VeriFast for Java: A Tutorial

Jan Smans, Bart Jacobs, and Frank Piessens

iMinds-DistriNet, KU Leuven, Belgium

Abstract. VeriFast is a separation logic-based program verifier for Java. This tutorial introduces the verifier's features step by step.

1 Introduction

When writing programs, developers make design decisions that allow them to argue — at least informally — why their program does what it is supposed to do. For example, a developer may design a certain sequence of statements to compute the average of an integer array and may implicitly decide that this sequence should never be applied to `null` or to an empty array. A violation of a design decision then corresponds to a bug in the program. However, keeping track of all such implicit decisions and ensuring that the code satisfies those decisions is hard. This is particularly true for concurrent programs where the effect of concurrently executing threads must be taken into account to avoid data races.

To help developers manage their design decisions, certain programming languages provide a type system where developers can express decisions regarding the kind of data memory locations can hold. For example, consider the signature of the method `Arrays.copyOf` in the standard Java library:

```
public static int[] copyOf(int[] original, int newLength)
```

The types in the method signature make explicit the decision that `newLength` is an integer and that both `original` and the return value are integer arrays. The Java compiler statically checks that the code and the decisions described by the types are consistent. For example, the statement

```
int[] copy = Arrays.copyOf(true, 10);
```

is rejected as ill-typed by the compiler, as `true` is not an integer array. By checking consistency of the code and the types at compile-time, the compiler rules out certain run-time errors. In particular, if a Java program typechecks, then that program does not contain field- and method-not-found errors.

The expressive power of traditional type systems is limited. For example, while the signature of `Arrays.copyOf` expresses that the list of actual parameters must consist of an array and an integer, the requirements that the array must be non-null and that the integer must be non-negative lies beyond the expressive power of the Java type system. Instead, these requirements are included only in the informal documentation and an unchecked exception is thrown when they are violated. Similarly, the implicit guarantee that the return value is non-null and that it is a proper copy of `original` is documented informally, but it is not expressed in `copyOf`'s return type. However, contrary to the requirement described above violations of the guarantee do not give rise to an exception, allowing erroneous results caused by a bug in the implementation to spread to other components potentially causing these components to malfunction.

Formal verification is a program analysis technique where developers can insert assertions to formally specify detailed design decisions that lie beyond the expressive power of traditional type systems and where consistency of these decisions and the code can be checked at compile-time by a program verifier. In general, if a program verifier deems a program to be correct, then that program's executions do not perform assertion violations.

In this tutorial, we describe a particular separation-logic based[1] program verifier for Java named VeriFast[2]. VeriFast takes a number of Java source files annotated with preconditions, postconditions and other specifications describing assumptions made by the developer as input, and checks whether the assumptions hold in each execution of the program for arbitrary input. If VeriFast deems a Java program to be correct, then that program does not contain assertion violations, data races, divisions by zero, null dereferences, array indexing errors and the program makes correct use of the Java API. This tutorial is targeted at users of VeriFast. We refer the reader to a technical report [4] for a formal description of the inner workings of the tool for a small imperative language. We compare VeriFast with related tools and approaches in Section 8.

We proceed by introducing VeriFast's features step by step. To try the examples and exercises in the paper yourself, download the VeriFast distribution from the following website:

http://distrinet.cs.kuleuven.be/software/VeriFast

The distribution includes a command-line tool (`verifast`) and a graphical user interface (`vfide`). The Java programs mentioned in this paper can be downloaded from the website as well.

2 Verification

2.1 Assert Statements

A Java assert statement [5, section 10.14] consists of the keyword `assert` followed by a boolean expression. By inserting an assert statement in the code, a developer indicates that he or she expects the corresponding boolean expression to evaluate to `true` whenever the statement is reached during the program's execution. If the expression evaluates to `false`, an `AssertionError` is thrown (provided assertion checking is enabled). As an example, consider the method `max` shown below:

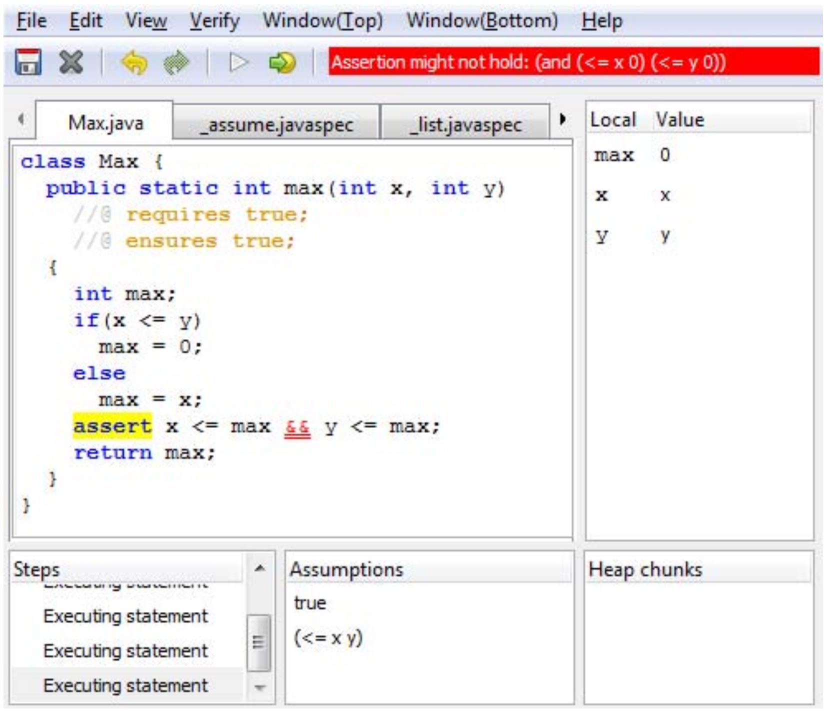
[1] Separation logic is an extension of Hoare logic oriented to reasoning about imperative programs with aliasing. The theory behind separation logic for Java is explained by Parkinson and Bierman in a different chapter of this book [1].

[2] In addition to Java, VeriFast supports C [2,3].

```
class Max {
  public static int max(int x, int y)
    //@ requires true;
    //@ ensures true;
  {
    int max;
    if(x <= y)
      max = 0;
    else
      max = x;
    assert x <= max && y <= max;
    return max;
  }
}
```

The goal of this method is to compute the maximum of `x` and `y`. The assert statement expresses the developer's assumption that the variable `max` should be larger than or equal to both `x` and `y`. The program is well-typed, and is therefore accepted by the Java compiler. The assert statement holds when `x` is larger than `y`, but fails for example when `x` equals `5` and `y` equals `10`.

VeriFast is a Java source code analysis tool. Contrary to standard Java compilers, VeriFast detects potential assertion violations (and other problems) at compile-time. To apply the tool to our example program, start the VeriFast integrated development environment (`vfide`), open `Max.java` and press the `Verify` button (▷). You should now see the following window:



Assertion might not hold: (and (<= x 0) (<= y 0))

```
class Max {
  public static int max(int x, int y)
    //@ requires true;
    //@ ensures true;
  {
    int max;
    if(x <= y)
      max = 0;
    else
      max = x;
    assert x <= max && y <= max;
    return max;
  }
}
```

Local	Value
max	0
x	x
y	y

Executing statement
Executing statement
Executing statement

true
(<= x y)

The large text box displays the source code of `Max.java`, the program being analysed. The body of the assert statement is highlighted in red indicating that the tool has located a potential bug in the statement. A description of the error is shown in red at the top of the window: `Assertion might not hold`. This error message indicates that VeriFast was unable to prove that the assert statement holds in each possible execution of the method. In other words, there might exist a context where execution of the assert statement fails. The other parts of the user interface contain information that can be used to diagnose verification errors as explained in Section 2.3.

To fix the bug, modify the then branch of the if statement and assign the correct value to `max`. Reverify the program. VeriFast now displays `0 errors found` in green at the top of the window, meaning that for all possible values for `x` and `y` the assert statement succeeds.

2.2 Method Contracts

The types in a method signature describe the kind of data expected and returned by a method. Each method call is then typechecked with respect to the callee's signature, not with respect to the callee's body. Therefore, typechecking is *modular*.

Performing modular analysis has a number of advantages. First of all, it is possible to analyse a method body with respect to the signature of a callee even if the callee is an interface method, the callee's body is not visible to the caller or the callee has simply not been implemented yet. Secondly, modular analysis scales to larger programs as each method body needs to be analysed only once instead of once per call. Finally, modifying the implementation of a method never breaks the correctness (e.g. type correctness in case of a type system) of its callers.

For the reasons outlined above, VeriFast performs modular verification: each method call is verified with respect to the callee's signature. However, as explained in Section 1, method signatures in traditional type systems can only express simple assumptions. Therefore, VeriFast mandates that each method signature is extended with a precondition and a postcondition. The precondition (keyword `requires`) refines the signature by defining additional constraints on the method parameters, while the postcondition (keyword `ensures`) defines additional constraints on the method's return value.

The pre- and postcondition of a method can be viewed as a contract between developers that call the method and those implementing the method body [6]. Implementers may assume that the precondition holds on entry to the method, and in return they are obliged to provide a method body that establishes the postcondition when the method returns[3]. Callers must ensure the precondition holds when they call the method, and in return they may assume that the postcondition holds when the method returns. VeriFast enforces method contracts.

As method contracts are mandatory in VeriFast, we have already annotated the method `max` in our running example with a default contract. More specifically, `max`'s current precondition is `true`, indicating that callers can pass any

[3] For now, we consider only normal, non-exceptional termination.

two integers for `x` and `y`. Similarly, the method's postcondition is `true`, indicating that the method body is allowed to return any integer. Note that method contracts — and all other VeriFast annotations — are written inside special comments (`/*@ ... @*/`). These comments are ignored by the Java compiler, but recognized by VeriFast.

To see that VeriFast only uses the callee's method contract to reason about a call, extend the class `Max` with a new method `max3` that computes the maximum of three integers by calling `max` twice as shown below.

```
class Max {
  ...
  public static int max3(int x, int y, int z)
  {
    int max;
    max = max(x, y);
    max = max(max, z);
    assert x <= max && y <= max && z <= max;
    return max;
  }
}
```

When verifying this program, VeriFast asks us to provide a method contract for `max3` (`Method must have contract`). To resolve this problem, annotate `max3` with the same contract as `max` and reverify the program. The verifier now reports that the assertion in `max3` might not hold. VeriFast is unable to prove this assertion because `max`'s postcondition provides no information about its return value. According to the contract, `max` could potentially be implemented as `return 0;`, which would clearly violate the assertion if either `x`, `y` or `z` is larger than zero. Resolve the problem by strengthening `max`'s postcondition to `x <= result && y <= result`. The variable `result` in the postcondition denotes `max`'s return value. After updating the postcondition, verification succeeds.

2.3 Symbolic Execution

How does VeriFast check that a program is correct (i.e. does not contain assertion violations, null dereferences, ...)? When the developer presses the `Verify` button, VeriFast checks that each method in the program satisfies its method contract via symbolic execution. A method body *satisfies a method contract* if for each program state s that satisfies the precondition, execution of the method body starting in s does not trigger illegal operations (such as assertion violations and divisions by zero) and the postcondition holds when the method terminates. VeriFast only checks partial correctness so a method is not required to terminate.

The number of program states that satisfy the precondition is typically infinite. For example, consider a method with a parameter of type `String`. The length of the `String` object passed to this method is only bounded by the available memory. Moreover, execution of the method body may not terminate for certain inputs. Therefore, it is generally not feasible to enumerate all initial states satisfying the precondition and to check for each of these states that the body satisfies the contract by simply executing the body.

To verify that a method body satisfies a method contract, VeriFast uses *symbolic* instead of concrete execution. More specifically, the tool constructs a symbolic state that represents an arbitrary concrete pre-state which satisfies the precondition and checks that the body satisfies the contract for this symbolic state[4]. The verifier symbolically executes the body starting in the initial symbolic state. At each statement encountered during symbolic execution, the tool checks that the statement cannot go wrong and it updates the symbolic state to reflect execution of that statement. Finally, when the method returns, VeriFast checks that the postcondition holds for all resulting symbolic states.

A symbolic state in VeriFast is a triple (γ, Σ, h), consisting of a symbolic store γ, a path condition Σ and a symbolic heap h. The symbolic store is a mapping from local variables to symbolic values. Each symbolic value is a first-order term, i.e. a symbol, or a literal number, or an operator ($+$, $-$, $<$, $=$, ...) or a function applied to first-order terms. A single symbolic value can represent a large, potentially infinite number of concrete values. For example, during verification of the method `max` the initial symbolic value of the parameter `x` is the symbol x. This symbol represents all 2^{32} possible values of `x`. The path condition is a set of first-order formulas describing the conditions that hold on the path being verified. For example, when verifying the then branch of an if statement, the path condition contains an assumption expressing that the condition of the if statement was true when entering the branch. Finally, the symbolic heap is a multi-set of heap chunks. The symbolic heap is the key to reasoning about aliasing and preventing data races. Its purpose will be explained in Section 3. A single symbolic state can represent a large, potentially infinite number of concrete states. For example, the symbolic pre-state of the method `max` represents all 2^{64} possible valuations of the parameters `x` and `y`.

When VeriFast reports an error, the symbolic states on the path leading to the error can be examined in the IDE to diagnose the problem. That is, the box in the bottom left of the IDE contains the list of symbolic states on the path to the error. When a particular symbolic state is selected, the corresponding statement or assertion is highlighted in yellow in the program. Moreover, the three components of that symbolic state are displayed: the path condition is shown in the bottom center, the symbolic heap in the bottom right and the symbolic store in the top right box. The symbolic store is a table where the left side contains the names of the local variables in scope, while the right side shows their symbolic values. One can inspect the symbolic states on a path leading to a particular statement by placing the cursor at that statement and by pressing the `Run to cursor` button (➡). Similarly, the symbolic states leading to the end of the method can be examined by placing the cursor on the closing brace of the

[4] This is similar to universal generalization where one proves $\forall x \bullet P(x)$ by using a symbolic value x to represent an arbitrary concrete value and by showing that P holds for x.

method body and by pressing `Run to cursor`. Place your cursor at the closing brace of `max` and examine the intermediate symbolic states.

The body of each pre- and postcondition consists of an *assertion*. For now, an assertion is a side-effect free, heap-independent Java boolean expression, but we will introduce additional types of assertions in the next sections. A key part of VeriFast's symbolic execution algorithm is checking whether a symbolic state satisfies an assertion and recording the assumption that an assertion is true in a symbolic state. We call the former operation *consuming an assertion* and the latter *producing an assertion*. Consuming a Java boolean expression means symbolically evaluating the expression yielding a first-order formula and checking that this formula is derivable from the path condition. VeriFast relies on an SMT solver[7], a kind of automatic theorem prover, to discharge such proof obligations. Producing a Java boolean expression corresponds to evaluating that expression yielding a first-order formula and adding it to the path condition.

Symbolic execution of each method starts by initializing the symbolic store by assigning a fresh first-order symbol to each parameter. VeriFast selects the symbol x as the fresh term representing the symbolic value of a parameter `x`. Initially, the path condition and the heap are both empty. The resulting symbolic state thus represents an arbitrary concrete pre-state. To consider only states that satisfy the precondition, VeriFast first produces the precondition. The verifier then proceeds by symbolically executing the method body. At each return in the method body, VeriFast checks that the symbolic post-state satisfies the post-condition by consuming it. In the remainder of this section, we explain symbolic execution for various statements in more detail.

Assignment. Symbolic execution of an assignment to a local variable x consists of two steps. First, the right hand side is symbolically evaluated yielding a first-order term. Afterwards, the value of x in the symbolic store is changed to this first-order term. As an example, consider the method `bar`[5] shown below:

```
public static int bar(int x)
  //@ requires 0 < x;
  //@ ensures 10 < result;
{ x = x + 10; return x; }
```

The contract of `bar` states that the method's return value is larger than `10`, provided `x` is non-negative. Symbolic execution of `bar` starts by constructing a symbolic pre-state that represents an arbitrary concrete pre-state by assigning a fresh symbol to each method parameter: the initial symbolic value of `x` is a fresh symbol x. To consider only program states that satisfy the precondition, VeriFast produces the precondition: the assumption that the symbolic value of the parameter `x` is non-negative is added to the path condition. Verification proceeds by symbolic execution of the method body. The assignment `x = x + 10;`

[5] To verify `Bar.java`, disable overflow warnings by unchecking `Check arithmetic overflow` in the `Verify` menu. In the remainder of this paper, we assume overflow checking is disabled.

updates the symbolic value of x to $\mathsf{x} + 10$, encoding the fact that x's new value is equal to its original value plus ten. The return statement sets the ghost[6] variable result, which represents bar's return value, to the symbolic value of x and ends execution of the method body. VeriFast finally checks that the postcondition holds in the symbolic post-state by consuming the postcondition. The postcondition holds as the corresponding first-order formula, $10 < \mathsf{x} + 10$, is derivable from the path condition, $0 < \mathsf{x}$ by the SMT solver.

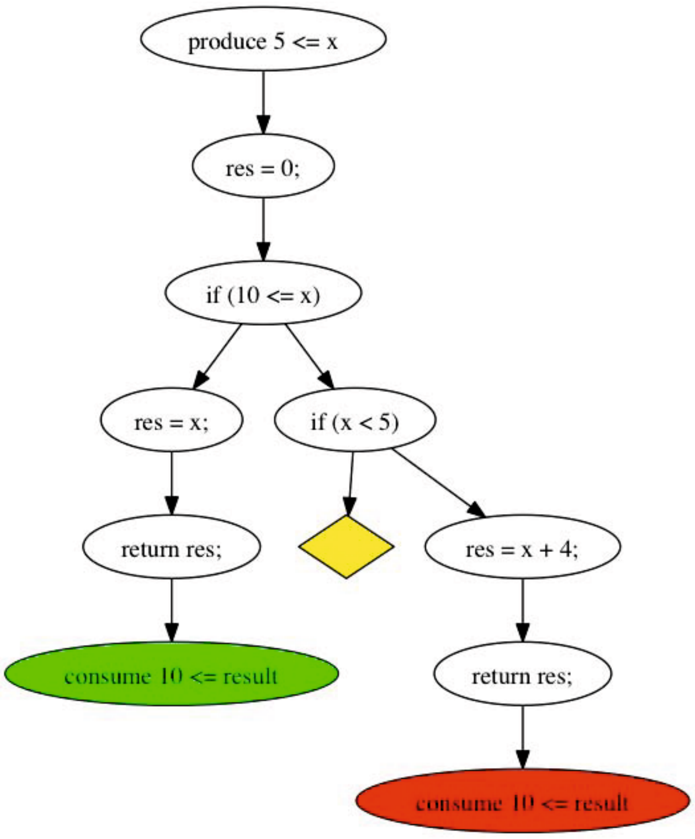
Assert. VeriFast checks that assert statements do not fail by consuming their bodies. That is, VeriFast checks for each such statement that its boolean expression evaluates to true by proving that the corresponding formula follows from the path condition. For example, consider the assert statement in the body of max3. The path condition of the symbolic state right before execution of this statement contains three assumptions: true, $\mathsf{x} \leq \mathsf{max0} \wedge \mathsf{y} \leq \mathsf{max0}$ and $\mathsf{max0} \leq \mathsf{max1} \wedge \mathsf{z} \leq \mathsf{max1}$. The first assumption represents the precondition of max3 itself, while the second and third assumption respectively correspond to the postcondition of the first and second call to max. Here, the symbols max0 and max1 respectively represent the return value of the first and second invocation of max. VeriFast concludes that the assert statement succeeds for all possible values for x, y and z as the SMT solver can prove that the corresponding formula, $\mathsf{x} \leq \mathsf{max1} \wedge \mathsf{y} \leq \mathsf{max1} \wedge \mathsf{z} \leq \mathsf{max1}$, follows from the path condition.

If. In a concrete execution, the condition of an if statement evaluates to either true or false. If the condition evaluates to true, then the then branch is executed; otherwise, the else branch is taken. However, symbolic evaluation of the condition of an if statement results in a first-order formula. Based on this formula, it is generally not possible to decide which branch must be taken. For example, consider the method foo shown below.

```
public static int foo(int x)
  //@ requires 5 <= x;
  //@ ensures 10 <= result;
{
  int res = 0;
  if (10 <= x)
    res = x;
  else if (x < 5)
    assert false;
  else
    res = x + 4;
  return res;
}
```

[6] A ghost variable is a variable introduced only to facilitate verification but which does not exist during concrete execution of the program.

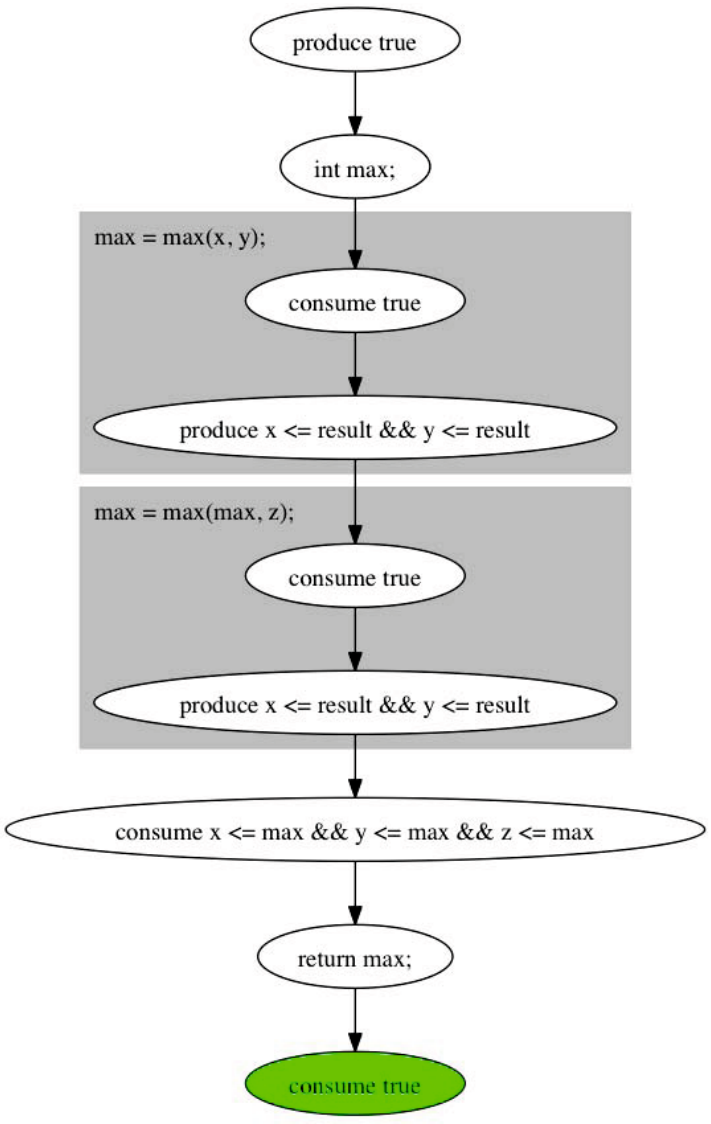
During symbolic execution of the body it is not known whether the condition of the outermost if statement, `10 <= x`, holds. For that reason, VeriFast examines both branches of the if statement: the then branch (and all subsequent statements after the if statement) are verified under the assumption that $10 \leq x$, while the else branch (and again all subsequent statements) are verified assuming the negation of the condition `10 > x`. The same strategy applies to the if statement in the else branch. This tactic for dealing with branches leads to the following symbolic execution tree:



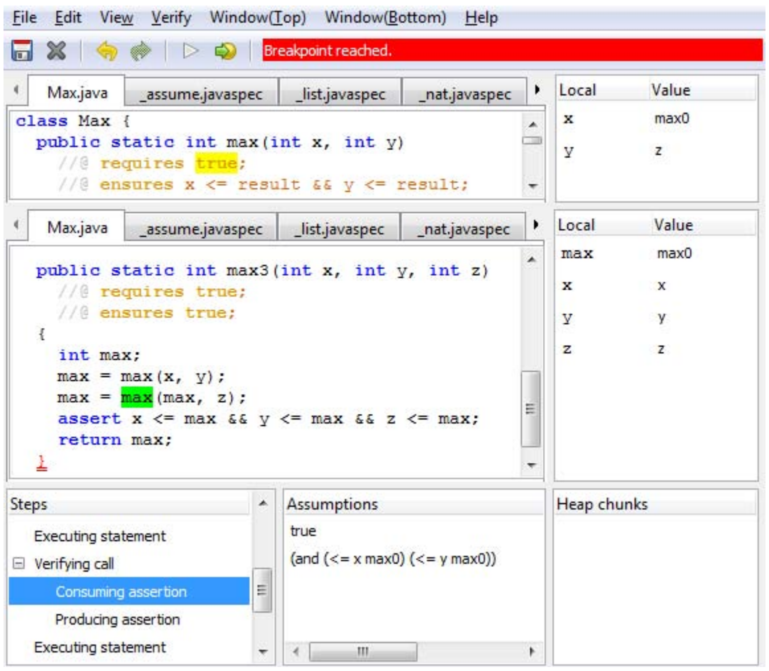
The leftmost branch of the tree corresponds to a path where the method terminates and the resulting symbolic state satisfies the postcondition. More specifically, the formula corresponding to the postcondition, $10 \leq x$, is derivable from the final path condition ($5 \leq x$ and $10 \leq x$). The diamond node represents a symbolic state with an inconsistent path condition. Such states are not reachable during concrete executions of the program. Indeed, the assert statement can never be reached as `x` cannot be both larger than or equal to `5` and at the same time less than `5`. VeriFast does not examine infeasible paths any further. Finally, the rightmost branch of the tree ends in a verification error: a postcondition violation was detected by VeriFast. The formula representing the postcondition, $10 \leq x + 4$, is not derivable from the path condition ($5 \leq x$, $10 > x$ and $x \geq 5$). Indeed, the postcondition does not hold if `x` equals `5`.

VeriFast traverses execution trees in a depth-first manner. Furthermore, the tool does not report all problems on all paths but stops when it finds the first error or when all paths successfully verify. The symbolic states that can be examined in the IDE are those on the path leading to the error. For our example, the IDE only displays the path from the root to the bottom right node.

Call. As explained in Section 2.2, VeriFast uses the callee's method contract to reason about a method call. More specifically, symbolic execution of a call consists of two steps: (1) consumption of the callee's precondition and (2) production of its postcondition. Both steps are executed under the callee's symbolic store. During production of the postcondition, the callee's return value is represented by the ghost variable `result`. This variable is initialized to a fresh symbol that represents an arbitrary concrete return value in the callee's symbolic store just before production of the postcondition. As an example, consider the symbolic execution tree of the method `max3`:


produce true
int max;
max = max(x, y);
consume true
produce x <= result && y <= result
max = max(max, z);
consume true
produce x <= result && y <= result
consume x <= max && y <= max && z <= max
return max;
consume true

Each call to `max` is symbolically executed by consuming `max`'s precondition and afterwards producing its postcondition. To aid developers in understanding verification of method calls, the VeriFast IDE displays the signature of the callee and the callee's symbolic store in addition to the call site itself as shown below. For example, the callee's symbolic store (shown in the top right) for the second call to `max` maps the variable `x` to `max0`, the symbolic value returned by the first call to `max`.



```
class Max {
  public static int max(int x, int y)
    //@ requires true;
    //@ ensures x <= result && y <= result;
```

Local	Value
x	max0
y	z

```
  public static int max3(int x, int y, int z)
    //@ requires true;
    //@ ensures true;
  {
    int max;
    max = max(x, y);
    max = max(max, z);
    assert x <= max && y <= max && z <= max;
    return max;
  }
```

Local	Value
max	max0
x	x
y	y
z	z

Assumptions

```
true
(and (<= x max0) (<= y max0))
```

3 Classes and Objects

In a Java program without objects and without static fields, the only program state visible to a method are the values of its local variables. Local variables are not shared among activation records, meaning that a local variable can only be read and written by the method itself. In particular, it is impossible for a callee to modify the local variables of its caller. As local variables are not shared, reasoning about such variables during verification is easy: one does not need to worry that a method call will unexpectedly modify the caller's local variables or that multiple threads concurrently access the same local variable.

In Java programs with objects, the program state visible to a method consists not only of its local variables but also includes all objects in the heap transitively reachable from those variables. The set of objects reachable from distinct activation records can overlap because of aliasing. Two expressions are *aliases* if they refer to the same object. This means that one activation record can directly access state visible to another activation record. In particular, if a caller and a callee share a reference to the same object, then the callee can modify the state of that object. Moreover, if activation records in different threads concurrently access the same memory location, a data race may arise. Thus, the state visible to a method can change even though the method itself does not explicitly perform an assignment.

Modular verification in the presence of aliasing is challenging. Suppose that at a particular point in the analysis of a method the verifier knows that property P

holds for part of the heap. For example, P might state that calling a particular method of a certain object will return 5 or that some object is in a consistent state. Whenever the method performs a call, P can potentially be invalidated by the callee. Even if the method itself does not perform any calls, P could be invalidated under its feet by a different thread. To perform modular verification, VeriFast applies an ownership regime. That is, the tool tracks during symbolic execution what part of the program state is *owned* by the method, meaning that properties about that part are *stable* under actions performed by callees and by other threads.

Aliasing also complicates reasoning because it can introduce *hidden* dependencies. Suppose that at a particular point in the analysis the verifier knows that the properties P and Q hold. If P and Q involve classes that hide their internal state, the verifier does not know exactly which heap locations P and Q depend on. However, they might depend on the same heap location because of aliasing. As such whenever the method performs an operation that affects P, Q also may no longer hold. To perform modular verification, the verifier must somehow be able to deduce which properties are independent of each other. Note that a property that does not contain any direct or indirect dereferences cannot contain hidden dependencies. The set of memory locations that such a property depends on are the local variables it mentions. This set can be determined syntactically. We will refer to such properties as pure assertions.

To support modular verification in the presence of aliasing, VeriFast explicitly tracks the set of heap locations owned by the method being analysed during symbolic execution. That is, the symbolic heap is a multiset of heap chunks. Each heap chunk represents a memory region that is owned by the method. In addition to indicating ownership, the chunk can contain information on the state of that memory region. For example, the heap chunk $\mathsf{C\_f}(\mathsf{o}, \mathsf{v})$ represents exclusive ownership of the field $\mathsf{C.f}$ of object o. The chunk additionally describes the property that the field's current value is v. Here, both o and v are symbolic values.

All heap chunks in the symbolic heap represent mutually disjoint memory regions. In particular, if the heap contains two field chunks $\mathsf{C\_f}(\mathsf{o}_1, \mathsf{v})$ and $\mathsf{C\_f}(\mathsf{o}_2, \mathsf{w})$, then o_1 and o_2 are distinct. As chunks on the symbolic heap do not share hidden dependencies, the verifier can safely assume that an operation that only affects a particular chunk does not invalidate the information in the remaining chunks.

VeriFast enforces the program-wide invariant that at any time each heap location is exclusively owned by at most one activation record[7]. Moreover, a method is only allowed to access a heap location if it owns that location. By enforcing these rules, the verifier guarantees that information about a memory region cannot be invalidated by callees or other threads, as long as the method retains ownership of that region. For example, as long as a method owns the heap chunk

[7] VeriFast also supports non-exclusive, partial ownership via fractions [8]. Partial ownership allows the owning method to read but not write the memory region. The program-wide invariant then states that for each heap location the total sum of fractions over all activation records is at most one.

`C_f(o, v)`, the property that `o.f` equals `v` cannot be invalidated by other activation records, as these activation records lack the permission to assign to `o.f`. Conversely, if a method does not own a particular memory location `o.f`, then that method has no direct information on the value of that field (as that information is attached to the heap chunk). In addition, the ownership methodology ensures the absence of data races. A data race occurs if two threads concurrently access the same heap location at the same time, and at least one of these accesses is a write operation. As no two activation records can own the same memory location at the same time and hence cannot concurrently access the same memory location, such races cannot occur.

A method is only allowed to access a memory region if it owns that region. How can a method acquire ownership of a memory region? First of all, a constructor in a class `C` gains ownership of the fields of the new object declared in `C` right after calling the superclass constructor. For this reason, the constructor of the class `Account` shown below is allowed to initialize `balance` to zero. Secondly, a method can indicate in its precondition that it requires ownership of a particular memory region in order to execute successfully. Ownership of the field `f` of an object `e1` with value `e2` is denoted in assertions as `e1.f |-> e2`. Here, `e1` and `e2` are side-effect free, heap-independent Java expressions. This assertion is read as `e1.f` *points to* `e2`. For example, the precondition of `deposit` specifies that the method requires ownership of `this.balance`. Because of this precondition, the method body is allowed to read and write `this.balance`. The question mark (`?`) before the variable `b` indicates that the precondition does not impose any restrictions on the field's pre-state value, but binds this value to `b`. By naming the pre-state value, the postcondition can relate the field's post-state value to the pre-state value. Thirdly, a method can acquire ownership of a memory region by calling another method. As an example, consider the method `clone`. This method creates a new `Account` object named `copy` and then assigns to `copy.balance`. This assignment is allowed because the constructor's postcondition includes `this.balance |-> 0`. This postcondition specifies that ownership of the field `balance` of the new object is transferred from the constructor to its caller when the constructor terminates.

The set of memory locations owned by a method can not only grow as the method (symbolically) executes, but also shrink. First of all, at each method call ownership of the memory locations described by the callee's precondition is conceptually transferred from the caller to the callee. Secondly, when a method returns, the method loses ownership of all memory locations enumerated in its postcondition. Ownership of those locations is conceptually transferred from the method to its caller when it returns. For example, `deposit`'s postcondition returns ownership of `this.balance` to its caller. In both cases, if the method does not own the required memory location, symbolic execution terminates with a verification error.

```
class Account {
  int balance;

  public Account()
    //@ requires true;
    //@ ensures this.balance |-> 0;
  {
    super();
    this.balance = 0;
  }

  public void deposit(int amount)
    //@ requires this.balance |-> ?b;
    //@ ensures this.balance |-> b + amount;
  {
    this.balance += amount;
  }

  public int getBalance()
    //@ requires this.balance |-> ?b;
    //@ ensures this.balance |-> b &*& result == b;
  {
    return this.balance;
  }

  public Account clone()
    //@ requires this.balance |-> ?b;
    /*@ ensures this.balance |-> b &*& result != null &*&
              result.balance |-> b; @*/
  {
    Account copy = new Account();
    copy.balance = balance;
    return copy;
  }

  public void transfer(Account other, int amount)
    /*@ requires this.balance |-> ?b1 &*& other != null &*&
               other.balance |-> ?b2; @*/
    /*@ ensures this.balance |-> b1 - amount &*&
              other.balance |-> b2 + amount; @*/
  {
    balance -= amount;
    other.deposit(amount);
  }
}
```

Assertions can be subdivided into two categories: *pure* and *spatial* assertions. Pure assertions such as `0 <= x` specify constraints on local variables. Spatial assertions such as `o.f |-> v` on the other hand denote ownership of a heap subregion and information about that region. As explained in Section 2.3, producing and consuming a pure assertion involves respectively extending and checking the path condition. Consumption and production of a spatial assertion however affects the symbolic heap. That is, production of a spatial assertion corresponds to the acquisition of ownership by the current activation record of the memory regions described by the assertion. Therefore, producing such an assertion is implemented by adding the corresponding heap chunk to the heap. Consumption of a spatial assertion corresponds to the current activation record relinquishing ownership of the memory regions described by the assertion. It is hence implemented by searching the heap for a chunk that matches the assertion and by removing this chunk from the symbolic heap. An assertion `e1.f |-> e2` matches a chunk `C_g(v, w)` if f equals g, f is declared in class `C` and it follows from the path condition that the symbolic values of `e1` and `e2` are respectively equal to `v` and `w`. If no matching chunk is found, VeriFast reports a verification error: `No matching heap chunk`. For example, weakening the precondition of `getBalance` to `true` causes VeriFast to report an error at the field access `this.balance` (as the method does not have permission to read the field).

Multiple atomic assertions can be conjoined via the separating conjunction, denoted `&*&`. For example, `clone`'s postcondition specifies that the method returns ownership of `this.balance` and `result.balance` to its caller and that `result` is non-null. Semantically, `A &*& B` holds if both `A` and `B` hold and `A`'s footprint is disjoint from `B`'s footprint. The *footprint* of an assertion is the set of memory locations for which that assertion claims ownership. Consuming (respectively producing) `A &*& B` is implemented by first consuming (respectively producing) `A` and afterwards `B`. Note that if `A` is a pure assertion, then `A &*& A` is equivalent to `A`. However, this property does not necessarily hold for spatial assertions. In particular, as a method can only own a field once, it cannot give up ownership of that field twice. For that reason, the assertion `e1.f |-> _ &*& e1.f |-> _` is equivalent to `false`.

To gain a better understanding of ownership and ownership transfer, it is instructive to inspect the symbolic states encountered during verification of the method `transfer`. Open `Account.java`, place the cursor after the closing brace of the method `transfer` and press `Run to cursor`. As before, VeriFast starts symbolic execution of the method by constructing a symbolic pre-state that represents an arbitrary concrete pre-state. The symbolic store of this initial state contains fresh symbols for the parameters `this` and `other`, its path condition contains the assumption that `this` is non-null, and its symbolic heap is empty. To consider only program states that satisfy the precondition, VeriFast produces the three subassertions in the precondition from left to right. The pure assertion `other != null` is added to the path condition, while the symbolic heap is extended with two heap chunks, one for each spatial assertion. As the precondition does not constrain the values of the `balance` fields, VeriFast initializes both fields with fresh symbolic values. These values are respectively bound to `b1` and `b2`. Note that the first arguments of both heap chunks are equal to the symbolic values of respectively `this` and `other`. After producing the precondition, Verifast proceeds by executing the method body. To symbolically execute the assignment to `this.balance`, the verifier first checks that the method is allowed to read and write the field by checking that the heap contains a matching heap chunk. The heap contains the required chunk: $\mathsf{Account\_balance}(\mathsf{this}, \mathsf{b1})$. The effect of the assignment is reflected in the symbolic state by replacing this chunk with $\mathsf{Account\_balance}(\mathsf{this}, \mathsf{b1} - \mathsf{amount})$. To verify the subsequent method call, VeriFast first symbolically evaluates all method arguments. As explained in Section 2.3 symbolic execution of the method call itself consists of two steps: consumption of the precondition and afterwards production of the postcondition. Consumption of the precondition removes $\mathsf{Account\_balance}(\mathsf{other}, \mathsf{b2})$ from the symbolic heap. As `transfer` loses ownership of `other.balance` during the call, the verifier cannot (wrongly) assume that `other.balance`'s value is preserved by `deposit`. Moreover, `transfer` retains ownership of `this.balance` during the call. Therefore, VeriFast can deduce that the callee will not modify `this.balance`. Production of `deposit`'s postcondition adds the heap chunk $\mathsf{Account\_balance}(\mathsf{other}, \mathsf{b2} + \mathsf{amount})$ to the symbolic heap. `setBalance` has effectively *borrowed* ownership of `other.balance` from `transfer` in order to update its value. Finally, VeriFast checks that the postcondition of `transfer` holds by consuming it. Consumption of `transfer`'s postcondition removes the two heap chunks from the symbolic heap.

`Account.java` successfully verifies. It is useful though to consider how VeriFast responds to incorrect variations of the program:

– If we weaken the precondition of `deposit` to `true`, then VeriFast reports an error (`No matching heap chunks`) which indicates that the method might not own the field `this.balance` when it reads the field. A method is only allowed to read a field when it owns the permission to do so.
– If we strengthen the precondition of `deposit` by adding the requirement that `amount` must be non-negative, then verification of `transfer` fails (`Cannot prove condition`) with a precondition violation. Indeed, `transfer`'s precondition does not impose any constraint on `amount` and therefore the value passed to `deposit` might be negative.
– If we insert a bug in `deposit`'s body, e.g. we replace `+=` with `=`, then a postcondition violation (`Cannot prove amount == (b + amount)`) is reported by VeriFast. Although the method owns the correct memory location after execution of the method body (`this.balance`), the value of that location is not the one expected by the postcondition.
– If we weaken the postcondition of `deposit` to `true`, then verification of `transfer` fails with a postcondition violation (`No matching heap chunks`). The method call `other.deposit(amount)` then consumes ownership of the field `other.balance`, but does not return ownership of this field when it terminates.

Note that the ownership methodology described in this section does not impose any restrictions on aliasing. For example, an object is allowed to leak internal references to helper objects to client code. However, the methodology does impose restrictions on the use of aliases. In particular, a method can only dereference a reference (i.e. access a field) if it owns that reference.

4 Data Abstraction

Data abstraction is one of the pillars of object-oriented programming. That is, an object typically does not permit client code to directly access its internal state. Instead, the object provides methods that allow clients to query and update its state in a safe way. If an object hides its internals and forces clients to use a well-defined interface, then those clients cannot depend on internal implementation choices. This means that the object's implementation can be changed without having to worry about breaking clients, as long the observable behaviour remains the same. Moreover, clients cannot inadvertently break the object's internal invariants.

As an example of client code, consider the `main` method shown below:

```
class AccountClient {
  public static void main(String[] args)
    //@ requires true;
    //@ ensures true;
  {
    Account a = new Account(); a.deposit(100);
    Account b = new Account(); b.deposit(50);
    a.transfer(b, 20);

    int tmp = a.getBalance();
    assert tmp == 80;
  }
}
```

This client program creates and interacts with `Account` objects by calling the class' public methods. VeriFast can prove correctness of this program with respect to `Account`'s method contracts. Unfortunately, those method contracts are not implementation-independent as they mention the internal field `balance`. The correctness proof of `main` constructed by VeriFast therefore also indirectly depends on `Account`'s internal representation. If we would make internal modifications to the class, for example we could store the balance as a linked list of transactions instead of in a single field, we would have to update the method contracts and consequently have to reconsider the correctness of *all* clients. Moreover, if `Account` were an interface, then it would be impossible to specify the behavior of the methods by declaring their effect on fields as interfaces do not have fields.

Performing internal modifications to a class that do not change its observable behaviour should only require reverification of the class itself but should not endanger the correctness of its clients. To achieve this goal, we must answer the following question: How can we specify the observable behaviour of a class or interface without exposing its internal representation?

VeriFast's answer to this question is predicates [9]. More specifically, assertions describing the state associated with instances of a class can be hidden inside *predicates*. A predicate is a named, parameterized assertion. For example, consider the predicate definition shown below:

```
//@ predicate account(Account a, int b) = a.balance |-> b;
```

account is a predicate with two parameters named a and b, and with body a.balance |-> b. The assertion account(e1, e2) is a shorthand for the assertion e1.balance |-> e2. The body of the predicate is visible only inside the module defining the class Account. Outside of that module, a predicate is just an opaque container of permissions and constraints on its parameters.

The extra level of indirection provided by predicates allows us to write implementation-independent contracts: instead of directly referring to the internal fields, pre- and postconditions can be phrased in terms of a predicate. As an example, consider the new version of Account shown below. The implementation is exactly the same as before, but the contracts now specify the effect of each method with respect to the predicate account.

```
class Account {
  private int balance;

  public Account()
    //@ requires true;
    //@ ensures account(this, 0);
  {
    super();
    this.balance = 0;
    //@ close account(this, 0);
  }

  public void deposit(int amount)
    //@ requires account(this, ?b);
    //@ ensures account(this, b + amount);
  {
    //@ open account(this, b);
    this.balance += amount;
    //@ close account(this, b + amount);
  }

  public int getBalance()
    //@ requires account(this, ?b);
    //@ ensures account(this, b) &*& result == b;
  {
    //@ open account(this, b);
    return this.balance;
    //@ close account(this, b);
  }
}
```

From the client's point of view, the constructor returns an opaque bundle of permissions and constraints that relates the newly created object to `0`, a value representing the balance of the account. Although the client is unaware of the exact permissions and constraints inside the bundle, he or she can deduce from the contracts of `deposit` and `getBalance` that the bundle can be passed to those methods to respectively increase and query the value representing the balance associated with the account.

VeriFast by default does not automatically fold and unfold predicates. Instead, developers must explicitly use ghost statements to switch between the external, abstract view offered by the predicate and the internal definition of the predicate. The *close* ghost statement folds a predicate: it consumes the body of the predicate, and afterwards adds a chunk representing the predicate to the symbolic heap. For example, symbolic execution of the `close` statement in the constructor replaces the chunk $\mathsf{Account\_balance}(\mathsf{this}, 0)$ by $\mathsf{account}(\mathsf{this}, 0)$. Without the ghost statement, the constructor does not verify as the heap does not contain a chunk that matches the postcondition. The *open* ghost statement unfolds a predicate: it removes a heap chunk that represents the predicate from the symbolic heap and afterwards produces its body. For example, verification of `deposit` starts by producing the precondition: the chunk $\mathsf{account}(\mathsf{this}, 0)$ is added to the symbolic heap. However, when accessing the field `this.balance`, the heap must explicitly contain a chunk that represents ownership of the field. As the necessary chunk is nested inside `account(this, 0)`, the predicate must opened first. If we omit the ghost statement, VeriFast would no longer find a chunk that matches the field assertion `Account_balance(this, _)` on the heap and report an error. The `open` and `close` statement respectively correspond to Parkinson and Bierman's OPEN and CLOSE axioms [9, Section 3.2].

Inserting open and close ghost statements in order for VeriFast to find the required chunks on the heap is tedious. To alleviate this burden, programmers can mark certain predicates as *precise*. VeriFast automatically opens and closes precise predicates (in many cases) whenever necessary during symbolic execution. A predicate can be marked as precise by using a semicolon instead of comma somewhere in the parameter list. The semicolon separates the input from the output parameters. VeriFast syntactically checks that the values of the input parameters together with the predicate body uniquely determine the values of the output parameters. In our example, `account` can be marked precise by placing a semicolon between the parameters `a` and `b`. All open and close statements in the class `Account` shown above can be omitted once `account` is marked as precise. Although open and close statements need not be inserted explicitly in the program text, the symbolic execution trace does contain the corresponding steps.

`AccountClient.main` verifies against the new, more abstract method contracts for `Account`. To demonstrate that internal changes can be made to `Account` without having to change the method contracts — and hence without having to reverify *any* client code —, consider the alternative implementation of the class shown below.

```
class Transaction {
  int amount; Transaction next;

  public Transaction(int amount, Transaction next)
    //@ requires true;
    //@ ensures this.amount |-> amount &*& this.next |-> next;
  { this.amount = amount; this.next = next; }
}

/*@
predicate transactions(Transaction t; int total) =
  t == null ?
    total == 0
  :
    t.amount |-> ?amount &*& t.next |-> ?next &*&
    transactions(next, ?ntotal) &*& total == amount + ntotal;

predicate account(Account a; int b) =
  a.transactions |-> ?ts &*& transactions(ts, b);
@*/

class Account {
  private Transaction transactions;

  public Account()
    //@ requires true;
    //@ ensures account(this, 0);
  { transactions = null; }

  public void deposit(int amount)
    //@ requires account(this, ?b);
    //@ ensures account(this, b + amount);
  { transactions = new Transaction(amount, transactions); }

  public int getBalance()
    //@ requires account(this, ?b);
    //@ ensures account(this, b) &*& result == b;
  { return getTotal(transactions); }

  private int getTotal(Transaction t)
    //@ requires transactions(t, ?total);
    //@ ensures transactions(t, total) &*& result == total;
  {
    //@ open transactions(t, total);
    return t == null ? 0 : t.amount + getTotal(t.next);
  }
}
```

The method contracts are exactly the same as before. However, the implementation now stores the balance of the account as a linked list of `Transaction` objects instead of in single field. The body of `deposit` prepends a new `Transaction` object to the list, while `getBalance` traverses the list to compute the total sum. The new body of the predicate `account` reflects the changes in the implementation. `account(a, b)` now means that `a.transactions` is the head of linked, `null`-terminated list of `Transaction` objects whose total sum equals `b`.

Note that `account` is defined in terms of `transactions`, a *recursive* predicate. Such recursive predicates are crucial for specifying data structures without a static bound on their size. The body of `transactions` is a conditional assertion: if `t` is `null`, then `total` is equal to zero; otherwise, `total` is the sum of `t.amount` and the total of the remaining `Transaction`s at `t.next`. Just as in symbolic execution of an if statement, VeriFast separately examines both branches when producing and consuming a conditional assertion.

Predicates typically specify what it means for an object to be in a *consistent* state. For example, consistency of an `Account` object as described by the predicate `account` implies that the list of `Transaction`s is non-cyclic. As such, predicates play the role of object invariants [10]. VeriFast does not impose any built-in rules that state when invariants must be hold and when they can be temporarily violated. Instead, if a certain object is supposed to satisfy a particular invariant, then the method contract must explicitly say so.

5 Inheritance

A Java interface defines a set of abstract methods. For example, the interface `java.util.List` defines methods for modifying and querying `List` objects such as `add`, `remove`, `size` and `get`. Each non-abstract class that implements the interface must provide an implementation for each interface method. For example, `ArrayList` implements `get` by returning the object stored at the given index in its internal array, while `LinkedList` implements the method by traversing a linked list of nodes.

In order to be able to modularly verify client code, each interface method must be annotated with a method contract. A straightforward, but naive approach to specifying interfaces is phrasing the method contracts in terms of a predicate with a fixed definition. For example, applying this approach to (part of) the interface `List` would look as follows:

```
//@ predicate list(List l, int size);

interface List {
  public void add(Object o);
    //@ requires list(this, ?size);
    //@ ensures list(this, size + 1);
}
```

This approach is problematic as a predicate can only have a single definition. Yet multiple classes can implement `List` and each such class requires a different definition of the predicate. For example, `ArrayList` requires the predicate to include ownership of the internal array, while `LinkedList` requires the predicate to contain ownership of the linked sequence of nodes.

One could solve the aforementioned problem by case splitting in the contracts on the dynamic type of `this`. In our example, we could check whether `this` is an `ArrayList` or a `LinkedList` as shown below.

```
interface List {
  public void add(Object o);
    /*@ requires this.getClass() == ArrayList.class ?
      arraylist(this, ?size)
        :
      linkedlist(this, ?size); @*/
    /*@ ensures this.getClass() == ArrayList.class ?
      arraylist(this, size + 1)
        :
      linkedlist(this, size + 1); @*/
}
```

The complete set of subclasses is typically not known when writing the interface. In particular, clients of the interface can define their own subclasses. Each time a new subclass is added, the contracts of the interface methods must be extended with another case (which means *all* clients must be reverified). Clearly, this approach is non-modular and does not scale.

VeriFast solves the conundrum described above via dynamically bound *instance predicates* [11]. An instance predicate is a predicate defined inside a class or interface. For example, the method contracts of `List` can be phrased in terms of the instance predicate `list` as follows:

```
interface List {
  //@ predicate list(int size);

  public void add(Object o);
    //@ requires list(?size);
    //@ ensures list(size + 1);
}
```

Just like an instance method, an instance predicate does not have a single, closed definition. Instead, each subclass must override the instance predicate's definition and provide its own body. For example, the body of `list` in `ArrayList` involves ownership of the internal array (i.e. `array_slice` denotes ownership of the array itself) and states that `size` lies between zero and the length of the array as shown below. Note that the variable `this` in the body of the instance predicate refers to the target object of the predicate.

```
class ArrayList implements List {
  /*@
  predicate list(int size) =
    this.elements |-> ?a &*& this.size |-> size &*&
    array_slice<Object>(a, 0, a.length, _) &*&
    0 <= size &*& size <= a.length;
  @*/

  private Object[] elements;
  private int size;
  ...
}
```

The body of `list` in `LinkedList` on the other hand states that `first` is the head of a valid sequence of nodes ending in `null` with `size` elements. The body is phrased in terms of the recursive, non-instance predicate `lseg`. The assertion `lseg(n1, n2, s)` denotes that `n1` is the start of a valid sequence of nodes of length `s` ending in (but not including) `n2`.

```
/*@
predicate lseg(Node first, Node last; int size) =
  first == last ?
    size == 0
  :
    first.value |-> _ &*& first.next |-> ?next &*&
    lseg(next, last, ?nsize) &*& size == nsize + 1;
@*/

class LinkedList implements List {
  /*@ predicate list(int size) =
    this.first |-> ?first &*& this.size |-> size &*&
    lseg(first, null, size); @*/

  private Node first;
  private int size;
  ...
}
```

An instance predicate has multiple definitions, one per subclass. For that reason, each heap chunk corresponding to an instance predicate has an additional parameter indicating what version of the predicate the chunk represents. For example, heap chunks corresponding to the predicate `list` are of the form $\mathsf{List\#list(target, C, size)}$, where target and size are the symbolic values of respectively the implicit target object and the parameter `size` and where C is a symbolic value that represents the class containing the definition of the predicate. Hence, the chunk $\mathsf{List\#list(l, ArrayList, s)}$ states that l refers to a valid list with s elements, where the definition of *valid list* is the one given in the class `ArrayList`.

When producing and consuming an instance predicate assertion of the form `e0.p(e1, ..., en)`, the verifier generally selects the dynamic type of `e0` as the class containing `p`'s definition. As an example, consider the steps in the symbolic execution of the method `addNull` shown below:

```
public static void addNull(List l)
  //@ requires l.list(?size);
  //@ ensures l.list(size + 1);
{ l.add(null); }
```

Open `Lists.java`, place the cursor after the closing brace of `addNull` and press `Run to cursor`. As shown in the symbolic execution trace, production of the precondition adds the chunk List#list(l, getClass(l), size) to the symbolic heap. Here, getClass is a function that represents the dynamic type of its argument. To symbolically execute the method call `l.add(null)`, VeriFast first consumes `add`'s precondition, and subsequently produces its postcondition. As `l.add(null)` is a dynamically bound call, the version of the instance predicate denoted by its pre- and postcondition is the one defined in the dynamic type of the target object, `l`.

There are a number of exceptions to the rule that the dynamic type of the target object is used as the version of an instance predicate chunk. First of all, when opening or closing an instance predicate, the definition in the static type of the target object is used. For example, if the static type of `l` is `LinkedList`, then execution of the statement `close l.list(5);` first consumes the body of `list` as defined in `LinkedList` and afterwards adds the chunk `List#list(l, LinkedList, 5)` to the symbolic heap. Secondly, when verifying the body of an instance method defined in a class `C` or a statically bound call of a method (e.g. super and constructor calls), instance predicate assertions of the form `p(e1, ..., en)` in the contract where the implicit argument `this` has been omitted are treated as statically bound meaning that the term representing the definition of `p` is equal to `C`. For example, when verifying `List.add` in `LinkedList`, production of the precondition produces the chunk List#list(this, LinkedList, size). This is sound since VeriFast checks that all methods are overridden [12].

VeriFast can prove correctness of the method `addNull` with respect to `List`'s method contracts. Correctness of `addNull` implies that there do not exist values for `l` or states of the heap that trigger an assertion violation during execution of the method body. However, consider the class `BadList` shown below.

Even though each method in `BadList` satisfies its method contract, execution of `BadList.main` triggers an assertion violation. The cause of the problem is the fact that the contract used for verifying the method call `l.add(null)` in `addNull` and the contract used for verifying the implementation of `BadList.add` (which is executed when the dynamic type of `l` is `BadList`) are not compatible. In particular, the precondition of `BadList.add` states that the method should never be called, while `List.add`'s precondition does allow calls provided `l` is a valid list.

To avoid problems such as the one described above, VeriFast checks that the contract of each overriding method is *compatible* [11] with the contract of its overridden method. VeriFast checks that the contract of an overriding method with precondition P and postcondition Q is compatible with the contract of an overridden method with precondition P' and postcondition Q' by checking that the body of the overridden method could statically call the overriding method: the verifier first produces P', then consumes P and afterwards produces Q, and finally consumes Q'. When verifying `BadList`, Verifast reports that the contract of `BadList.add` is not compatible with the contract of `List.add` as `false` is not provable after producing the chunk $\mathsf{List\#list}(\mathsf{this}, \mathsf{BadList}, 0)$.

```
class BadList implements List {
  /*@ predicate list(int size) = true; @*/

  public BadList()
    //@ requires true;
    //@ ensures list(0);
  {
    //@ close list(0);
  }

  public void add(Object o)
    //@ requires false;
    //@ ensures true;
  { assert false; }

  public static void main(String[] args)
    //@ requires true;
    //@ ensures true;
  {
    BadList bad = new BadList();
    addNull(bad);
  }
}
```

6 Inductive Data Types and Fixpoints

The contract of `List.add` is incomplete. More specifically, its postcondition states that `add` increments the number of elements in the list by one, but it does not specify that the object `o` should be added to the end of the list. As a consequence, an implementation that prepends `o` to the front of the list or even inserts a different object altogether is considered to be correct.

To rule out such implementations, we must strengthen `add`'s postcondition. However, the current signature of the instance predicate `list` does not allow us to do so as it only exposes the number of elements in the list via the parameter `size`. The predicate does not expose the elements themselves and their positions.

To allow developers to specify rich properties, VeriFast supports inductive data types and fixpoints. For example, we can represent a sequence using the inductive data type `list`:

```
//@ inductive list<t> = nil | cons(t, list<t>);
```

This declaration declares a type `list` with two constructors: `nil` and `cons`. `nil` represents the empty sequence, while `cons(h, t)` represents a concatenation of a head element `h` and a tail list `t`. The definition is generic in the type of the list elements (here `t`). For example, the sequence $1, 2, 3$ can be written as `cons(1, cons(2, cons(3, nil)))`.

A fixpoint is a total, mathematical function that operates on an inductively defined data type. For example, consider the fixpoints `length`, `nth` and `append` shown below. The body of each of these functions is a switch statement over one of the inductive arguments. The function `length` for instance returns zero if the sequence is empty; otherwise, it returns the length of the tail plus one. Note that `default_value<t>` is a built-in function that returns the default value for a particular type `t`.

To ensure that fixpoints are well-defined, VeriFast syntactically checks that they terminate. In particular, VeriFast enforces that whenever a fixpoint `g` is called in the body of a fixpoint `f` that either `g` appears before `f` in the program text or that the call decreases the size of the inductive argument (i.e. the argument switched on by the fixpoint's body). For example, the call `length(xs0)` in the body of `length` itself is allowed because `xs0` is a subcomponent of `xs` (and hence smaller than `xs` itself).

```
/*@
fixpoint int length<t>(list<t> xs) {
  switch (xs) {
    case nil: return 0;
    case cons(x, xs0): return 1 + length(xs0);
  }
}

fixpoint t nth<t>(int n, list<t> xs) {
  switch (xs) {
    case nil: return default_value<t>;
    case cons(x, xs0): return n == 0 ? x : nth(n - 1, xs0);
  }
}

fixpoint list<t> append<t>(list<t> xs, list<t> ys) {
  switch (xs) {
    case nil: return ys;
    case cons(x, xs0): return cons(x, append(xs0, ys));
  }
}
@*/
```

We can now make the specification of the interface `List` complete as shown below. First of all, we modify the signature of the instance predicate `list` such that it exposes a sequence of `Objects`. Secondly, we refine the method contracts by defining the return value and the effect of each method in terms of a fixpoint function on the exposed sequence of objects. For example, the postcondition of `add` specifies that the method adds `o` to the end of the list.

```
interface List {
  //@ predicate list(list<Object> elems);

  public void add(Object o);
    //@ requires list(?elems);
    //@ ensures list(append(elems, cons(o, nil)));

  public int size();
    //@ requires list(?elems);
    //@ ensures list(elems) &*& result == length(elems);

  public Object get(int i);
    //@ requires list(?elems) &*& 0 <= i && i < length(elems);
    //@ ensures list(elems) &*& result == nth(i, elems);
}
```

The specification of the interface `List` is idiomatic in VeriFast. That is, specifying the behaviour of a class or interface typically involves the following steps:

1. Define an inductive data type `T` to represent the abstract state of instances of the class. In our example, we defined the inductive data type `list` to represent the state of `List` objects.
2. Define an instance predicate `p` with a single parameter of type `T`. For example, the interface `List` defines the instance predicate `list`. Its parameter `elems` is of type `list`.
3. The postcondition of each constructor guarantees that the predicate holds.
4. The contract of each instance method requires that the predicate holds on entry to the method and guarantees that the predicate holds again when the method terminates. The effect and return value of the method are related to the abstract state via fixpoint functions. For example, `size`'s postcondition relates the return value to the sequence of elements via the function `length`.

Each inductive data type constructor has a corresponding first-order function. For example, the inductive data type `list` has two corresponding functions named nil and cons. Instances of inductive data types are encoded as applications of these functions. For example, the expression `cons(o, nil)` in the postcondition of `add` symbolic evaluates to the term $\mathsf{cons}(\mathsf{o}, \mathsf{nil})$. The fact that all constructors are distinct is given as an axiom to the SMT solver. For the inductive data type `list`, the solver can hence deduce that $\mathsf{nil} \neq \mathsf{cons}(\mathsf{h}, \mathsf{t})$, for arbitrary h and t.

Each fixpoint function also has an associated first-order function. For example, the fixpoint `length` has a corresponding first-order function named length. Applications of fixpoints are encoded as applications of the corresponding function. The behaviour of the fixpoint is encoded in the SMT solver via one or more axioms that relate the function to the fixpoint's body. In particular, if the body of the fixpoint is a switch over one of its arguments, then one axiom is generated for each case. For example, the behavior of length is encoded via the following axioms:

$$\mathsf{length}(\mathsf{nil}) = 0$$
$$\forall \mathsf{h}, \mathsf{t} \bullet \mathsf{length}(\mathsf{cons}(\mathsf{h}, \mathsf{t})) = 1 + \mathsf{length}(\mathsf{t})$$

The SMT solver can use these axioms to simplify terms and formulas involving fixpoint functions. For example, the aforementioned axioms allow the solver to deduce that $\mathsf{length}(\mathsf{cons}(1, \mathsf{cons}(2, \mathsf{l})))$ and $2 + \mathsf{length}(\mathsf{l})$ are equal.

7 Lemmas

To determine whether a particular formula follows from the path condition — for example when consuming a pure assertion — VeriFast relies on an SMT solver. The SMT solver however does not perform induction. For that reason, it can fail to prove properties that require proof by induction. For example, proving that the function `append` is associative requires induction.

Lemma functions allow developers to prove properties about their fixpoints and predicates, and allow them to use these properties when reasoning about programs. A lemma is a method without side effects marked `lemma`. The contract of a lemma function corresponds to a theorem, its body to the proof, and a lemma call to an application of the theorem. VeriFast has two types of lemma methods: pure lemmas and spatial lemmas.

A lemma is *pure* if its contract does not contain spatial assertions. The contract of a pure lemma corresponds to a theorem that states that the precondition implies the postcondition for all possible values of the lemma parameters. The lemma `append_assoc` shown below is an example of a pure lemma that states that applying the fixpoint `append` is associative. The lemma's body proves the theorem by induction on `xs`. More specifically, the case `nil` of the switch statement corresponds to the base case, while the case `cons` corresponds to the inductive step. The recursive call in the case `cons` is an application of the induction hypothesis.

```
/*@
lemma void append_assoc<t>(list<t> xs, list<t> ys, list<t> zs)
  requires true;
  ensures append(append(xs, ys), zs) == append(xs, append(ys, zs));
{
  switch (xs) {
    case nil:
    case cons(h, t): append_assoc(t, ys, zs);
  }
}
@*/
```

Contrary to pure lemmas, *spatial* lemmas can mention spatial assertions in their method contracts. The contract of a spatial lemma corresponds to a theorem that states that the symbolic state described by the precondition is equivalent to the one described by the postcondition. Equivalent means that the symbolic state described by the postcondition can be reached by applying a finite number of open and close statements to the state described by the precondition. A spatial lemma does not modify the values in the heap, but only rewrites the representation of the symbolic state. Spatial lemmas are crucial whenever the symbolic state is required to have a particular form but the current state cannot be rewritten to the required one by a statically known number of open and close statements. Moreover, a spatial lemma allows a class to expose properties to clients without having to reveal its internal representation. As an example, consider the spatial lemma `lseg_merge` shown below. This lemma states that two list segments, one from `a` to `b` and another from `b` to `c`, are equivalent to a single list segment from `a` to `c`, provided there exists an additional list segment from `c` to `null`.

```
/*@
lemma void lseg_merge(Node a, Node b, Node c)
  requires lseg(a, b, ?elems1) &*& lseg(b, c, ?elems2) &*&
    lseg(c, null, ?elems3);
  ensures lseg(a, c, append(elems1, elems2)) &*&
    lseg(c, null, elems3);
{
  open lseg(a, b, elems1);
  open lseg(c, null, elems3);
  if(a != b) lseg_merge(a.next, b, c);
}
@*/
```

VeriFast checks termination of a lemma method by allowing only direct recursion and by checking each recursive call as follows: first, if, after consuming the precondition of the recursive call, a field chunk is left in the symbolic heap, then the call is allowed. This is induction on heap size. Otherwise, if the body of the lemma method is a switch statement on a parameter whose type is an inductive data type, then the argument for this parameter in the recursive call must be a constructor argument of the caller's argument for the same parameter. This is induction on an inductive parameter. Finally, if the body of the lemma function is not such a switch statement, then the first heap chunk consumed by the precondition of the callee must have been obtained from the first heap chunk consumed by the precondition of the caller through one or more open operations. This is induction on the derivation of the first conjunct of the precondition.

The method `LinkedList.add` shown below adds an object `o` to the end of the list. If the list is empty, then the method body assigns a new node holding `o` to `first`; otherwise, the method transverses the linked list in a loop until it reaches the last node and assigns a new node holding `o` to the `next` pointer of the last node. The method calls the lemmas `lseg_merge` and `append_assoc` to prove that the loop body preserves the loop invariant and to prove that the postcondition follows from the invariant and the negation of the condition.

```
public void add(Object o)
  //@ requires list(?elems);
  //@ ensures list(append(elems, cons(o, nil)));
{
  if(first == null) {
    //@ open lseg(first, null, _);
    first = new Node(null, o);
  } else {
    Node first = this.first, curr = this.first;
    while(curr.next != null)
      /*@ invariant curr != null &*&
          lseg(first, curr, ?elems1) &*& lseg(curr, null, ?elems2) &*&
          elems == append(elems1, elems2);
      @*/
      //@ decreases length(elems2);
    {
      //@ Object v = curr.value;
      //@ Node oldcurr = curr;
      curr = curr.next;
      //@ open lseg(curr, null, _);
      //@ lseg_merge(first, oldcurr, curr);
      //@ append_assoc(elems1, cons(v, nil), tail(elems2));
    }
    //@ open lseg(null, null, _);
    Node nn = new Node(null, o); curr.next = nn;
    //@ lseg_merge(first, curr, null);
    //@ append_assoc(elems1, elems2, cons(o, nil));
  }
  this.size++;
}
```

Like many other verification tools, VeriFast requires each loop to be annotated with a *loop invariant* (keyword `invariant`), describing the assertion that must hold right before evaluation of the condition in each iteration. For example, the loop invariant in the method `add` states (1) that `curr` is non-null, (2) that the method owns two valid list segments, one from `first` to `curr` and another from `curr` to `null` and (3) that the concatenation of the elements of both segments is equal to the elements of the original list. VeriFast verifies a loop as follows.

1. First, the tool consumes the loop invariant. This step proves that the invariant holds on entry to the loop.
2. Then, it removes the remaining heap chunks from the heap (but it remembers them).
3. Then, it assigns a fresh logical symbol to each local variable that is modified in the loop body.
4. Then, it produces the loop invariant.
5. Then, it performs a case split on the loop condition:

- If the condition is true, it verifies the loop body and afterwards consumes the loop invariant. This step proves that the loop body preserves the invariant. After this step, this execution path is finished.
- If the condition is false, VeriFast puts the heap chunks that were removed in Step 2 back into the heap and verification continues after the loop.

Notice that this means that the loop can access only those heap chunks that are mentioned in the loop invariant.

By default, VeriFast does not check loop termination. However, developers may provide an optional *loop measure* to force the verifier to check termination of a particular loop. A loop measure (keyword `decreases`) is an integer-valued expression. If the value of the expression decreases in each loop iteration but never becomes negative, then it follows that the loop terminates. In the example, the length of the list segment from `curr` to `null` is the measure.

The VeriFast distribution includes a specification library that contains many commonly used inductive data types (such as `list`, `option` and `pair`), fixpoints (such as `append`, `take` and `drop`) and lemmas (such as `length_nonnegative` and `append_assoc`). As a consequence, developers can use existing definitions to define abstract states and the effect of methods.

8 Related Work

VeriFast is a separation logic-based program verifier. Separation logic [13,14,1] is an extension of Hoare logic [15] targeted at reasoning about imperative programs with shared mutable state. It extends Hoare logic by adding spatial assertions to describe the structure of the heap. Spatial assertions allow for *local reasoning*: the specification of a sequence of statements S only needs to mention heap locations accessed by S. The effect of S on other heap locations can be inferred via the *frame rule*:

$$\frac{\{P\}\ S\ \{Q\}}{\{P * R\}\ S\ \{Q * R\}}$$

The separating conjunction $P * R$ (written `P &*& R` in VeriFast) holds if both P and R hold and the part of the heap described by P is disjoint from the part of the heap described by R. The frame rule then states that if the separating conjunction $P * R$ holds before execution of a statement S and S's precondition does not require R, then R still holds after execution of S. VeriFast's symbolic heap represents a separating conjunction of its heap chunks. The verifier implicitly applies the frame rule for example when verifying a method call: the chunks not consumed by the callee's precondition are stable under the call and remain in the heap.

Parkinson and Bierman [9,11] extend the early work on separation logic of O'Hearn, Reynolds and Yang with abstract predicates and abstract predicate families to allow for local reasoning for object-oriented programs. VeriFast supports abstract predicate families in the form of instance predicates. Each instance predicate override in a class `C` corresponds to a predicate family member definition for `C`. Contrary to Parkinson and Bierman's work, our verifier does not support widening and narrowing to change the arity of a predicate family definition.

The specification of a method in Parkinson and Bierman's paper [11] consists of a static and a dynamic contract. Those contracts are respectively used to reason about statically and dynamically bound calls. Developers need not explicitly write both static and dynamic contracts in VeriFast. Instead, a single "polymorphic" contract suffices and the tool interprets this contract differently for statically and dynamically bound calls [12].

Berdine, Calcagno and O'Hearn [16] demonstrate that a fragment of separation logic (i.e. without implication and magic wand) is amenable to automatic static checking by building a verifier, called Smallfoot, for a small procedural language. Smallfoot checks that each procedure in the program satisfies its separation logic specification via symbolic execution. The symbolic state consists of a spatial formula and a pure formula. VeriFast extends the ideas of Berdine *et al.* to full-fledged programming languages (Java and C). The symbolic heap corresponds to Smallfoot's spatial formula and the symbolic store and the path condition to the pure formula. Smallfoot contains a small number of built-in predicates, and rules for rewriting symbolic states involving these predicates. Contrary to VeriFast, developers do not need to explicitly open and close predicates and need not provide loop invariants because of these built-in rules.

In addition to VeriFast, Smallfoot has inspired several other separation logic-based program verifiers. For example, jStar [17] is another semi-automatic program verifier for Java. Unlike VeriFast, jStar automatically infers certain loop invariants. Whether the right loop invariant can be inferred depends on the abstraction rules provided by the developer in a separate file. Heap-Hop [18] is an extension of Smallfoot targeted at proving memory safety and deadlock freedom of concurrent programs that rely on message passing. HIP [19] is a variant of Smallfoot that focuses on automatically proving size properties (e.g. that `List.add` increases the size of the list by one). Tuerk [20] has developed HOLFoot, a port of Smallfoot to HOL. He has mechanically proven that HOLFoot is sound. An interesting feature of HOLFoot is that one can resort to interactive proofs in HOL4 when the tool is unable to construct a proof automatically. As it is challenging for fully automatic tools to prove full functional correctness, several researchers [21,22,23] have used separation logic within interactive proof assistants. While this approach typically requires more input from the developer, it has resulted in a number of impressive achievements. For example, Tuch *et al.* [23] report on verifying the memory allocator of the L4 microkernel.

Besides separation logic, the research literature contains many other approaches for reasoning about imperative programs with shared mutable state such as the Boogie methodology [12], dynamic frames [24], regional logic [25], data groups [26], the VCC methodology [27], universe types [28], dynamic logic [29], etc. These approaches are often implemented in a corresponding program verifier [30,31,32,33,34,29]. A strategy for dealing with aliasing common to many of these approaches is to represent the heap as a global map from addresses to values. As a method can change the heap, verification of a method call entails havocking (i.e. assigning a fresh value to) the global map. Each approach then provides a way to relate the value of the map before the call to its new value. For example, in the dynamic frames approach each method includes a modifies clause describing a set of addresses. After each call, the verifier assumes via a quantifier that the new map is equal to the old map except at addresses included in the modified set.

Before starting the development of VeriFast, we contributed to several program verifiers based on verification condition generation and automated theorem proving [35,32,36]. However, the verification experience provided by those tools left us frustrated for three reasons. First of all, verification can be slow (which we believe is caused by the quantifiers needed to encode frame properties). Secondly, it can be hard to diagnose why verification fails. Finally, automated theorem proving can be unpredictable, as small changes to the input can cause huge differences in verification time (or even whether the proof succeeds at all). For that reason, we put a very strong premium on predictable performance and diagnosability when designing VeriFast. The only quantifiers that are made available to the SMT solver are those that axiomatize the inductive data types and fixpoint functions; these behave very predictably. The VeriFast IDE allows developers to diagnose verification errors by inspecting the symbolic states on the path leading to the error.

9 Conclusion

VeriFast is a separation logic-based program verifier for Java. In this paper, we have informally explained the key features of this verifier.

Based on our experience with VeriFast, we identify three main areas of future work. First of all, in the separation logic ownership system used by VeriFast a method either exclusive owns a memory region (meaning that it has permission to read and arbitrarily modify the state of that region), partially owns a region (meaning it has permission to read but not write that region), or does not have any permission on that region at all. However, for some programs — in particular concurrent ones — more precise kinds of permissions are required. For example, it is not possible to directly express a permission that allows a memory location to be incremented but not decremented (which may be crucial for proving correctness of a ticketed lock). As shown by Owicki and Gries [37] and more recently by Jacobs and Piessens [38,39], arbitrarily "permissions" can be constructed indirectly by adding ghost state to lock invariants and by using fractional permissions. However, using ghost state does not lead to intuitive proofs. Therefore, we consider designing a more flexible ownership system where the developer can construct his own permissions to relate the current state of a memory region to earlier, observed states a key challenge for the future. Concurrent abstract predicates [40] and superficially substructural types [41] form promising directions in this area.

VeriFast sometimes requires developers to explicitly fold and unfold predicate definitions. Moreover, to prove inductive properties, lemmas must be written and called whenever the property is needed. A second important challenge to be addressed in future work is reducing the annotation overhead by automatically inferring more open and close statements and lemma calls.

Finally, VeriFast supports only a subset of Java. For example, generics are not supported yet by the tool. In order for VeriFast to be applicable to large Java programs, we must extend the supported subset of Java.

References

1. Parkinson, M., Bierman, G.: Separation Logic for Object-Oriented Programming. In: Clarke, D., Noble, J., Wrigstad, T. (eds.) Aliasing in Object-Oriented Programming. LNCS, vol. 7850, Springer, Heidelberg (2013)
2. Jacobs, B., Smans, J., Piessens, F.: A Quick Tour of the VeriFast Program Verifier. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 304–311. Springer, Heidelberg (2010)
3. Jacobs, B., Smans, J., Piessens, F.: The VeriFast Program Verifier: A Tutorial (2012)
4. Jacobs, B., Smans, J., Piessens, F.: Verification of imperative programs: The VeriFast approach. a draft course text. Technical Report CW578, Department of Computer Science, K.U.Leuven (2010)
5. Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A.: The Java Language Specification (2012)
6. Meyer, B.: Applying "Design by Contract". IEEE Computer 25(10) (1992)
7. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
8. Boyland, J.: Checking Interference with Fractional Permissions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 55–72. Springer, Heidelberg (2003)
9. Parkinson, M., Bierman, G.: Separation logic and abstraction. In: Symposium on Principles of Programming Languages (POPL). ACM (2005)
10. Parkinson, M.: Class invariants: The end of the road? In: International Workshop on Aliasing, Confinement and Ownership in Object-oriented Programming, IWACO (2007)
11. Parkinson, M., Bierman, G.: Separation logic, abstraction and inheritance. In: Symposium on Principles of Programming Languages (POPL). ACM (2008)
12. Barnett, M., DeLine, R., Fähndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. Journal of Object Technology 3(6) (2003)
13. O'Hearn, P., Reynolds, J., Yang, H.: Local Reasoning about Programs that Alter Data Structures. In: Fribourg, L. (ed.) CSL 2001. LNCS, vol. 2142, pp. 1–19. Springer, Heidelberg (2001)
14. Reynolds, J.: Separation logic: A logic for shared mutable data structures. In: Symposium on Logic in Computer Science, LICS (2002)
15. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10) (1969)
16. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic Execution with Separation Logic. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 52–68. Springer, Heidelberg (2005)

17. Distefano, D., Parkinson, M.: jStar: Towards practical verification for Java. In: Object-Oriented Programming, Systems, Languages and Applications (OOPSLA). ACM (2008)
18. Villard, J., Lozes, É., Calcagno, C.: Proving Copyless Message Passing. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 194–209. Springer, Heidelberg (2009)
19. Nguyen, H.H., David, C., Qin, S.C., Chin, W.-N.: Automated Verification of Shape and Size Properties Via Separation Logic. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 251–266. Springer, Heidelberg (2007)
20. Tuerk, T.: A Formalisation of Smallfoot in HOL. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 469–484. Springer, Heidelberg (2009)
21. Nanevski, A., Morrisett, G., Shinnar, A., Govereau, P., Birkedal, L.: Ynot: Reasoning with the awkward squad. In: International Conference on Functional Programming (ICFP). ACM (2008)
22. Hobor, A., Appel, A.W., Nardelli, F.Z.: Oracle Semantics for Concurrent Separation Logic. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 353–367. Springer, Heidelberg (2008)
23. Tuch, H., Klein, G., Norrish, M.: Types, bytes, and separation logic. In: Symposium on Principles of Programming Languages (POPL). ACM (2007)
24. Kassios, I.T.: Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 268–283. Springer, Heidelberg (2006)
25. Banerjee, A., Naumann, D.A., Rosenberg, S.: Regional Logic for Local Reasoning about Global Invariants. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 387–411. Springer, Heidelberg (2008)
26. Leino, K.R.M.: Data groups: Specifying the modification of extended state. In: Object-Oriented Programming, Systems, Languages and Applications (OOPSLA). ACM (1998)
27. Cohen, E., Moskal, M., Schulte, W., Tobies, S.: Local Verification of Global Invariants in Concurrent Programs. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 480–494. Springer, Heidelberg (2010)
28. Dietl, W.M.: Universe Types: Topology, Encapsulation, Genericity, and Tools. PhD thesis, ETH Zurich, Switzerland (2009)
29. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
30. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: the Spec# experience. Communications of the ACM 54(6) (2011)
31. Leino, K.R.M.: Dafny: An Automatic Program Verifier for Functional Correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 348–370. Springer, Heidelberg (2010)
32. Smans, J., Jacobs, B., Piessens, F., Schulte, W.: An Automatic Verifier for Java-Like Programs Based on Dynamic Frames. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 261–275. Springer, Heidelberg (2008)
33. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: Programming Language Design and Implementation (PLDI). ACM (2002)
34. Dahlweid, M., Moskal, M., Santen, T., Tobies, S., Schulte, W.: VCC: Contract-based modular verification of concurrent C. In: International Conference on Software Engineering (ICSE). IEEE (2009)

35. Jacobs, B., Piessens, F., Smans, J., Leino, K.R.M., Schulte, W.: A programming model for concurrent object-oriented programs. ACM Transactions on Programming Languages and Systems 31(1) (2008)
36. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames. ACM Transactions on Programming Languages and Systems 34(1) (2012)
37. Owicki, S., Gries, D.: Verifying properties of parallel programs: an axiomatic approach. Communications of the ACM 19(5) (1976)
38. Jacobs, B., Piessens, F.: Expressive modular fine-grained concurrency specification. In: Symposium on Principles of Programming Languages (POPL). ACM (2011)
39. Jacobs, B., Piessens, F.: Dynamic owicki-gries reasoning using ghost fields and fractional permissions. Technical Report CW549, Department of Computer Science, K.U.Leuven (2009)
40. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent Abstract Predicates. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 504–528. Springer, Heidelberg (2010)
41. Krishnaswami, N.R., Turon, A., Dreyer, D., Garg, D.: Superficially substructural types. In: International Conference on Functional Programming (ICFP). ACM (2012)