false**, since both conjuncts demand access to the same location. **old** expressions and **untouched** assertions may only be used in postconditions. **old**(e) represents e 's value on entry to the method. **untouched**(ϕ) holds only if all locations for which ϕ demands access have the same value in the pre and post-state heap. For example, **untouched**($\text{acc}(\text{this}.x)$) holds only if the location $\text{this}.x$ holds the same value on entry and on exit to the method. **open**, **close**, **use**, **opening** and **using** are ghost operations. Ghost operations have no effect at run-time, but are only used during symbolic execution. **open**, **opening** and **close** are used to switch between a predicate and its definition (similar to the way **pack** and **unpack** expose, respectively hide the invariant in the Boogie methodology [1]). **use** and **using** allow the verifier to assume that an invocation of a pure method equals evaluating the method body (similar to the implementation axiom described in [7]). We explain the purpose of the ghost operations in more detail in Sections 3.3 and 3.4.

Section 4 shows a number of programs, including *Cell*, written in the language described above.

3.2 Logic

In our symbolic execution, we represent symbolic values by first-order terms and assumptions over those symbolic values by first-order formulas. More specifically, we target a multi-sorted, first-order logic with equality. A term t is either a variable, a function application or an if-then-else term. A formula ψ is either *true*, *false*, an equality between terms, a

conjunction, a disjunction, a negation or a quantification.

t	::=	$x \mid f(\bar{t}) \mid \text{ite}(\psi, t, t)$
ψ	::=	$\text{true} \mid \text{false} \mid t = t \mid \psi \wedge \psi \mid \psi \vee \psi \mid \neg \psi \mid \forall \bar{x} \bullet \psi$

Functions with arity zero are called constants. We omit the trailing parenthesis for constants.

3.2.1 Signature

Each term in the logic has a corresponding sort. The sorts are the following: *ref*, the sort of references and *snapshot*, the sort of heap snapshots. In the remainder of this paper, we omit sorts whenever they are clear from the context.

The signature of the logic contains a number of built-in functions and a number of program-specific functions. The *built-in functions* are the following:

<i>unit</i>	: snapshot
<i>fromRef</i>	: $\text{ref} \rightarrow \text{snapshot}$
<i>combine</i>	: $\text{snapshot} \times \text{snapshot} \rightarrow \text{snapshot}$
<i>first</i>	: $\text{snapshot} \rightarrow \text{snapshot}$
<i>second</i>	: $\text{snapshot} \rightarrow \text{snapshot}$
<i>toRef</i>	: $\text{snapshot} \rightarrow \text{ref}$
<i>null</i>	: ref

A heap snapshot represents the values in a subset of the heap. A snapshot is either *unit*, *fromRef*(v) or *combine*(l, r). *unit* is the value of an empty heap. *fromRef*(v) is the value of a single location that holds v . *combine*(l, r) is the union of two heap snapshots. *toRef* deconstructs a *fromRef*(v) snapshot to v . Similarly, *first* and *second* deconstruct *combine*(l, r) to l , respectively r . The constant *null* is the symbolic representation of the value *null*.

In addition to the built-in functions, the signature contains a number of program-specific functions. More specifically, for each pure method p defined in a class C with parameters $C_1 \ x_1, \dots, C_n \ x_n$, the logic includes a function symbol $C.p$ with sort $\text{snapshot} \times \text{ref} \times \text{ref}_1 \times \dots \times \text{ref}_n \rightarrow \text{ref}$. We encode an invocation of a pure method as an application of the corresponding function. The first argument of the function encodes the values in the heap covered by p 's precondition and is used for framing the return value of the pure method. More specifically, if the snapshot of the heap covered by the precondition is the same in two symbolic states, then it follows that the function has the same value in those states. For example, $\text{this}.getX()$ is encoded as $\text{Cell}.getX(s, t_{\text{this}})$, where s is the current snapshot of getX 's precondition and t_{this} is the value of **this** in the symbolic store (see Section 3.3).

3.2.2 Theory

We define the theory Σ_{prelude} as follows:

$(\forall l, r \bullet \text{first}(\text{combine}(l, r)) = l) \wedge$
$(\forall l, r \bullet \text{second}(\text{combine}(l, r)) = r) \wedge$
$(\forall v \bullet \text{toRef}(\text{fromRef}(v)) = v) \wedge$
$(\forall v \bullet \text{fromRef}(\text{toRef}(v)) = v)$

This theory encodes the property that *combine* is injective and that *toRef* and *fromRef* are inverses. In the remainder of this paper, we omit $\Sigma_{prelude}$ for brevity. For example, we write $\Sigma \vdash \psi$ to denote ψ is provable from the theory Σ instead of $\Sigma_{prelude} \cup \Sigma \vdash \psi$. Note that our implementation relies on the Z3 theorem prover [9] to check whether $\Sigma \vdash \psi$. To make this explicit, we write $\Sigma \vdash_{Z3} \psi$ instead.

3.3 Symbolic State

A symbolic state has 4 components: a symbolic heap H , a symbolic old heap G , a symbolic store Γ and a path condition Σ (Definition 2). A symbolic heap is a multiset of heap chunks. A heap chunk is either a field chunk or a predicate chunk. If the symbolic heap contains a field chunk $t_1.f \mapsto t_2$, then (1) $t_1.f$ may currently be accessed and (2) $t_1.f$ holds the value t_2 . In other words, a field chunk represents a permission for accessing a single location and additionally determines the value of that location. A predicate chunk $t_0.q[t_s](t_1, \dots, t_n)$ is “shorthand” for a number of field chunks and assumptions. The term t_s is the snapshot of the predicate. This snapshot determines the value of the field chunks hidden by the predicate chunk. For example, consider the predicate *Cell.valid* from Figure 3(a). The predicate chunk $o.valid[v]()$ is a shorthand for $o.x \mapsto toRef(v)$.

DEFINITION 1. A symbolic heap H is a multiset of heap chunks. A heap chunk is one of the following:

- a field chunk, $t_1.f \mapsto t_2$.
- a predicate chunk $t_0.q[t_s](t_1, \dots, t_n)$.

DEFINITION 2. A symbolic state σ is a quadruple consisting of:

- a symbolic heap, H .
- a symbolic heap, G . G represents the value of the heap on entry to the method under verification.
- a symbolic store, Γ . Γ is a partial function from variables names to terms.
- a path condition, Σ . Σ is a set of formulas.

The ghost statement **close** replaces a number of field chunks by a predicate chunk in the symbolic heap. The body of the predicate determines what field chunks are replaced and which assumptions must hold for the close operation to succeed. Vice versa, **open** removes a predicate chunk from the heap and replaces it with the corresponding field chunks. In addition, **open** adds the assumptions made by the predicate’s body to the path condition. The ghost statement **use** $e_0.p(e_1, \dots, e_n)$; adds the assumption that evaluation of $e_0.p(e_1, \dots, e_n)$ equals evaluation of p ’s body to the path condition, provided to the precondition of the call holds. The formal definitions of open, close and use statements are given in Section 3.4.

As an example, consider the following symbolic state σ :

$$(\{t_1.x \mapsto t_3, t_2.x \mapsto t_3\}, \emptyset, [a \mapsto t_1, b \mapsto t_2], \{t_1 \neq null, t_2 \neq null, t_1 \neq t_2\})$$

The symbolic heap contains two field chunks, the old heap is empty, the store maps the program variables a and b to t_1 , respectively t_2 , and the path condition contains three assumptions: t_1 is not *null*, t_2 is not *null* and t_1 and t_2 are different. Execution of the statement **close** $a.valid()$; changes the symbolic state above into the symbolic state σ' :

$$(\{t_1.valid[fromRef(t_3)](), t_2.x \mapsto t_3\}, \emptyset, [a \mapsto t_1, b \mapsto t_2], \{t_1 \neq null, t_2 \neq null, t_1 \neq t_2\})$$

If we open $a.valid()$; again, we get back the original symbolic state σ . In σ' , we can perform **use** $a.getX()$, since $a.getX()$ ’s precondition holds. This use statement adds the following extra assumption to the path condition:

$$Cell.getX(fromRef(t_3), t_1) = toRef(fromRef(t_3))$$

In other words, we can assume that the symbolic value of $a.getX()$ equals t_3 .

3.4 Execution

In this subsection, we define symbolic execution for expressions (3.4.1), assertions (3.4.2) and statements (3.4.3). The function *eval* is used for evaluating expressions. In several places in the symbolic execution, we need to assume or check that an assertion holds and at the same time add or remove permissions from the heap. For example, when verifying a mutator invocation $e_0.m(e_1, \dots, e_n)$; we must (1) check that m ’s precondition holds and (2) remove the permissions demanded by that precondition from the heap. In addition, we may (1) assume that m ’s postcondition holds after the call and (2) may add the chunks returned by the postcondition to the symbolic heap. We model the transfer of permissions and assuming/checking of assertions by two operations: produce and consume. Checking the precondition is modeled by consume and assuming the postcondition by produce. The function *exec* symbolically executes a statement. The latter functions (*eval*, *consume*, *produce* and *exec*) are all written in continuation passing-style: each function takes a continuation Q that describes the work that remains to be done in the verification of a particular path. Moreover, all four functions return a boolean. If symbolic execution was successful, the result is true; otherwise, the result is false.

We start by giving and describing the signatures of the four functions described above. Their bodies are defined in Sections 3.4.1, 3.4.2 and 3.4.3. The signature of *eval* is the following:

$$eval : SymbolicStates \times Expressions \times (Terms \rightarrow \mathbb{B}) \rightarrow \mathbb{B}$$

That is, *eval* is a function that takes a symbolic state σ , an expression e and a function Q from terms to booleans and returns a boolean itself. $eval(\sigma, e, Q)$ evaluates e in σ and applies Q to the resulting term. If the expression e is not well-defined, *eval* immediately returns false instead of

applying Q to the resulting term. The signature of `produce` is the following:

$$\text{produce} : \text{SymbolicStates} \times \text{Terms} \times \text{Assertions} \\ \times (\text{SymbolicStates} \rightarrow \mathbb{B}) \rightarrow \mathbb{B}$$

`produce` is a function that takes a symbolic state σ , a term t_s , an assertion ϕ and a function Q from symbolic states to booleans. `produce`(σ, t_s, ϕ, Q) adds the permissions (i.e. heap chunks) asserted by ϕ to σ 's symbolic store, adds a formula for each assumption asserted by ϕ to the path condition and passes the resulting state to Q . The values of field chunks and the snapshots of predicate chunks (which are added to the symbolic heap) are determined by the term t_s . If the assertion ϕ is not well-defined, `consume` does not apply the continuation, but instead immediately returns false. The opposite of `produce` is `consume`. The function's signature is the following:

$$\text{consume} : \text{SymbolicStates} \times \text{Assertions} \times \\ (\text{SymbolicStates} \times \text{Terms} \rightarrow \mathbb{B}) \rightarrow \mathbb{B}$$

`consume` takes a symbolic state σ , an assertion ϕ and a function Q from symbolic states and terms to booleans. `consume` checks that ϕ holds, removes the permission asserted by ϕ from the symbolic heap and passes the resulting state, together with the snapshot of heap chunks removed by `consume` to Q . Finally, the signature of `exec` is the following:

$$\text{exec} : \text{SymbolicStates} \times \text{Statements} \times \\ (\text{SymbolicStates} \rightarrow \mathbb{B}) \rightarrow \mathbb{B}$$

`exec` takes a symbolic state σ , a statement s and a function Q from symbolic states to booleans. `exec`(σ, s, Q) executes the statement s in σ and passes the resulting state to Q .

We use the following notation. $\lambda x_1, \dots, x_n \bullet b$ denotes a function with arguments x_1, \dots, x_n and body b . The symbolic heap corresponding to a state σ is denoted H_σ . Similar rules are used for denoting σ 's other components. $\text{old}(\sigma)$ is a shorthand for $(G_\sigma, G_\sigma, \Gamma_\sigma, \Sigma_\sigma)$. $\text{fields}(C)$ is the set of fields defined by the class C . $\text{mpre}(C.m)$, $\text{mbody}(C.m)$, $\text{mpost}(C.m)$ and $\text{mparams}(C.m)$ respectively denote m 's precondition, body, postcondition and parameters. fresh denotes a fresh variable. Note that each occurrence of fresh returns a different variable.

3.4.1 Expressions

The definition of `eval` for each type of expression is shown in Figure 5. The expression `null` evaluates to the constant `null`. Evaluation of a variable x corresponds to looking up the value for x in the symbolic store. To evaluate a field read $e.f$, we first evaluate the target e . Suppose e evaluates to t . We then look up all field chunks in the heap that match $t.f$. That is, matches is a subset of the symbolic heap that contains all field chunks with field f for which we can prove that the target of the field chunk equals t . If the set matches is empty (i.e. no such field chunk exists), then we have no

permission to read $e.f$ and `eval` immediately returns `false`. If matches is not empty, then we pick a field chunk $o.f \mapsto v$ from matches and pass v to the continuation. To evaluate an invocation of a pure method $e_0.p(e_1, \dots, e_n)$, we evaluate all arguments, we check that it follows from the path condition that the target is non-null, and finally we check that p 's precondition holds by consuming it. The function `consume` returns a new state σ' and the snapshot of the consumed part of the heap. An invocation of a pure method is encoded as an application of the corresponding function: $e_0.p(e_1, \dots, e_n)$ evaluates to $C.p(\text{snapshot}, t_0, \dots, t_n)$. A conditional expression $e_1 = e_2 ? e_3 : e_4$ evaluates to an if-then-else term, if we cannot determine from the path condition whether e_1 is equal to e_2 or e_1 is different from e_2 . However, if it is possible to deduce from the path condition which branch is taken, we only evaluate the corresponding expression. An old expression `old`(e) evaluates e under the old heap G_σ . An opening expression `opening` $e_0.q(e_1, \dots, e_n)$ `in` e evaluates e in symbolic state, where the predicate chunk corresponding to $e_0.q(e_1, \dots, e_n)$ is replaced by q 's body. A using expression `using` $e_0.p(e_1, \dots, e_n)$ `in` e evaluates e in a symbolic state with one extra assumption: the evaluation of $e_0.p(e_1, \dots, e_n)$ equals evaluation of p 's method body.

3.4.2 Assertions: produce and consume

The function `produce` is defined in terms of a helper function `produce'`. More specifically,

$$\text{produce}((H, G, \Gamma, \Sigma), \text{snapshot}, \phi, Q) \equiv \\ \text{produce}'(G, \Gamma, \Sigma, \emptyset, \text{snapshot}, \phi, (\lambda H', \Sigma' \bullet \\ Q((H + H', G, \Gamma, \Sigma'))))$$

The signature of `produce'` is

$$\text{produce}' : \text{SymbolicHeaps} \times \text{SymbolicStores} \times \\ \text{PathConditions} \times \text{SymbolicHeaps} \times \text{Terms} \times \text{Assertions} \times \\ (\text{SymbolicHeaps} \times \text{PathConditions} \rightarrow \mathbb{B}) \rightarrow \mathbb{B}$$

The definition of `produce'` for each type of assertion is shown in Figure 6. Producing `true` does not change the heap nor the path condition.

Producing `false` means that the path we are currently investigating is not reachable. Therefore, we can simply disregard the continuation (i.e. the work to be done on the current path) and return `true`. To produce an access assertion `acc`($e.f$), we first evaluate the target e to t . If it does not follow from the path condition that t is non-null, we stop and return `false`. Otherwise, we extend the heap with a field chunk $t.f \mapsto \text{toRef}(\text{snapshot})$. Moreover, we may assume that other field chunks with the same field have different targets. However, if it follows from the path condition that the target one of those other field chunks equals t , then the current path is not reachable and we stop. To produce an equality $e_1 = e_2$, we evaluate e_1 and e_2 to t_1 , respectively t_2 and extend the path condition with $t_1 = t_2$. However, if

$$\begin{aligned}
\text{eval}(\sigma, \text{null}, Q) &\equiv Q(\text{null}) \\
\text{eval}(\sigma, x, Q) &\equiv Q(s_\sigma(x)) \\
\text{eval}(\sigma, e.f, Q) &\equiv \text{eval}(\sigma, e, (\lambda t \bullet \\
&\quad \mathbf{let} \text{ matches} = \{o.f \mapsto v \in H_\sigma \mid \Sigma_\sigma \vdash_{Z3} t = o\} \mathbf{in} \\
&\quad \exists o.f \mapsto v \in \text{matches} \bullet Q(v))) \\
\text{eval}(\sigma, e_0.p(e_1, \dots, e_n), Q) &\equiv \\
&\quad \text{eval}(\sigma, e_0, (\lambda t_0 \bullet \dots \text{eval}(\sigma, e_n, (\lambda t_n \bullet \\
&\quad (\Sigma_\sigma \vdash_{Z3} t_0 \neq \text{null}) \wedge \text{consume}((H_\sigma, G_\sigma, [\mathbf{this} \mapsto t_0, \dots, x_n \mapsto t_n], \Sigma_\sigma), \\
&\quad \text{mpre}(C.p), (\lambda \sigma', \text{snapshot} \bullet \\
&\quad Q(C.p(\text{snapshot}, t_0, \dots, t_n))))))))) \\
\text{eval}(\sigma, e_1 = e_2 ? e_3 : e_4, Q) &\equiv \\
&\quad \text{eval}(\sigma, e_1, (\lambda t_1 \bullet \text{eval}(\sigma, e_2, (\lambda t_2 \bullet \\
&\quad (\Sigma_\sigma \not\vdash_{Z3} t_1 \neq t_2 \wedge \Sigma_\sigma \not\vdash_{Z3} t_1 = t_2 \Rightarrow \\
&\quad \text{eval}(\sigma \cup \{t_1 = t_2\}, e_3, (\lambda t_3 \bullet \\
&\quad \text{eval}(\sigma \cup \{t_1 \neq t_2\}, e_4, (\lambda t_4 \bullet Q(\text{ite}(t_1 = t_2, t_3, t_4)))))) \wedge \\
&\quad (\Sigma_\sigma \vdash_{Z3} t_1 = t_2 \Rightarrow \\
&\quad \text{eval}(\sigma, e_3, (\lambda t_3 \bullet Q(t_3)))) \wedge \\
&\quad (\Sigma_\sigma \vdash_{Z3} t_1 \neq t_2 \Rightarrow \\
&\quad \text{eval}(\sigma, e_4, (\lambda t_4 \bullet Q(t_4)))) \\
&\quad)))) \\
\text{eval}(\sigma, \mathbf{old}(e), Q) &\equiv \\
&\quad \text{eval}(\mathbf{old}(\sigma), e, Q) \\
\text{eval}(\sigma, \mathbf{opening} e_0.q(e_1, \dots, e_n) \mathbf{in} e, Q) &\equiv \\
&\quad \text{eval}(\sigma, e_0, (\lambda t_0 \bullet \dots \text{eval}(\sigma, e_n, (\lambda t_n \bullet \\
&\quad \text{consume}(\sigma, e_0.q(e_1, \dots, e_n), (\lambda(H', G, \Gamma, \Sigma'), \text{snapshot} \bullet \\
&\quad \text{produce}((H', G, [\mathbf{this} \mapsto t_0, \dots, x_n \mapsto t_n], \Sigma'), \text{snapshot}, \text{mbody}(C.q), (\lambda \sigma'' \bullet \\
&\quad \text{eval}((H_{\sigma''}, G, \Gamma, \Sigma_{\sigma''}), e, Q))))))))) \\
\text{eval}(\sigma, \mathbf{using} e_0.p(e_1, \dots, e_n) \mathbf{in} e, Q) &\equiv \\
&\quad \text{eval}(\sigma, e_0.p(e_1, \dots, e_n), (\lambda t_{\text{call}} \bullet \text{eval}(\sigma, \text{body}, (\lambda t_{\text{body}} \bullet \\
&\quad \Sigma \not\vdash_{Z3} t_{\text{call}} \neq t_{\text{body}} \Rightarrow \text{eval}(\sigma \cup \{t_{\text{call}} = t_{\text{body}}\}, e, Q)))) \\
&\quad \text{where } \text{body} \text{ is } \text{mbody}(C.p)[e_0/\mathbf{this}, e_1/x_1, \dots, e_n/x_n]
\end{aligned}$$

Figure 5. Evaluation of expressions.

adding this assumption makes the path condition inconsistent, then the current program point is not reachable and we instead stop investigating this path. To produce a separate conjunction $\phi_1 * \phi_2$, we first produce ϕ_1 and afterwards produce ϕ_2 in the resulting state. Note that *snapshot* is broken down using *first* and *second* and is distributed over ϕ_1 and ϕ_2 . Producing a predicate $e_0.q(e_1, \dots, e_n)$ adds a predicate chunk to the symbolic heap. The snapshot of the predicate equals the snapshot given to *produce'*. Production of a conditional assertion splits symbolic execution into two branches. In one branch, the left-hand side is produced under the assumption the condition holds. In the other branch, the right-hand side is produced under the assumption the condition does not hold. Finally, production of $\mathbf{untouched}(\phi)$ simply adds the assumption that consumption of ϕ in the old and current state yields the same snapshot.

The function *consume* is defined in terms of a helper function *consume'*. More specifically,

$$\begin{aligned}
\text{consume}((H, G, \Gamma, \Sigma), \phi, Q) &\equiv \\
&\quad \text{consume}'((H, G, \Gamma, \Sigma), H, \phi, (\lambda H', \Sigma', \text{snapshot} \bullet \\
&\quad Q((H', G, \Gamma, \Sigma'), \text{snapshot})))
\end{aligned}$$

The signature of *consume'* is

$$\text{consume}' : \text{SymbolicStates} \times \text{Heaps} \times \text{Assertions} \times (\text{SymbolicHeaps} \times \text{PathConditions} \times \text{Terms} \rightarrow \mathbb{B}) \rightarrow \mathbb{B}$$

The definition of *consume'* for each type of assertion is shown in Figure 7. In this definition, H_2 represents the remainder of the original heap which is not consumed yet by the assertion. **true** always holds. On the other hand, **false** never holds, and hence *consume'* stops and returns *false*. Consumption of an access assertion $\mathbf{acc}(e.f)$ checks whether H_2 contains a field chunk that matches $e.f$.

$$\begin{aligned}
& \text{produce}'(G, \Gamma, \Sigma, H, \text{snapshot}, \mathbf{true}, Q) \equiv \\
& \quad Q(H, \Sigma) \\
& \text{produce}'(G, \Gamma, \Sigma, H, \text{snapshot}, \mathbf{false}, Q) \equiv \\
& \quad \mathbf{true} \\
& \text{produce}'(G, \Gamma, \Sigma, H, \text{snapshot}, \mathbf{acc}(e.f), Q) \equiv \\
& \quad \text{eval}((H, G, \Gamma, \Sigma), e, (\lambda t \bullet \\
& \quad \quad \mathbf{let} \{o_1.f_1 \mapsto v_1, \dots, o_n.f_n \mapsto v_n\} = \{o.g \mapsto v \in H \mid f = g\} \mathbf{in} \\
& \quad \quad (\Sigma \vdash_{Z3} t \neq \mathbf{null}) \wedge \\
& \quad \quad ((\Sigma \not\vdash_{Z3} o_1 = t \vee \dots \vee o_n = t) \Rightarrow \\
& \quad \quad \quad Q(H + \{t.f \mapsto \text{toRef}(\text{snapshot})\}, \Sigma \cup \{o_1 \neq t \wedge \dots \wedge o_n \neq t\}))) \\
& \text{produce}'(G, \Gamma, \Sigma, H, \text{snapshot}, e_1 = e_2, Q) \equiv \\
& \quad \text{eval}((H, G, \Gamma, \Sigma), e_1, (\lambda t_1 \bullet \text{eval}((H, G, \Gamma, \Sigma), e_2, (\lambda t_2 \bullet \\
& \quad \quad \Sigma \not\vdash t_1 \neq t_2 \Rightarrow Q(H, \Sigma \cup \{t_1 = t_2\})))))) \\
& \text{produce}'(G, \Gamma, \Sigma, H, \text{snapshot}, \phi_1 * \phi_2, Q) \equiv \\
& \quad \text{produce}'(G, \Gamma, \Sigma, H, \text{first}(\text{snapshot}), \phi_1, (\lambda H' \bullet \Sigma' \bullet \\
& \quad \quad \text{produce}'(G, \Gamma, \Sigma', H', \text{second}(\text{snapshot}), \phi_2, Q))) \\
& \text{produce}'(G, \Gamma, \Sigma, H, \text{snapshot}, e_0.q(e_1, \dots, e_n), Q) \equiv \\
& \quad \text{eval}((H, G, \Gamma, \Sigma), e_0, (\lambda t_0 \bullet \dots \text{eval}((H, G, \Gamma, \Sigma), e_n, (\lambda t_n \bullet \\
& \quad \quad (\Sigma \vdash_{Z3} t_0 \neq \mathbf{null}) \wedge Q(H + \{t_0.q[\text{snapshot}](t_1, \dots, t_n)\}, \Sigma)))))) \\
& \text{produce}'(G, \Gamma, \Sigma, H, \text{snapshot}, e_1 = e_2 ? \phi_1 : \phi_2, Q) \equiv \\
& \quad \text{eval}((H, G, \Gamma, \Sigma), e_1, (\lambda t_1 \bullet \text{eval}((H, G, \Gamma, \Sigma), e_2, (\lambda t_2 \bullet \\
& \quad \quad \Sigma \not\vdash_{Z3} t_1 \neq t_2 \Rightarrow \text{produce}'(G, \Gamma, \Sigma \cup \{t_1 = t_2\}, H, \text{snapshot}, \phi_1, Q) \wedge \\
& \quad \quad \Sigma \not\vdash_{Z3} t_1 = t_2 \Rightarrow \text{produce}'(G, \Gamma, \Sigma \cup \{t_1 \neq t_2\}, H, \text{snapshot}, \phi_2, Q)))))) \\
& \text{produce}'((H, G, \Gamma, \Sigma), \text{snapshot}, \text{untouched}(\phi), Q) \equiv \\
& \quad \text{consume}((H, G, \Gamma, \Sigma), \phi, (\lambda(-, -, -), \Sigma'), \text{snapshot}' \bullet \\
& \quad \quad \text{consume}((G, G, \Gamma, \Sigma'), \phi, (\lambda(-, -, -), \Sigma''), \text{snapshot}'' \bullet \\
& \quad \quad \Sigma'' \vdash_{Z3} \text{snapshot}' = \text{snapshot}'' \wedge Q((H, G, \Gamma, \Sigma''))))
\end{aligned}$$

Figure 6. Production of assertions.

If this is the case, then that chunk is removed from H_2 and its value is returned as the snapshot of the consumed heap. Consumption of $e_1 = e_2$ corresponds to checking whether e_1 evaluates to the same term as e_2 . However, no heap is consumed by $e_1 = e_2$ and therefore, the returned snapshot is *unit*. To consume a separating conjunction $\phi_1 * \phi_2$, we first consume ϕ_1 and then consume ϕ_2 in the resulting heap. The snapshot of $\phi_1 * \phi_2$ is the combination of the snapshots of ϕ_1 and ϕ_2 . A predicate is consumed by removing the corresponding chunk from the symbolic heap. The snapshot of the predicate is passed to Q . Consumption of a conditional assertion is similar to production of such an assertion. Finally, $\text{untouched}(\phi)$ holds only if the snapshot of ϕ is the same in the current and the old state.

3.4.3 Statements

The definition of `exec` for each statement type is shown in Figure 8. A variable declaration $C \ x;$ adds an entry for x to the symbolic store. A variable update $x := e$ first evaluates e . Suppose e evaluates to t , then the entry for x in the symbolic store is updated to t . A field update $e_1.f := e_2$; starts by evaluating e_1 to t_1 and e_2 to t_2 . Then, we look up all field chunks in the heap that match $t_1.f$. That is, the set *matches* is the subset of H that contains all field

chunks $o.f \mapsto v$ such that it follows from the path condition that $o = t_1$. If *matches* is empty, then we do not have permission to write to $e_1.f$, and hence we immediately return false. Otherwise, we change the value of the field chunk for $t_1.f$ in the heap to v . Execution of a mutator invocation $e_0.m(e_1, \dots, e_n)$; proceeds in 4 steps: (1) we evaluate the arguments, (2) we check that it follows from the path condition that the target is non-null, (3) we consume the precondition and (4) produce the postcondition. An object creation $x := \mathbf{new} \ C;$ introduces a fresh variable o , that represents the identity of the new object. Moreover, the heap is extended with a field chunk for each of o 's fields, the store is updated at x with o , and the assumption that o is not *null* is added to the path condition. To execute an if-then-else statement $\mathbf{if}(e_1 = e_2) \{ \bar{s}_1 \} \mathbf{else} \{ \bar{s}_2 \}$, we split the symbolic execution into two branches. More specifically, we execute the then-branch under the assumption $e_1 = e_2$ and the else-branch under the assumption that $e_1 \neq e_2$. However, we only consider a branch if we cannot prove it is unreachable. For example, the then-branch is only considered if it does not follow from the path condition that e_1 and e_2 are different.

Execution of an assert statement $\mathbf{assert} \ e_1 = e_2;$ is equivalent to skip, if it follows from the path condition that e_1 equals e_2 . Otherwise, `exec` immediately returns *false*.

$$\begin{aligned}
& \text{consume}'((H, G, \Gamma, \Sigma), H_2, \text{true}, Q) \equiv \\
& \quad Q(H_2, \Sigma, \text{unit}) \\
& \text{consume}'((H, G, \Gamma, \Sigma), H_2, \text{false}, Q) \equiv \\
& \quad \text{false} \\
& \text{consume}'((H, G, \Gamma, \Sigma), H_2, \text{acc}(e.f), Q) \equiv \\
& \quad \text{eval}((H, G, \Gamma, \Sigma), e, (\lambda t \bullet \\
& \quad \quad \text{let matches} = \{o.g \mapsto v \in H_2 \mid f = g \wedge \Sigma \vdash t = o\} \text{ in} \\
& \quad \quad \exists o.g \mapsto v \in \text{matches} \bullet Q(H_2 - \{o.g \mapsto v\}, \Sigma, \text{fromRef}(v)))) \\
& \text{consume}'((H, G, \Gamma, \Sigma), H_2, e_1 = e_2, Q) \equiv \\
& \quad \text{eval}((H, G, \Gamma, \Sigma), e_1, (\lambda t_1 \bullet \text{eval}((H, G, \Gamma, \Sigma), e_2, (\lambda t_2 \bullet \\
& \quad \quad \Sigma \vdash_{Z3} t_1 = t_2 \wedge Q(H_2, \Sigma, \text{unit})))))) \\
& \text{consume}'((H, G, \Gamma, \Sigma), H_2, \phi_1 * \phi_2, Q) \equiv \\
& \quad \text{consume}'((H, G, \Gamma, \Sigma), H_2, \phi_1, (\lambda H'_2, \Sigma', \text{lhs_snapshot} \bullet \\
& \quad \quad \text{consume}'((H, G, \Gamma, \Sigma'), H'_2, \phi_2, (\lambda H''_2, \Sigma'', \text{rhs_snapshot} \bullet \\
& \quad \quad \quad Q(H''_2, \Sigma'', \text{combine}(\text{lhs_snapshot}, \text{rhs_snapshot})))))) \\
& \text{consume}'((H, G, \Gamma, \Sigma), H_2, e_0.q(e_1, \dots, e_n), Q) \equiv \\
& \quad \text{eval}((H, G, \Gamma, \Sigma), e_0, (\lambda t_0 \bullet \dots \text{eval}((H, G, \Gamma, \Sigma), e_n, (\lambda t_n \bullet \\
& \quad \quad \text{let matches} = \{t'_0.q[t'_s](t'_1, \dots, t'_n) \in H_2 \mid \Sigma \vdash_{Z3} t_0 = t'_0 \wedge \dots \wedge t_n = t'_n\} \text{ in} \\
& \quad \quad \exists t'_0.q[t'_s](t'_1, \dots, t'_n) \in \text{matches} \bullet Q(H_2 - \{t'_0.q[t'_s](t'_1, \dots, t'_n)\}, \Sigma, t'_s)))))) \\
& \text{consume}'((H, G, \Gamma, \Sigma), H_2, e_1 = e_2 ? \phi_1 : \phi_2, Q) \equiv \\
& \quad \text{eval}((H, G, \Gamma, \Sigma), e_1, (\lambda t_1 \bullet \text{eval}((H, G, \Gamma, \Sigma), e_2, (\lambda t_2 \bullet \\
& \quad \quad (\Sigma_\sigma \not\vdash_{Z3} t_1 \neq t_2 \Rightarrow \text{consume}((H, G, \Gamma, \Sigma \cup \{t_1 = t_2\}), H_2, \phi_1, Q)) \wedge \\
& \quad \quad (\Sigma_\sigma \not\vdash_{Z3} t_1 = t_2 \Rightarrow \text{consume}((H, G, \Gamma, \Sigma \cup \{t_1 \neq t_2\}), H_2, \phi_2, Q)))))) \\
& \text{consume}'((H, G, \Gamma, \Sigma), H_2, \text{untouched}(\phi), Q) \equiv \\
& \quad \text{consume}'((H, G, \Gamma, \Sigma), H, \phi, (\lambda -, -, \text{snapshot}' \bullet \\
& \quad \quad \text{consume}'((G, G, \Gamma, \Sigma), G, \phi, (\lambda -, -, \text{snapshot}'' \bullet \Sigma \vdash_{Z3} \text{snapshot}' = \text{snapshot}''))) \wedge \\
& \quad Q(H_2, \Sigma, \text{unit})
\end{aligned}$$

Figure 7. Consumption of assertions.

An open statement **open** $e_0.q(e_1, \dots, e_n)$; replaces a predicate chunk in the symbolic heap by the body of the predicate. More specifically, the predicate itself is consumed and the predicate's body is produced. Note that the snapshot of the predicate chunk (which is returned by **consume**) is used for producing the body. **close** is the opposite of **open**: it replaces the body of a predicate in the symbolic heap by a predicate chunk. More specifically, the body is consumed and the predicate itself is added to the symbolic heap. Note that the version of the predicate chunk equals the snapshot returned by consumption of the body. Finally, a **use** $e_0.p(e_1, \dots, e_n)$; statement adds an assumption to the path condition: the function application $C.p(\text{snapshot}, t_0, t_1, \dots, t_n)$ is equal to evaluation of p 's body.

3.5 Validity

A program is valid if the symbolic execution of each method succeeds. More specifically, a *valid program* only contains valid methods and has a valid main routine. A mutator is valid if (1) its method contract is well-defined and (2) the method body satisfies the method contract. Both (1) and (2) are checked in a symbolic store that contains arbitrary (i.e. fresh) variables. Proof obligation (2) is checked by produc-

ing the precondition, by executing the method body in the resulting state and by finally consuming the postcondition. Since the snapshot used for producing the precondition is a fresh variable, the body is verified for arbitrary values of the heap that satisfy the precondition. Definedness of the contract is checked by producing both the pre- and postcondition. The main routine is valid if it satisfies the contract **requires true**; **ensures true**; Predicates and pure methods are valid if their preconditions and bodies are well-defined.

DEFINITION 3. *A mutator*

$$\begin{aligned}
& \text{void } m(C_1 t_1, \dots, C_n t_n) \\
& \quad \text{requires } \phi_{pre}; \text{ ensures } \phi_{post}; \\
& \quad \{ \bar{s} \}
\end{aligned}$$

is valid if all of the following hold:

- The precondition is well-defined and the postcondition is well-defined, provided the precondition held in the method pre-state.

$$\begin{aligned}
& \text{let } \Gamma = [\text{this} \mapsto \text{fresh}, x_1 \mapsto \text{fresh}, \dots, x_n \mapsto \text{fresh}] \text{ in} \\
& \text{produce}((\emptyset, \emptyset, \Gamma, \{\Gamma(t_0) \neq \text{null}\}), \text{fresh}, \phi_{pre}, \\
& \quad (\lambda(H', -, \Gamma', \Sigma') \bullet \\
& \quad \quad \text{produce}((\emptyset, H', \Gamma', \Sigma'), \text{fresh}, \phi_{post}, (\lambda -, - \bullet \text{true}))))
\end{aligned}$$

$$\begin{aligned}
& \text{exec}((H, G, \Gamma, \Sigma), C \ x; , Q) \equiv \\
& \quad Q((H, G, \Gamma[x \mapsto \text{null}], \Sigma)) \\
& \text{exec}((H, G, \Gamma, \Sigma), x := e; , Q) \equiv \\
& \quad \text{eval}((H, G, \Gamma, \Sigma), e, (\lambda t \bullet Q((H, G, \Gamma[x \mapsto t], \Sigma)))) \\
& \text{exec}((H, G, \Gamma, \Sigma), e_1.f := e_2; , Q) \equiv \\
& \quad \text{eval}((H, G, \Gamma, \Sigma), e_1, (\lambda t_1 \bullet \text{eval}((H, G, \Gamma, \Sigma), e_2, (\lambda t_2 \bullet \\
& \quad \quad \text{let matches} = \{o.f \mapsto v \mid \Sigma \vdash_{Z3} o = t_1\} \text{ in} \\
& \quad \quad \exists o.f \mapsto v \in \text{matches} \bullet Q((H - \{o.f \mapsto v\} + \{o.f \mapsto t_2\}, G, \Gamma, \Sigma)))))) \\
& \text{exec}((H, G, \Gamma, \Sigma), e_0.m(e_1, \dots, e_n); , Q) \equiv \\
& \quad \text{eval}((H, G, \Gamma, \Sigma), e_0, (\lambda t_1 \bullet \dots \text{eval}((H, G, \Gamma, \Sigma), e_n, (\lambda t_n \bullet \\
& \quad \quad \Sigma \vdash_{Z3} t_0 \neq \text{null} \wedge \text{consume}((H, G, \Gamma, \Sigma), \text{pre}, (\lambda(H', -, -, \Sigma'), \bullet \\
& \quad \quad \text{produce}((H', H, [\text{this} \mapsto t_0, \dots, x_n \mapsto t_n], \Sigma'), \text{fresh}, \text{mpost}(C.m), (\lambda(H'', -, -, \Sigma'')) \bullet \\
& \quad \quad \quad Q((H'', G, \Gamma, \Sigma'')))))))) \\
& \quad \text{where pre is mpre}(C.m)[e_0/\text{this}, e_1/x_1, \dots, e_n/x_n] \\
& \text{exec}((H, G, \Gamma, \Sigma), x := \text{new } C; , Q) \equiv \\
& \quad \text{let } o = \text{freshvar in} \\
& \quad \quad Q((H + \{o.f_1 \mapsto \text{null}, \dots, o.f_n \mapsto \text{null}\}, G, \Gamma[x \mapsto o], \Sigma \cup \{o \neq \text{null}\})) \\
& \quad \quad \text{where fields}(C) = C_1 f_1 \dots C_n f_n \\
& \text{exec}((H, G, \Gamma, \Sigma), \text{if}(e_1 = e_2) \{ \bar{s}_1 \} \text{ else } \bar{s}_2; , Q) \equiv \\
& \quad \text{eval}((H, G, \Gamma, \Sigma), e_1, (\lambda t_1 \bullet \text{eval}((H, G, \Gamma, \Sigma), e_2, (\lambda t_2 \bullet \\
& \quad \quad \Sigma \not\vdash_{Z3} t_1 \neq t_2 \Rightarrow \text{exec}((H, G, \Gamma, \Sigma \cup \{t_1 = t_2\}), \bar{s}_1, Q) \wedge \\
& \quad \quad \Sigma \not\vdash_{Z3} t_1 = t_2 \Rightarrow \text{exec}((H, G, \Gamma, \Sigma \cup \{t_1 \neq t_2\}), \bar{s}_2, Q)))) \\
& \text{exec}((H, G, \Gamma, \Sigma), \text{assert } e_1 = e_2; , Q) \equiv \\
& \quad \text{eval}((H, G, \Gamma, \Sigma), e_1, (\lambda t_1 \bullet \text{eval}((H, G, \Gamma, \Sigma), e_2, (\lambda t_2 \bullet \\
& \quad \quad \Sigma \vdash_{Z3} t_1 = t_2 \wedge Q((H, G, \Gamma, \Sigma)))))) \\
& \text{exec}((H, G, \Gamma, \Sigma), \text{open } e_0.q(e_1, \dots, e_n); , Q) \equiv \\
& \quad \text{eval}((H, G, \Gamma, \Sigma), e_0, (\lambda t_0 \bullet \dots \text{eval}((H, G, \Gamma, \Sigma), e_n, (\lambda t_n \bullet \\
& \quad \quad \text{consume}((H, G, \Gamma, \Sigma), e_0.q(e_1, \dots, e_n), (\lambda(H', -, -, \Sigma'), \text{snapshot} \bullet \\
& \quad \quad \text{produce}((H, G, [\text{this} \mapsto t_0, \dots, x_n \mapsto t_n], \Sigma), \text{snapshot}, \text{mbody}(C.q), (\lambda(H'', -, -, \Sigma'')) \bullet \\
& \quad \quad \quad Q((H'', G, \Gamma, \Sigma'')))))))) \\
& \text{exec}((H, G, \Gamma, \Sigma), \text{close } e_0.q(e_1, \dots, e_n); , Q) \equiv \\
& \quad \text{eval}((H, G, \Gamma, \Sigma), e_0, (\lambda t_0 \bullet \dots \text{eval}((H, G, \Gamma, \Sigma), e_n, (\lambda t_n \bullet \\
& \quad \quad \text{consume}((H, G, [\text{this} \mapsto t_0, \dots, x_n \mapsto t_n], \Sigma), \text{mbody}(C.m), (\lambda(H', -, -, \Sigma'), \text{snapshot} \bullet \\
& \quad \quad \quad Q((H' + \{t_0.q[\text{snapshot}](t_1, \dots, t_n)\}, G, \Gamma, \Sigma')))))))) \\
& \text{exec}((H, G, \Gamma, \Sigma), \text{use } e_0.p(e_1, \dots, e_n); , Q) \equiv \\
& \quad \text{eval}((H, G, \Gamma, \Sigma), e_0.p(e_1, \dots, e_n), (\lambda t_{\text{call}} \bullet \text{eval}((H, G, \Gamma, \Sigma), \text{body}, (\lambda t_{\text{body}} \bullet \\
& \quad \quad Q((H, G, \Gamma, \Sigma \cup \{t_{\text{call}} = t_{\text{body}}\})))))) \\
& \quad \text{where body is mbody}(C.p)[e_0/\text{this}, e_1/x_1, \dots, e_n/x_n]
\end{aligned}$$

Figure 8. Executions of statements.

- The method body satisfies the method contract.

$$\begin{aligned}
& \text{let } \Gamma = [\text{this} \mapsto \text{fresh}, x_1 \mapsto \text{fresh}, \dots, x_n \mapsto \text{fresh}] \text{ in} \\
& \text{produce}((\emptyset, \emptyset, \Gamma, \{\Gamma(t_0) \neq \text{null}\}), \text{fresh}, \phi_{\text{pre}}, \\
& \quad (\lambda(H', -, \Gamma', \Sigma') \bullet \\
& \quad \quad \text{exec}((H', H', \Gamma', \Sigma'), \bar{s}, (\lambda(H'', G'', \Gamma'', \Sigma'') \bullet \\
& \quad \quad \quad \text{consume}((H'', G'', \Gamma'', \Sigma''), \phi_{\text{post}}, (\lambda_{-} \bullet \text{true}))))))
\end{aligned}$$

DEFINITION 4. A predicate

$$\text{predicate } q(C_1 t_1, \dots, C_n t_n) \{ \text{return } \phi_{\text{body}}; \}$$

is valid if the body is a well-defined assertion:

$$\begin{aligned}
& \text{let } \Gamma = [\text{this} \mapsto \text{fresh}, x_1 \mapsto \text{fresh}, \dots, x_n \mapsto \text{fresh}] \text{ in} \\
& \text{produce}((\emptyset, \emptyset, \Gamma, \{\Gamma(t_0) \neq \text{null}\}), \text{fresh}, \phi_{\text{body}}, (\lambda_{-} \bullet \text{true}))
\end{aligned}$$

DEFINITION 5. A pure method

$$\begin{aligned}
& \text{pure } C \ p(C_1 t_1, \dots, C_n t_n) \\
& \quad \text{requires } \phi_{\text{pre}}; \\
& \quad \{ \text{return } e_{\text{body}}; \}
\end{aligned}$$

is valid if the precondition is a well-defined assertion and the body is well-defined, provided the precondition holds:

$$\begin{aligned}
& \text{let } \Gamma = [\text{this} \mapsto \text{fresh}, x_1 \mapsto \text{fresh}, \dots, x_n \mapsto \text{fresh}] \text{ in} \\
& \text{produce}((\emptyset, \emptyset, \Gamma, \{t_0 \neq \text{null}\}), \text{fresh}, \phi_{\text{pre}}, (\lambda(H, -, \Gamma, \Sigma) \bullet \\
& \quad \text{eval}((H', \emptyset, \Gamma', \Sigma'), e_{\text{body}}, (\lambda_{-} \bullet \text{true}))))
\end{aligned}$$

DEFINITION 6. The main routine \bar{s} is valid if the following holds

$$\text{exec}((\emptyset, \emptyset, \emptyset, \emptyset), \bar{s}, \lambda_- \bullet \text{true})$$

DEFINITION 7. A program is valid if all methods and the main routine are valid.

3.6 Pure method termination

It is essential for the soundness of our approach that pure methods terminate. Verification therefore includes a phase that checks sufficient conditions for pure method termination. Specifically, it is checked for each pure method call in a pure method body that either (1) the callee is defined earlier in the program text, or (2) the call is in the body of an **opening** expression, or (3) there is some symbolic heap chunk that is not consumed by the precondition of the call. This ensures that at each call, either the size of the symbolic heap decreases, or the *derivation depth* decreases (i.e. the number of close operations required to construct the heap from one that contains only field chunks), or the position in the program text decreases. Since the size and the derivation depth are always finite and a pure method cannot increase the size or the derivation depth of the symbolic heap, this ensures termination.

4. Examples

This section contains a number of programs written in the language of Figure 4.

We assume the usual syntactic sugar. That is, the receiver **this** can be omitted in field accesses, access assertions and method invocations. A contract can contain multiple preconditions **requires** $\phi_1; \dots$ **requires** ϕ_n ; which is a shorthand for **requires** $\phi_1 * \dots * \phi_n$; . The same rule goes for postconditions. Omitted pre- and postconditions default to **true**. A constructor

$C(C_1 x_1, \dots, C_k x_k)$ **requires** ϕ_1 ; **ensures** ϕ_2 ; $\{\bar{s}\}$

is a shorthand for the mutator method

```
void initC(C1 x1, ..., Ck xk)
  requires acc(f1) * ... * acc(fn) *  $\phi_1$ ; ensures  $\phi_2$ ;
   $\{\bar{s}\}$ 
```

where f_1, \dots, f_n are the fields of C . Accordingly, a constructor invocation $x := \text{new } C(e_1, \dots, e_k)$; abbreviates $x := \text{new } C$; $x.\text{init}_C(e_1, \dots, e_k)$;

4.1 Cell

The first program we consider is the class *Cell* shown in Figure 9. The intermediate symbolic states encountered during symbolic execution of *setX* are highlighted in blue. The difference between this version of *Cell* and the one shown in Figure 3(a) are the ghost operations. Omitting one of these operations causes symbolic execution to fail. For example, consider the body of *setX*. If one omits the first open statement, then the assignment $x := v$; fails, since **this.x** is

not accessible. Omitting the close statement causes the consumption of the postcondition to fail, as the heap contains no predicate chunk for **this.valid**() . Finally, if one removes the using statement, then the second conjunct of the postcondition is not be provable from the path condition.

Note that the main routine of Figure 3(b) still verifies. In particular, no additional ghost statements need to added to ensure symbolic execution succeeds.

```
class Cell {
  int x;

  Cell(int v)
    ensures valid() * getX() = v;
    { x := v; close valid(); use getX(); }

  void setX(int v)
    requires valid();
    ensures valid() * getX() = v;
    {
      ({t.valid[ts](), {t.valid[ts}](),
      [this  $\mapsto$  t, v  $\mapsto$  tv], {this  $\neq$  null})
      open valid();
      ({t.x  $\mapsto$  toRef(ts), {t.valid[ts}](),
      [this  $\mapsto$  t, v  $\mapsto$  tv], {this  $\neq$  null})
      x := v;
      ({t.x  $\mapsto$  tv, {t.valid[ts}](),
      [this  $\mapsto$  t, v  $\mapsto$  tv], {this  $\neq$  null})
      close valid();
      (t.valid[fromRef(tv)](), {t.valid[ts}](),
      [this  $\mapsto$  t, v  $\mapsto$  tv], {this  $\neq$  null})
      use getX();
      (t.valid[fromRef(tv)](), {t.valid[ts}](),
      [this  $\mapsto$  t, v  $\mapsto$  tv],
      {this  $\neq$  null, Cell.getX(fromRef(tv), t) = tv})
    }

  predicate valid()
    { return acc(this.x); }

  pure int getX()
    requires valid();
    { return opening valid() in x; }
}
```

Figure 9. The class *Cell*.

Eliminating Ghost Operations

The ghost operations give the programmer strong control over the actions performed during symbolic execution. However, writing ghost statements is cumbersome. In this section, we describe two techniques that help eliminate the need

for ghost operations: unfolding contracts and inference of open, close and use statements.

When verifying the body of a method m in a class C , one can automatically open predicates and pure methods used in m 's method contract, provided their definition is visible to C . For example, instead of using the contract of Figure 9, we verify $setX$'s body with respect to the contract

```
requires acc(this.x);
ensures acc(this.x) * this.x = v;
```

The unfolding described above suffices to remove all ghost operations from Figure 9.

Another technique to reduce the annotation overhead is inference of open, close and use statements. That is, one can modify the rules of Section 3.4 as follows:

- Instead of returning an error when consumption of a predicate $e_0.q(e_1, \dots, e_n)$ fails, one can attempt to close the predicate first and retry consumption if closing succeeds. The number of nested close attempts must be bounded to ensure termination in the presence of recursive predicates.
- If evaluation of an invocation of a pure method succeeds, then the path condition is updated as though there was a use operation. This rule requires modifying the definition of eval such that it additionally passes a (potentially updated) path condition to its continuation.
- Suppose one imposes the restriction that a predicate always includes permission to access the fields of the receiver. If access to a field $o.f$ is required, but no corresponding field chunk is found, then one can instead look for a predicate with receiver o and open that predicate. This rule was first proposed in [19].

4.2 Iterator

Figure 10 shows an implementation of the iterator pattern (we use the rules described above to avoid having to write ghost operations). Reasoning about this pattern is challenging because of *sharing*: when multiple iterators share the same list, the invariants of those iterators all access the list's internal state. Therefore, an iterator's invariant must demand access to both the locations of the iterator itself and the locations of the corresponding list. How can we ensure that the invariant $i_1.valid()$ is preserved when calling $i_2.next()$?

The key to solving the problem is the fact that the iterators share only locations of their list and that those locations are never modified by *next*. The contract of *next* specifies that the method only reads locations covered by $getList().valid()$ via its last postcondition. Informally, this postcondition guarantees that the locations to which $getList().valid()$ demands access are not modified by *next*. During symbolic execution of a call to *next*, this postcondition allows the verifier to deduce that $getList().valid()$'s snapshot is the same before and after the call.

Note that the conjunct $acc(items.elems)$ in *ArrayList*'s invariant is a special access assertion that gives permission to access the elements of the array. Also, it is ok for the invariant to read $items.length$ without demanding access since $length$ is immutable.

Modifying a list while iterators are iterating over it can give rise to unexpected exceptions. For example, removing an element can give rise to an *ArrayOutOfBoundsException* when calling the iterator's *next* method. Java's implementation of *Iterator* guards against concurrent modifications by tracking and checking a version field.

Our implementation does not need to track such a field to stay safe. Instead, when an *ArrayList* object is modified, all its iterators are automatically invalidated. Indeed, the precondition of the method *Iterator.valid* includes the invariant of the list. Hence, any modification to the list causes all information about the invariant to be lost.

5. Soundness

We highlight the main ideas underlying the soundness of our approach. A detailed soundness proof is available separately [20].

Our approach for framing pure method calls depends on the property that the value of a pure method call depends only on the argument values and the *snapshot* of the part of the heap described by the method's precondition. This is the case since the snapshot includes the value of each field read directly or indirectly by the pure method's body.

Soundness can be proved by a traditional preservation and progress approach, based on a notion of a *valid configuration*, defined in terms of our exec function: a configuration is valid if exec of the statements that remain to be executed is satisfied by some symbolic state that represents the current execution state, with a postcondition of **true**. A symbolic state represents a given concrete state if there is an interpretation of the logical symbols that satisfies the path condition and that maps the symbolic store onto the concrete store, and the symbolic heap onto a heap that can be obtained by performing a finite number of **close** operations on the concrete heap. Validity of the initial configuration follows directly from validity of the main routine. Preservation of validity follows directly from the definition of exec for most execution steps.

For mutator calls, preservation of validity follows from (1) validity of the callee with respect to its contract, (2) the fact that produce and consume are each other's inverse, and (3) a separation-logic-style frame rule that can be proved for our exec function. The frame rule states that if a statement is valid in a given heap, then it is valid in any extension thereof, and the statement leaves the added chunks unchanged.

6. Experience

We implemented the approach described in Section 3.4 in a verifier prototype. The prototype was used to verify several

```

class ArrayList {
  Object[] items; int size;

  ArrayList()
    ensures valid();
    ensures getSize() = 0;
  { items := new Object[10]; }

  void add(Object o)
    requires valid();
    ensures valid();
    ensures getSize() = old(getSize()) + 1;
  { ... }

  predicate valid()
  {
    return acc(items) * acc(size) *
      items ≠ null * acc(items.elems) *
      0 ≤ size ≤ items.length;
  }

  pure int getSize()
    requires valid();
  { return size; }
}

```

(a)

```

class Iterator {
  ArrayList list; int index;

  Iterator(ArrayList l)
    requires l.valid();
    ensures fp() * l.valid() * valid();
    ensures getList() = l;
    ensures untouched(l.valid());
  { list := l; }

  Object next()
    requires fp() * getList().valid() * valid();
    requires hasNext();
    ensures fp() * getList() = old(getList());
    ensures getList().valid() * valid();
    ensures untouched(getList().valid());
  { return list.items[index++]; }

  predicate fp()
  { return acc(list) * acc(index) * list ≠ null; }

  pure ArrayList getList()
    requires fp();
  { return list; }

  pure bool valid()
    requires fp() * getList().valid();
  { return 0 ≤ index ≤ list.size(); }

  pure bool hasNext()
    requires fp() * getList().valid();
  { return index < list.size(); }
}

```

(b)

Figure 10. The *Iterator* pattern.

programs, including the ones described in this paper. *Cell* and *Iterator* both verify in under a second on a machine with a Pentium Core Duo 2.66GHz processor and 4 GB of memory running Windows Vista. We use the Z3 theorem prover [9] to check whether a formula follows from the path condition. For these examples, the performance is similar to the implementation based on verification condition generation (VCG) [7]. However, verification of the visitor pattern with VCG took 127 seconds. The implementation based on symbolic execution on the other hand needs only a second! In general, we noticed that for simple, small methods the performances of VCG and symbolic execution are comparable. However, for complex, larger methods symbolic execution beats VCG hands down. For example, we verified the

following code snippet for increasing sizes of n :

```

Cell c1 := new Cell();
c1.setX(1);
...
Cell cn := new Cell();
cn.setX(n);
assert c1.getX() = 1;

```

The time taken using both verification condition generation and symbolic execution is shown in Figure 11. We stopped at n equal to 14, because the VCG-based implementation was not able to prove the assertion for n larger than 14.

The verifier prototype can be downloaded from the author's homepage: <http://www.cs.kuleuven.be/~jans/vericool3>.

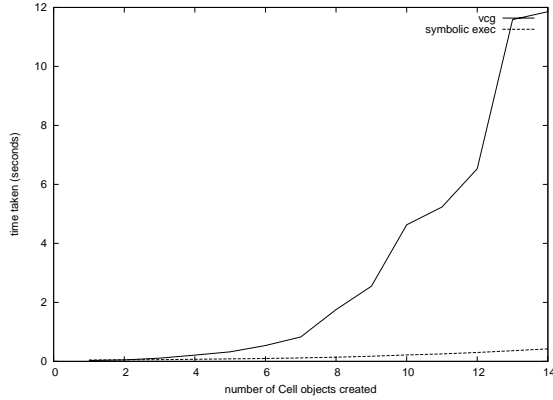


Figure 11. Verification times for a simple code snippet.

6.1 Join

The symbolic execution branches at each if-then-else statement: one branch executes the then-path under the assumption the condition holds, while the other branch checks the else-path under the assumption the condition is false. Note that both the then-path and the else-path include the statements following the if-then-else statement. In principle, this may lead to an exponential explosion in the number of paths that need to be considered. For example, consider the method m shown below.

```

void m(int x1, . . . , int xn) {
  if(x1 = 0) {} else {}
  . . .
  if(xn = 0) {} else {}
}

```

The number of paths through m is 2^n . That is, the number of paths grows exponentially with the number of if-statements. In practice, this exponential behavior is rarely problematic for several reasons. First of all, it occurs only if the conditions of the if-statements are completely independent. If those conditions are not independent, then some paths are infeasible. Before starting the verification of a branch, our verifier checks that the branch is reachable (by checking that the path condition remains consistent). Secondly, our tool verifies the method m in less than a second for n up to 10. Methods with more than 10 independent if-statements rarely occur in practice. Finally, to speed up verification for programs that do contain many independent if-statements, one can use joins. A join statement consists of the keyword **join** followed by an assertion ϕ . Such a statement effectively splits the verification of a method into two parts. One part verifies that all paths occurring before the join end up in a state satisfying ϕ . The second part verifies that a symbolic state that satisfies ϕ starting from the join ends up in a state satisfying the postcondition. A single join statement reduces the number of paths from 2^n to $2 \times 2^{n/2}$.

6.2 Symbolic Debugging

If a verification condition generation-based verifier detects an error, it typically reports which (part of the) proof obligation it was unable to prove. For example, a typical error message is “Precondition at (line:column) might not hold”. However, one of the reasons why verification is considered to be difficult, is that is hard to understand why the theorem prover fails to prove a particular proof obligation. Symbolic execution can help in debugging error messages in two ways. First of all, it can exactly pinpoint the part of the precondition that failed. For example, it can show that a particular $\text{acc}(e.f)$ conjunct of the precondition was not satisfied, because the symbolic heap did not contain the corresponding field chunk. Secondly, when verification fails, our tool can show the symbolic states on the path leading to the failure. The programmer can then for instance observe that the missing field chunk was consumed, but not returned by an earlier call.

7. Related Work

Our approach was heavily inspired by separation logic [21, 22, 11]. In particular, the access assertion $\text{acc}(e.f)$ is similar to separation logic’s points-to predicate $e.f \mapsto _$ and Parkinson and Bierman’s abstract predicates inspired our predicate methods. A difference between separation logic and our approach is that we allow using heap-dependent expressions, in particular field reads and pure method invocations, and **old** expressions inside assertions. Distefano and Parkinson [12] implemented a verifier for Java based on separation logic, called jStar. Our symbolic states are very similar to theirs. One notable difference is that our predicate chunks carry around a snapshot, which determines the values of the field chunks hidden by the predicate. Another difference is that we use an SMT solver, namely Z3, to check whether certain properties follow from the path condition.

The classical dynamic frames approach [23, 13, 24, 25, 26, 27] solves the frame problem by explicitly annotating methods with effect annotations. More specifically, the contract of a mutator consists of a modifies clause and a “swinging pivot postcondition”, while a pure method’s contract includes a reads clause. The expressiveness of the dynamic frames approach stems from the fact that these effect annotations can mention arbitrary sets of memory locations. To support data abstraction, these location sets may be specified in terms of dynamic frames, i.e. pure methods or ghost fields that denote sets of locations. Implicit dynamic frames is a variant of the classical dynamic frames approach. Instead of relying on explicit modifies and reads annotations, frame information is implicitly inferred from method preconditions. This typically leads to more concise contracts and faster verification.

In [28], the authors propose using data groups to specify side-effects. To ensure soundness, their approach imposes two methodological restrictions: the pivot uniqueness and

owner exclusion restriction. Our approach imposes no such restrictions, and as a consequence it can handle programs that [28] cannot. For example, the former restriction rules out sharing of representation objects, as is the case in the iterator pattern.

In the universe type system [29] and the Boogie methodology [30], abstractions (pure methods, invariants or model fields) can depend on the fields of owned objects and the fields of peers (i.e. objects with the same owner as the receiver), provided the abstraction is visible to the peer. For example, the method *hasNext* of an iterator would have to be visible to the list class. Our approach has no such restriction. Our **open** and **close** statements are similar to **pack** and **unpack** from the Boogie methodology.

The use of pure methods in specifications has been discussed extensively in the literature [31, 32, 33, 34, 35, 36]. In particular, encoding pure methods as functions in the logic is a standard technique in verification. Other authors have also proposed using a form of heap snapshots to frame the return values of pure methods. In our approach, snapshots are not deduced from ownership relations, but from the precondition (an assertion) of a pure method. To the best of our knowledge, this is the first approach that uses pure methods in the context of symbolic execution. Some authors propose broadening the range of admissible pure methods by allowing certain side-effects. We believe our approach can be extended to support such weakly pure methods.

Implicit dynamic frames and a corresponding verification technique based on verification condition generation was proposed by Smans *et al.* [37, 7]. In this paper, we propose an alternative way of verifying whether a program satisfies its implicit dynamic frames annotations via symbolic execution. The main advantages of this new approach are that (1) verification is faster (compared to the verification condition generation-based technique), (2) verification times are more predictable and (3) that the programmer can determine why verification failed by inspecting the symbolic states on the path to the failure.

Leino *et al.* extend the implicit dynamic frames approach with fractional permissions and concurrency. Their Exhale and Inhale functions are similar to our consume and produce, as they are also used for checking/assuming assertions and transferring permissions. However, they do not show how their encoding can handle data abstraction.

8. Conclusion

In this paper, we propose an approach for automatically verifying whether a Java program satisfies its specification based on symbolic execution. Compared to approaches based on a weakest precondition calculus and verification condition generation, our new approach is faster, more predictable and can give better feedback on why a verification fails. These advantages are a consequence of the fact that the symbolic execution algorithm handles many heap-related opera-

tions directly instead of encoding the state of the heap in quantifier-rich, first-order logical formulae and leaving it up to the theorem prover to reason about this encoded heap.

References

- [1] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. *CASSIS*, 3362, 2004.
- [2] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI*, 2002.
- [3] Karen Zee, Viktor Kuncak, and Martin Rinard. Full functional verification of linked data structures. In *PLDI*, 2008.
- [4] Ernie Cohen, Michal Moskał, Wolfram Schulte, and Stephan Tobies. A practical verification methodology for concurrent programs. Technical Report MSR-TR-2009-15, Microsoft Research, 2009.
- [5] Patrice Chalin, Perry R. James, and George Karabotsos. JML4: Towards an industrial grade IVE for java and next generation research platform for JML. In *VSTTE*, 2008.
- [6] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The Krakatoa tool for certification of Java/JavaCard programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2), 2004.
- [7] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. In *ECOOP*, 2009.
- [8] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
- [9] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.
- [10] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, 2003.
- [11] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.
- [12] Dino Distefano and Matthew Parkinson. jstar: Towards practical verification for java. In *OOPSLA*, 2008.
- [13] Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *FM*, 2006.
- [14] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.
- [15] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. *Behavioral Specifications for Businesses and Systems*, 1999.
- [16] Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3), 2005.
- [17] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice

- Chalin, and Daniel M. Zimmerman. JML reference manual. 2008.
- [18] Matthew Parkinson. *Local Reasoning for Java*. PhD thesis, University of Cambridge, 2005.
- [19] Huu Hai Nguyen, Cristina David, Shengchao Qin, and Wei-Ngan Chin. Automated verification of shape and size properties via separation logic. In *VMCAI*, 2007.
- [20] Jan Smans, Bart Jacobs, and Frank Piessens. Symbolic execution for implicit dynamic frames: Soundness proof. URL: <http://www.cs.kuleuven.be/~jans/vericool3>, 2009.
- [21] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *POPL*, 2005.
- [22] Matthew Parkinson and Gavin Bierman. Separation logic, abstraction and inheritance. In *POPL*, 2008.
- [23] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Regional logic for local reasoning about global invariants. In *ECOOP*, 2008.
- [24] Anindya Banerjee, Mike Barnett, and David A. Naumann. Boogie meets regions: a verification experience report. In *VSTTE*, 2008.
- [25] K. Rustan M. Leino. Specification and verification of object-oriented software. In *Marktoberdorf International Summer School*, 2008.
- [26] Bernd Schoeller. *Making Classes Provable through Contracts, Models and Frames*. PhD thesis, Departement Informatik ETH Zurich, 2007.
- [27] Jan Smans, Bart Jacobs, Frank Piessens, and Wolfram Schulte. An automatic verifier for Java-like programs based on dynamic frames. In *FASE*, 2008.
- [28] K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. In *PLDI*, 2002.
- [29] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, FernUniversität Hagen, 2001.
- [30] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6), 2003.
- [31] Arsenii Rudich, Ádám Darvas, and Peter Müller. Checking well-formedness of pure-method specifications. In *FM*, 2008.
- [32] Ádám Darvas and Peter Müller. Reasoning about method calls in interface specifications. *Journal of Object Technology*, 5(5), 2006.
- [33] Bart Jacobs and Frank Piessens. Inspector methods for state abstraction. *Journal of Object Technology*, 6(5), 2007.
- [34] Mike Barnett, David A. Naumann, Wolfram Schulte, and Qi Sun. 99.44% pure: Useful abstractions in specifications. In *FTFJP*, 2004.
- [35] Ádám Darvas and K. Rustan M. Leino. Practical reasoning about invocations and implementations of pure methods. In *FASE*, 2007.
- [36] K. Rustan M. Leino and Ronald Middelkoop. Proving consistency of pure methods and model fields. In *FASE*, 2009.
- [37] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. In *FTFJP*, 2008.