The VeriFast Program Verifier
A Tutorial for Java Card Developers

Bart Jacobs Jan Smans Pieter Philippaerts Frank Piessens

September 19, 2011

1 Introduction

VeriFast is a program verification tool for verifying certain correctness properties of single-threaded and multithreaded C, Java, and Java Card programs. In this tutorial, we introduce VeriFast's support for Java Card (Classic) programs.

When verifying a Java Card program, VeriFast reads the program's source code files and reports either "0 errors found" or indicates the location of a potential error. If the tool reports "0 errors found", this means[1] that the program

- does not perform null pointer dereferences;
- does not access arrays with an index that is out of bounds;
- does not perform arithmetic overflow;
- does not perform illegal calls of Java Card API methods;
- does not violate any **assert** statements in the program; and
- complies with method preconditions, method postconditions, loop invariants, and other assertions specified by the programmer in the form of special comments (known as *annotations*) in the source code.

We will now proceed to introduce the tool's features step by step. To try the examples and exercises in this tutorial yourself, please download the release from the VeriFast website at

`http://www.cs.kuleuven.be/~bartj/verifast/`

. You will find a command-line version of the tool (`verifast.exe`), and a version that presents a graphical user interface (`vfide.exe`).

2 Method Contracts; The VeriFast IDE

Let's start with a very simple program:

```
class Program {
   static short min(short x, short y, short z) {
      if (x < y) {
         if (x < z) {
            return x;
         } else {
```

[1] There are a few known reasons (known as *unsoundnesses*) why the tool may sometimes incorrectly report "0 errors found"; see the "Known unsoundnesses" section in the Reference Manual. There may also be unknown unsoundnesses.

```
            return z;
         }
      } else {
         return y;
      }
   }
}
```

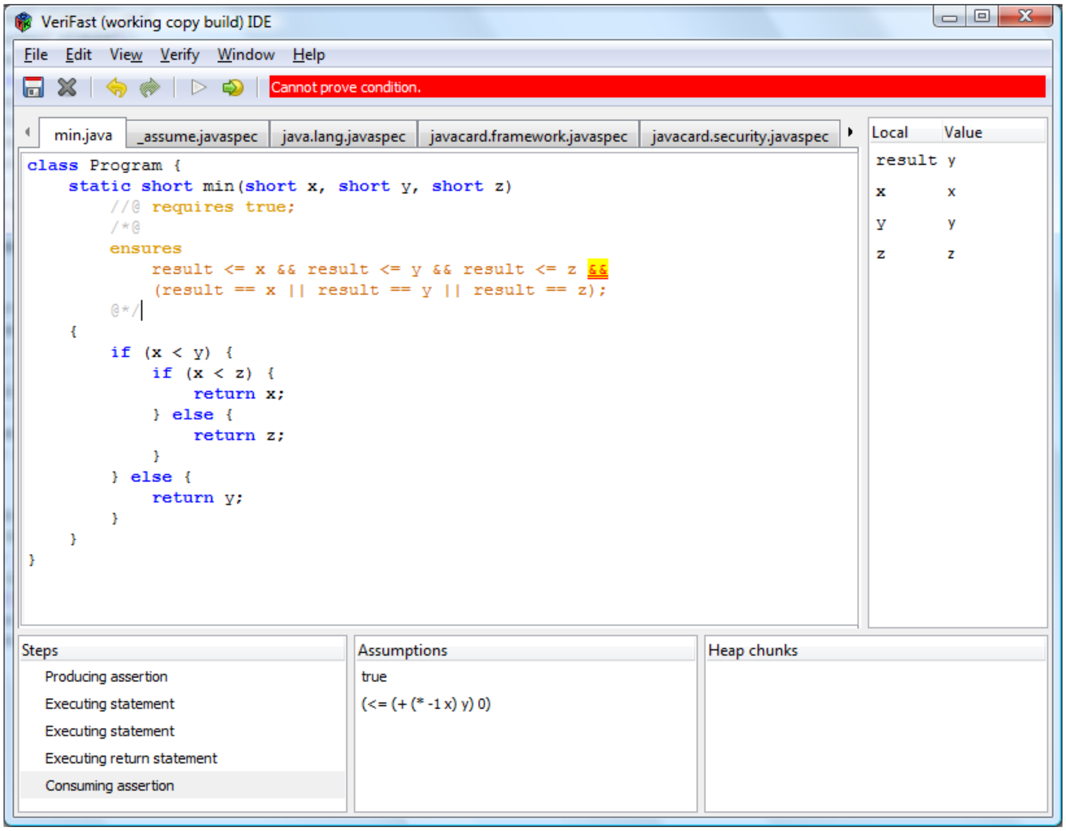
Does this method indeed return the smallest of `x`, `y`, and `z`? Let's ask VeriFast to check this. To do so, we need to insert a *method contract* into the program, as follows:

```
class Program {
   static short min(short x, short y, short z)
      //@ requires true;
      /*@
      ensures
         result <= x && result <= y && result <= z &&
         (result == x || result == y || result == z);
      @*/
   {
      if (x < y) {
         if (x < z) {
            return x;
         } else {
            return z;
         }
      } else {
         return y;
      }
   }
}
```

VeriFast requires that the program specify a method contract for each method. The method contract consists of a *precondition* or ***requires*** *clause*, and a *postcondition* or ***ensures*** *clause*. The **requires** clause consists of the **requires** keyword, followed by an *assertion* and a semicolon. Similarly, the **ensures** clause consists of the **ensures** keyword, followed by an assertion and a semicolon. In its simplest form, an assertion is simply a boolean expression. Each clause must be inside an *annotation*, a special comment marked by an `@` sign. An annotation can be in the form of a single-line comment (*//@*) or a multiline comment (*/*@ @*/*). Both clauses must appear between the method header and the method body.

The precondition is a condition that must be true when the method is called; it is an obligation of the caller. The postcondition is a condition that must be true when the method returns; it is an obligation of the callee. In the example, the precondition is **true**, so it effectively imposes no obligation. The postcondition states that when the method returns, the return value (indicated by the special variable name `result`) must equal one of the arguments, and must be no greater than any of the arguments.

Let's check this program with VeriFast. Save this program in a file called `min.java`, and then start the VeriFast IDE using the command `vfide min.java`. The VeriFast IDE window will appear, showing `min.java` in an editor pane. To verify the program, click the **Verify** toolbar button (which looks like a **Play** button), or press the **F5** key. You will see something like this:


VeriFast (working copy build) IDE
File Edit View Verify Window Help
min.java
_assume.javaspec
java.lang.javaspec
javacard.framework.javaspec
javacard.security.javaspec
Steps
Assumptions
Heap chunks

Cannot prove condition.

```
class Program {
    static short min(short x, short y, short z)
        //@ requires true;
        /*@
        ensures
            result <= x && result <= y && result <= z &&
            (result == x || result == y || result == z);
        @*/
    {
        if (x < y) {
            if (x < z) {
                return x;
            } else {
                return z;
            }
        } else {
            return y;
        }
    }
}
```

Local	Value
result	y
x	x
y	y
z	z

Steps:

Producing assertion
Executing statement
Executing statement
Executing return statement
Consuming assertion

Assumptions:

true
(<= (+ (* -1 x) y) 0)

Notice first of all that a red bar has appeared in the toolbar, showing the message *Cannot prove condition.* The red bar indicates that verification has failed, and the message suggests the reason of the failure. If verification succeeds, this bar turns green. If verification fails, the location in the program where VeriFast encountered the failure is indicated in the editor by a double red underline. In the example, the central *and* operator of the postcondition is shown with such a double red underline. This indicates that verification of method `min` failed because, on one of the method's execution paths, VeriFast could not prove the postcondition.

VeriFast verifies a program by verifying each method in turn, in the order in which they appear in the program. If one method calls another method, the call is verified using the callee's method contract, not by recursively verifying its body at that point. This improves modularity and performance.

Each method is verified by *executing it symbolically*. Symbolic execution is like ordinary, concrete, execution, except that symbolic values are used instead of concrete values. The symbolic value of each local variable is shown in the Locals pane on the right-hand side of the VeriFast IDE window. Notice that, in the above figure, the value of local variable `x` is the symbol x, rather than some specific value such as 0 or 42. By using symbols, a single symbolic execution can cover all possible concrete executions, of which in the example there are $(2^{16})^3$ or approximately 2×10^{14}, since there is one concrete execution for each value of each 16-bit argument.

The price that we pay, however, for using symbols instead of concrete values, is that we do not necessarily know the value of the condition of an **if** statement. In that case, symbolic execution *forks*; that is, it splits into two symbolic executions: one where the condition is assumed to be true and the *then* branch of the **if** statement is executed, and one where the condition is assumed to be false and the *else* branch of the **if** statement is executed. As a result, we obtain a number of *symbolic execution paths* through the method, each characterized by a different set of assumptions.

When verification fails, the list of steps of the current symbolic execution path is shown in the VeriFast IDE in the bottom left corner. Clicking on a step highlights the corresponding program location with a yellow backhground, and shows the state of the execution at that step in the **Locals** pane, the **Assump-**

tions pane, and the **Heap chunks** pane. The Locals pane shows the current symbolic value of each local variable, and the Assumptions pane shows the list of assumptions that were made on the current path. The Heap chunks pane is explained later in this tutorial.

The path where verification of method `min` failed, consists of five steps. The first step of symbolic execution is always to *produce* the precondition. Producing a boolean expression simply means *assuming* it, i.e., adding it to the Assumptions pane. The second step corresponds to the first **`if`** statement of the method body. After this step, the condition

```
(<= (+ (* -1 x) y) 0)
```

is added to the Assumptions pane. This is merely a convoluted prefix notation for the condition $y \leq x$, i.e., we are assuming that the condition of the **`if`** statement is false, and therefore symbol y is less than or equal to symbol x. The third step corresponds to the *else* branch of the **`if`** statement, and the fourth one to the **`return`** statement. The fifth one *consumes* the postcondition; for a boolean expression this means attempting to *prove* the condition based on the assumptions in the Assumptions pane. In the example, this fails, because it is simply not true; there is a bug in the program.

Unfortunately, VeriFast does not tell us which part of the postcondition it fails to prove. We can make it do so with a trick: replace each *and* operator `&&` by a *separating conjunction* operator `&*&`. It will become clear later why this operator exists and why it is called the separating conjunction operator. When placed between boolean expressions, its meaning is the same as the *and* operator, but VeriFast processes each operand (each *conjunct*) of a separating conjunction in a separate step, so we will get a more precise error location:

```
static short min(short x, short y, short z)
   //@ requires true;
   /*@
   ensures
      result <= x &*& result <= y &*& result <= z &*&
      (result == x || result == y || result == z);
   @*/
```

Now, VeriFast highlights the `<=` operator of the `result <= z` conjunct, indicating that it cannot prove that on the failing path, the result is not greater than `z`. And indeed, our program is missing a comparison between `y` and `z`.

Exercise 1 *Correct the program and verify it. Notice the green bar.*

3 Arithmetic overflow

Let's write and specify a method that computes the absolute value of a number:

```
class Program {
   static short abs(short x)
      //@ requires true;
      //@ ensures 0 <= result &*& result == x || result == -x;
   {
      if (x < 0) {
         x = (short)-x;
         return x;
      } else {
         return x;
      }
   }
}
```

If we attempt to verify this program, VeriFast highlights the left parenthesis of the `(short)-x` cast expression, and shows a red bar with the *Potential arithmetic overflow* error message, indicating that the cast from `int` to `short` might cause an overflow. Indeed, if `x` equals −32768, the smallest `short` value, then `-x` equals 32768, which is greater than the maximum `short` value 32767.

There are a few different ways to deal with this error. Perhaps the most principled approach is to **eliminate** the error **statically**, that is, to change the specifications so that we can prove that the error never occurs. In the example, we can achieve this by stating in the precondition that `x` must be different from −32768:

```
  static short abs(short x)
    //@ requires x != -32768;
    //@ ensures 0 <= result &*& result == x || result == -x;
```

This produces a green bar. Note that this effectively propagates the problem to the caller, who will have to somehow establish the condition.

A somewhat more practical approach is to check for the condition at run time and to throw an exception:

```
import javacard.framework.*;

class Program {
  static short abs(short x)
    //@ requires true;
    //@ ensures 0 <= result &*& result == x || result == -x;
  {
    if (x == -32768)
      ISOException.throwIt(ISO7816.SW_UNKNOWN);
    if (x < 0) {
      x = (short)-x;
      return x;
    } else {
      return x;
    }
  }
}
```

This also produces a green bar.

A third option is to specify that overflow is allowed, by prefixing the overflowing operation with the `truncating` keyword, inside an annotation, as follows:

```
      x = /*@truncating@*/(short)-x;
```

However, this means the cast operation is no longer a no-op mathematically, and more specifically the postcondition no longer holds. After replacing the postcondition with `ensures true;`, we get a green bar.

The fourth option is to stick one's head in the sand and not worry about arithmetic overflow. VeriFast supports this by offering the option of disabling arithmetic overflow checking. In the VeriFast IDE, this option is found in the **Verify** menu. On the command-line, one can specify the

```
-disable_overflow_check
```

option. Selecting this option produces a green bar for our example program. Note that if this option is selected, the guarantees mentioned in the introduction of this tutorial provided by the *0 errors found* message, are weakened: they only hold for program executions where no arithmetic overflow occurs. This is often a reasonable approach for non-safety-critical applications, such as games or ordinary desktop applications. However, for Java Card applications, this is probably not an appropriate approach.

3.1 Run to Cursor; Variables versus Symbols

Consider the original program. For now, disable arithmetic overflow checking. Now, suppose that we wish to inspect the values of the local variables just before the first `return` statement. To do so, place the cursor at this `return` statement and click the **Run to cursor** toolbar button. VeriFast now shows a symbolic execution trace that reaches this location, as well as the symbolic state at this point. In the **Locals** pane, we see that the current symbolic value of program variable `x` is the term

```
(* -1 x)
```

which is a complicated prefix notation for $-x$. That is, the current value of the *program variable* `x` is the negation of the *symbol* *x*. Always carefully distinguish these two. The program variable is the name that appears in the program text, and its value can be different at different points in time on a given execution path. On the other hand, the symbol appears only in VeriFast's symbolic states (the Locals, Assumptions, and Heap chunks panes) and it represents an immutable value, which is fixed on any given execution path. In the example, symbol *x* represents the *initial value* of parameter `x`, not its current value. Note that the names on the right-hand sides in the Locals pane and all names in the Assumptions and Heap chunks panes refer to symbols, not program variables.

4 Objects; Heap Chunks

Consider the following program.

```
class Purse { short balance; }

class Program {
   Purse p1, p2;

   void deposit(Purse p, short amount) {
      p.balance += amount;
   }

   Program() {
      p1 = new Purse();
      deposit(p1, (short)100);
      p2 = new Purse();
      deposit(p2, (short)50);

      short b1 = p1.balance;
      short b2 = p2.balance;
      assert b1 == 100 && b2 == 50;
   }
}
```

Our ultimate goal is to prove that the `assert` statement succeeds.

This program may seem completely trivial. However, remember that we wish to perform *modular* verification. The proof of one method or constructor should not look inside the *body* of another method or constructor; it should look only at the other method or constructor's *contract.* Notice in the example that proving the `assert` statement requires not only that method `deposit` correctly update the balance of `Purse` object `p`, but also that it *not modify any other object.* In other words, the contract of method `deposit` should express not just that it updates `p` correctly, but also that that is its only effect. For example, the `assert` statement would not hold if we replaced method `deposit` with the following one:

```
   void deposit(Purse p, short amount) {
      p1.balance += amount;
   }
```

We will now see how we can verify this program modularly with VeriFast.

Remember that VeriFast will refuse to verify a program unless each method and each constructor has a contract. Let's first verify method `deposit`. Therefore, insert the contract

```
//@ requires true;
//@ ensures true;
```

for method `deposit` and the contract

```
//@ requires false;
//@ ensures true;
```

for the constructor. The latter contract causes VeriFast to skip this constructor for now.

If we now attempt to verify this program, VeriFast points to the assignment operator `+=` in method `deposit` with the error message

No matching heap chunks: `Purse_balance(p, _)`

The meaning of this error message is as follows. In VeriFast's verification approach, whenever a method accesses a field of an object (like `p.balance`), it must have *permission* to do so. Since there is one permission for each part of memory (or *chunk of heap*), permissions are also known as *heap chunks*. Heap chunks have a *name* and an *argument list*. A heap chunk that gives permission to access a field f of an object o of class C has name C_f and first argument o. (It also has a second argument; we will discuss the meaning of the second argument below.)

We can now interpret the error message: it states that the method is missing a permission to access field `balance` of object *p* of class `Purse`. There are three main ways in which a method or constructor can acquire such permissions: 1) a constructor receives a heap chunk for each field of the object being constructed; 2) a method or constructor can request permissions from its caller by mentioning them in its precondition; and 3) when a method or constructor calls another method or constructor, when the call returns the caller receives the permissions mentioned in the callee's postcondition.

In the case of method `deposit`, the logical approach is to request that the caller provide the necessary permission. Essentially, `deposit`'s contract must state: *If you want to call me, you need to give me permission to access the `p.balance` field.* We do this simply by mentioning the heap chunk in the precondition:

```
//@ requires Purse_balance(p, _);
//@ ensures true;
```

We use an underscore for the second argument to indicate that we do not care about its value.

Method `deposit` now verifies. To see what is going on, place the cursor on the brace that closes the body of `deposit` and click the **Run to cursor** button. We get an execution trace in the bottom left corner. Select the first step. The first step is to *produce* the precondition. If the precondition is a permission, producing it means acquiring it. More specifically, the permission is added to the **Heap chunks** pane in the bottom right corner of the VeriFast IDE window. To see this, select the second step and notice that the Heap chunks pane now lists the following heap chunk:

`Purse_balance(p, dummy)`

The second argument is a fresh symbol *dummy* which represents an arbitrary value.

At any point during execution, the Heap chunks pane shows which permissions are currently owned by the method or constructor.

Let's now verify the constructor. After we replace **`false`** by **`true`** in the precondition, VeriFast signals an error in the `Person` constructor, at the access of `p1.balance`. We get almost the same error message as before:

No matching heap chunks: `Purse_balance(object, _)` The constructor does not have permission to access `p1.balance` at this point. (Symbol *object* is the symbol that represents the value of variable `p1`.) To understand what went wrong, let's step through the execution path.

- The first step produces the precondition; in this case, the assumption `true` is added to the Assumptions pane.

- The second step indicates that we are going to verify the call of the `Object` constructor that is added implicitly at the top of the `Person` constructor. We first consume its precondition and then produce its postcondition. Since they are both `emp`, meaning *empty*, nothing happens.

- After the superclass constructor call, the heap chunks for the fields of the object being constructed (the `this` object) are added to the Heap chunks. If you select the next step, you will see that the following two chunks have appeared:

  ```
  Program_p1(this, 0)    Program_p2(this, 0)
  ```

  Note that `0` here denotes a null pointer. The second argument of a field chunk denotes the field's current value. The above chunks denote that fields `p1` and `p2` currently hold a null pointer.

- Now, the statement `p1 = new Purse();` is executed. First, the implicit `Purse` superclass constructor call is executed; then the permission for the new `Purse` object's `deposit` field is added to the Heap chunks. Notice that the new `Purse` object's identity is denoted by the fresh symbol *object*. Furthermore, the second argument of the `Program_p1` chunk is changed to *object*, to represent that field `p1` now points to the new `Purse` object.

- Now, the first `deposit` call is executed. Executing a call always means first *consuming* the precondition, and then *producing* the postcondition. Consuming the precondition means 1) if the precondition mentions a heap chunk, looking for the permission in the Heap chunks pane, and removing it, or 2) if the precondition is a boolean expression, proving it from the assumptions in the Assumptions pane. Producing the postcondition means 1) if the postcondition mentions a heap chunk, adding it to the Heap chunks pane, or 2) if the postcondition is a boolean expression, adding it to the Assumptions pane. In the case of the `deposit` call, consuming the precondition means removing the `Purse_balance(object,0)` permission from the Heap chunks pane. Producing the postcondition simply adds `true` to the Assumptions pane.

- The second and third statements are executed analogously.

- The access of `p1.balance` fails.

We can now see what has gone wrong: the required permission was taken away by the `deposit` call, and not given back. The solution is to modify the `deposit` postcondition so that it gives back the permission when it returns:

```
void deposit(Purse p, short amount)
   //@ requires Purse_balance(p, _);
   //@ ensures Purse_balance(p, _);
```

The accesses of the `balance` field in the `Person` constructor now verify.

VeriFast now complains that it cannot prove the **`assert`** statement. It cannot prove that `b1` equals 100 and `b2` equals 50. Notice in the Locals pane that the symbolic value of `b1` is *dummy* and the symbolic value of `b2` is *dummy0*. These dummy symbols are due to the underscore in the postcondition of `deposit`. The contract of `deposit` leaves the value of the field unspecified. We need to strengthen the contract so that it states that the new value of the field is the old value, plus `amount`. This means that we need to be able to mention the old value of the field in the postcondition. We can do so by *binding* the value of the field to a name in the *precondition*, by replacing the underscore in the precondition by a *variable pattern* of the form `?b`. We can then use this name in the postcondition:

```
   void deposit(Purse p, short amount)
      //@ requires Purse_balance(p, ?b);
      //@ ensures Purse_balance(p, (short)(b + amount));
```

The program now verifies. To see the effect of the new contract, place the cursor on the `assert` statement and click the **Run to Cursor** button. Notice that the two `Purse_balance` chunks in the Heap chunks pane now mention the correct values as their second argument.

Notice how VeriFast approaches the problem highlighted at the start of this section: that a method contract should mention not only what it *does* but also what it *does not do*. This is achieved in VeriFast by requiring that a method request permission for everything it does; as a result, if a method's contract does not request permission for something, then it will not do it. Specifically, if a caller has permission to modify some field, and the callee does not ask for this permission, then the caller knows that the callee will not modify this field. For example, in the `Person` constructor, since the second `deposit` call does not request permission to access `p1.balance`, it follows that it does not modify `p1.balance`.

5 Arrays; Loops

Let's verify the following program. Method `rotate` rotates the elements of array `xs` forward by one position.

```
class Program {
   static void rotate(byte[] xs, short start, short end) {
      if (start >= end - 1)
         return;
      byte last = xs[end - 1];
      for (short i = start; i < end - 1; i++) {
         xs[i + 1] = xs[i];
      }
      xs[start] = last;
   }
}
```

Specifically, let's verify that there are no out-of-bounds array accesses.[2]

As usual, we start by simply specifying

```
/*@ requires true; @*//*@ ensures true; @*/
```

Doing so results in an error message at the first array access:

No matching heap chunks: `java.lang.array_element(xs, (- end 1), _)`

As in the previous section, whenever the program access a memory location, it needs to have permission to do so. Permission to access an array element is represented by a `java.lang.array_element` heap chunk, with three arguments: the array, the index of the element, and the value of the element.

As before, method `rotate`'s contract should specify that the method's caller should provide it with array element permissions for each of the elements of the array between indices `start` and `end`. Expressing this directly is non-trivial, since we cannot simply enumerate these permissions. Fortunately, there exists a built-in type of permission that bundles all array element permissions for some given range of indices of a given array; it's called an *array slice* permission. To request the necessary permissions from the caller, we can use the following contract:

```
   static void rotate(byte[] xs, short start, short end)
      //@ requires array_slice(xs, start, end, _);
      //@ ensures array_slice(xs, start, end, _);
```

[2] The VeriFast features required to specify and verify that the method actually rotates the elements will be taught in a future lesson.

Notice two things: 1) the final argument of an `array_slice` permission represents the list of values of the elements; 2) as in the previous section, we specify the same permissions in the postcondition, so that when the method returns, the permissions are given back to the caller, so that it can continue to access the array elements.

Now, the first array access is verified successfully. VeriFast now complains at the **for** loop, saying

Loop invariant required.

Indeed, VeriFast needs a *loop invariant* in order to be able to symbolically execute a loop. A loop invariant is an assertion that describes the program state at the start of an arbitrary iteration of the loop. More specifically, it describes two things: 1) the permissions required by an arbitrary loop iteration; and 2) any information about the variables modified by the loop that is necessary to verify an arbitrary iteration. Thanks to a loop invariant, VeriFast can symbolically execute a loop by symbolically executing the loop body only once, starting from an arbitrary state that satisfies the loop invariant. More specifically, VeriFast verifies a loop as follows:

- It *consumes* the loop invariant. That is, it checks the conditions and removes the permissions specified by the loop invariant. This checks that the start of the first iteration satisfies the loop invariant.

- The heap chunks that remain after consumption of the loop invariant are called the *loop frame*. These are now temporarily removed from the Heap chunks pane, so that it is now empty.

- Each local variable that is modified by the loop is now given an arbitrary value, in the form of a fresh symbol.

- The loop invariant is *produced*. That is, it assumes the conditions and adds the permissions described by the loop invariant. This step and the previous one together cause the current state to represent the start of an arbitrary iteration of the loop.

- A case split is performed on the loop condition. That is, symbolic execution forks into two branches.

- On the first branch, the loop condition is assumed to be false. The loop frame is added back to the Heap chunks pane, and symbolic execution continues after the loop.

- On the second branch, the loop condition is assumed to be true. The loop body is executed. After the loop body finishes, the loop invariant is again consumed. This checks that if the start of some iteration satisfies the loop invariant, then so does the end of that iteration (which is the start of the next iteration).

To verify the loop in the example, let's start with the simplest possible loop invariant:

```
for (short i = start; i < end - 1; i++)
  //@ invariant true;
```

Unsurprisingly, this causes VeriFast to complain at the first array access (temporally speaking) in the loop body. Since we did not specify any permissions in the loop invariant, the loop body is verified with an empty Heap chunks pane, so we do not have permission to access this array element. Since the loop eventually accesses all elements of our array slice, we need to specify permissions for all of these elements in the loop invariant. The obvious invariant would seem to be the following:

```
for (short i = start; i < end - 1; i++)
  //@ invariant array_slice(xs, start, end, _);
```

Unfortunately, VeriFast now still complains at the access of the element at index `i`, even though the array slice permission is now in the Heap chunks pane. The problem is that there is nothing in the symbolic state that says that the value of variable `i` lies between `start` and `end`. Remember that all variables modified by the loop (specifically, `i`, in the example) are given an arbitrary value, so that the symbolic state represents the start of an arbitrary loop iteration. However, in the example, not all possible values of `i` will actually occur in any loop iteration. Specifically, `i` will always lie between `start` and `end`. To make VeriFast aware of this, we need to add some information to the loop invariant:

```
for (short i = start; i < end - 1; i++)
   //@ invariant array_slice(xs, start, end, _) &*& start <= i;
```

The program now verifies successfully. Notice that we need to add only the lower bound. The fact that `i` is less than `end` follows from the fact that if we reach the loop body, the loop condition must be true, which states that `i` is less than `end - 1`.

6 Abstraction: Predicates

Let's verify that the following program doesn't crash:

```
class ArrayList {
   byte[] elems;
   short count;

   ArrayList(short size) {
      elems = new byte[size];
   }

   short getCount() {
      return count;
   }

   byte get(short index) {
      return elems[index];
   }

   boolean add(byte value) {
      if (count == elems.length)
         return false;
      elems[count++] = value;
      return true;
   }
}

class Program {
   static void test() {
      ArrayList list = new ArrayList((short)10);
      if (list.add((byte)1) && list.add((byte)2) && list.add((byte)3)) {
         short count = list.getCount();
         assert count == 3;
         list.get((short)2);
      }
   }
}
```

Let's start out by giving each constructor and method the following contract:

```
//@ requires false;
//@ ensures true;
```

That way, we can choose which method we work on first. Let's start with method `add`. Notice that this method accesses the `elems` and `count` fields, as well as the array pointed to by field `elems`. Furthermore, it relies on the property that `count` is nonnegative and less than or equal to `elems.length`; otherwise, the array access could be out of bounds. From this analysis, it seems clear what the precondition of method `add` should be:

```
/*@
requires
    ArrayList_elems(this, ?es) &*& ArrayList_count(this, ?n) &*&
    array_slice(es, 0, n, _) &*& array_slice(es, n, es.length, _) &*&
    es.length <= 32767;
@*/
//@ ensures true;
```

And indeed, with this contract method `add` verifies successfully. Let's read the precondition. The first item in the precondition specifies the permission to access the `elems` field of `this`, and binds the value of the field to name `es`. Similarly, the second item binds the value of the `count` field to name `n`. The third and fourth items specify the permission to access the array elements of array `es` from index zero to `n`, and `n` to `es.length`, respectively. The last item records that the length of the array is a **`short`** value; this is necessary to prove absence of arithmetic overflow.

Notice that the second line of the precondition is equivalent to:

```
array_slice(es, 0, es.length, _) &*& 0 <= n &*& n <= es.length &*&
```

The choice between the two notations is a matter of style.

Furthermore, we can write the first line in a slightly more readable way:

```
elems |-> ?es &*& count |-> ?n &*&
```

Here, `elems ->` ?es| is an abbreviation for `this.elems ->` ?es|, which in turn is a nicer notation for `ArrayList_elems(this, ?es)`. The intention is that this assertion is read as: `elems` *points to* `?es`.

While this precondition works, it is not very satisfactory. The problem with it is that it exposes the internal complexities of the `ArrayList` class to its callers. Ideally, we would be able to hide this complexity. Fortunately, we can do so, using VeriFast's support for *predicates*. A *predicate* is essentially a named assertion. We can obtain a nicely abstract precondition as follows:

```
/*@
predicate ArrayList() =
    elems |-> ?es &*& count |-> ?n &*&
    array_slice(es, 0, n, _) &*& array_slice(es, n, es.length, _) &*&
    es.length <= 32767;
@*/

boolean add(byte value)
    //@ requires ArrayList();
    //@ ensures true;
```

What is happening here is that we define a predicate named `ArrayList`, whose definition is exactly the precondition that we had before. In the precondition itself, we now simply refer to this predicate. (Note that the notation `ArrayList()` in the precondition is an abbreviation of `this.ArrayList()`; a predicate defined inside a class must always be implicitly or explicitly applied to an instance of the class, just like a method.)

Now that we have a nice precondition for method `add`, let's jump to method `get`. Let's try the same precondition:

```
byte get(short index)
    //@ requires ArrayList();
    //@ ensures true;
```

Unfortunately, this precondition is not sufficient. We get an error at the array access in method get's body. The problem is that we do not know that index is inside the bounds of the array. Actually, we only want users to call this method with indices that are less than the number of values stored in the array, i.e., the index should be less than the value of field count. To specify this, we need to expose this value to the user, by exposing it as a *parameter* of the ArrayList predicate. We modify the definition of predicate ArrayList as follows:

```
/*@
predicate ArrayList(short n) =
   elems |-> ?es &*& count |-> n &*&
   array_slice(es, 0, n, _) &*& array_slice(es, n, es.length, _) &*&
   es.length <= 32767;
@*/
```

Notice that we added a parameter n and we use it instead of the variable pattern ?n in the count points-to assertion.

We now need to update the contracts of methods get and add. In fact, let's go ahead and also fill in the postcondition of each method:

```
byte get(short index)
   //@ requires ArrayList(?n) &*& 0 <= index &*& index < n;
   //@ ensures ArrayList(n);

boolean add(byte value)
   //@ requires ArrayList(?n);
   //@ ensures result ? ArrayList((short)(n + 1)) : ArrayList(n);
```

Both methods now verify successfully. Notice that we use a *conditional assertion*, analogous to Java's conditional expressions, to specify that depending on the return value of add, the number of elements in the ArrayList has either increased by one or stayed the same.

We are now ready to fill out the specifications of the remaining constructor and methods:

```
ArrayList(short size)
   //@ requires 0 <= size;
   //@ ensures ArrayList(0);

short getCount()
   //@ requires ArrayList(?n);
   //@ ensures ArrayList(n) &*& result == n;
```

These specifications are sufficient to successfully verify method test.

7 Applets; Transactions; Fractions

Consider the following Java Card applet.

```
import javacard.framework.*;

public final class MyApplet extends Applet {
   int tokensLeft, tokensUsed;

   MyApplet() {
      tokensLeft = 10;
   }

   public static void install(byte[] array, short offset, byte length) {
        MyApplet applet = new MyApplet();
        applet.register();
    }

    public void process(APDU apdu) {
        //@ { int x = tokensLeft + tokensUsed; assert x == 10; }
        if (tokensLeft == 0)
            ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
        tokensLeft--;
        tokensUsed++;
    }
}
```

The annotation at the start of the body of method `process` asserts that whenever applet execution reaches that point, the sum of `tokensLeft` and `tokensUsed` equals 10.[3] Our goal is to verify that this assertion holds in all applet executions.

The applet is somewhat contrived, but it illustrates an important concept that appears in all applets: the need for transactions.

As usual, we need to start by providing a contract for each constructor and method. Let's first give each constructor and method the contract

```
    //@ requires true;
    //@ ensures true;
```

We get an error at the call of method **register** in the `install` method. Notice that when showing a call step, the VeriFast window splits into two halves; the top half displays the callee, and the bottom half displays the call site. The error occurred while consuming method **register**'s precondition, which reads as follows:

```
    public void register();
        //@ requires this.valid() &*& system();
        //@ ensures true;
```

This specification is located, together with the rest of the specification of class `Applet` and other classes from the Java Card API, in file `javacard.framework.javaspec` in the `bin/rt` subdirectory of the VeriFast distribution. The precondition of method **register** specifies two heap chunks: a `valid` chunk for the applet, and a `system` chunk. The `valid` chunk refers to the `valid` predicate defined in class `Applet`. It is intended to denote the permissions and information needed by the various applet methods, such as `select`, `deselect`, and `process`. In the example, the `process` method needs permission to access fields `tokensLeft` and `tokensUsed`; therefore, we *override* predicate `valid` in our example class `MyApplet` so that it specifies these permissions:

```
public final class MyApplet extends Applet {
    int tokensLeft, tokensUsed;

    //@ predicate valid() = tokensLeft |-> _ &*& 0 <= m &*& tokensUsed |-> _;
```

Furthermore, we strengthen the postcondition of the `MyApplet` constructor so that it returns these permissions, in the form of a `valid` chunk:

```
    MyApplet()
        //@ requires true;
        //@ ensures valid();
```

Now, VeriFast no longer complains at the first item in the precondition of method **register**; it now complains at the second item, the `system()` chunk. This chunk represents some system resources, which

[3] The annotation uses a temporary variable x because the assertion in an assert statement cannot contain field dereferences.

we discuss in a later section. These resources are made available to the applet by the Java Card runtime environment when it calls the install method; therefore, we can specify this chunk in the precondition of method install:

```
public static void install(byte[] array, short offset, byte length)
   //@ requires system();
   //@ ensures true;
```

Method install now verifies.

VeriFast now complains at the access of field tokensLeft in the annotation in method process. The problem is that the method does not have permission to access this field. Interestingly, if we look at the precondition of method process in class Applet, we see that the item [1/2]this.valid() is one of the items in that precondition. A chunk that is preceded by a fractional number enclosed in square brackets, is called a *fractional permission.* The number itself is called the *coefficient.* If a chunk is not preceded by any number, it is called a *full permission*; it implicitly has coefficient 1.

The difference between a fractional permission and a full permission depends on the kind of permission. For a field, a fractional permission grants only read access, not write access; a full permission grants both read and write access. For a predicate, such as valid, its coefficient is simply distributed over the items in its definition. That is, in the example, [1/2]valid grants read-only access to field tokensLeft and field tokensUsed.

The reason why VeriFast's Java Card API specification specifies only a fractional valid permission in method process's precondition, is to enforce the correct use of Java Card *transactions.* Specifically, applets are prevented from modifying their state outside of a transaction.[4]

We copy the [1/2]valid() item into the precondition of our process method, so that VeriFast allows the field accesses inside the annotation:

```
public void process(APDU apdu)
   //@ requires [1/2]valid();
   //@ ensures true;
```

The next error we get points to the postcondition of method process in the specification of class Applet. What is happening is that VeriFast found an error while checking the conformance of the contract of method process in class MyApplet with the contract of the same method in class Applet. To check conformance of an overriding method with an overridden method, VeriFast first produces the precondition of the overridden method, then consumes the precondition of the overriding method, then produces the postcondition of the overriding method, and finally consumes the postcondition of the overridden method. In words, the specification of class Applet states that method process must return the [1/2]valid() permission to the caller when the method is finished. Our process method is in violation of this specification, since it does not return this permission; indeed, it does not mention it in its postcondition. This is easily fixed:

```
public void process(APDU apdu)
   //@ requires [1/2]valid();
   //@ ensures [1/2]valid();
```

The conformance check now succeeds, and so do the field accesses inside the annotation. However, now the assert statement fails. VeriFast cannot prove that the sum of fields tokensLeft and tokensUsed equals 10. The solution is to state this invariant in our valid predicate:

```
//@ predicate valid() = tokensLeft |-> ?m &*& 0 <= m &*& tokensUsed |-> ?n &*& m + n == 10;
```

The next error we get is at the decrement of field tokensLeft. The error message is: *Writing to a field requires full permission.* VeriFast here points out an actual bug in our program: our program does not always respect our assertion that the sum of tokensLeft and tokensUsed equals 10. Specifically, consider an

[4] In reality, modifications consisting of a single field or array element assignment do not need to be protected by a transaction. However, VeriFast does not yet support this scenario. Currently, all modifications, even single assignments, need to occur inside a transaction.

execution where a card tear occurs between the decrement of tokensLeft and the increment of tokensUsed. The resulting state would violate our assertion. We fix the program by inserting calls to the Java Card transaction system into our code:

```
JCSystem.beginTransaction();
tokensLeft--;
tokensUsed++;
JCSystem.commitTransaction();
```

We now get an error at the call of method beginTransaction. Its precondition requires a current_applet0 chunk. This chunk is necessary to indicate to beginTransaction which applet's state is being transacted on. Notice that a current_applet chunk appears in the precondition for method process in class Applet. This chunk bundles a current_applet0 chunk and a system chunk. To fix the problem, we mention the chunk in our own precondition. We also return the chunk to the caller when we are finished:

```
public void process(APDU apdu)
    //@ requires current_applet(this) &*& [1/2]valid();
    //@ ensures current_applet(this) &*& [1/2]valid();
```

Strangely, we now still get the *Writing to a field requires full permission* error at the decrement operation. If we look in the Heap chunks pane, we can see the cause of the problem. Method beginTransaction has provided us with another [1/2]valid() chunk, on top of the [1/2]valid() chunk that we received from the caller at the start of the method. This should yield a full valid() chunk, so we should be able to modify the fields. However, VeriFast did not figure out that it needs to unfold the definition of valid here and replace the [1/2]valid() chunk with its definition. To force VeriFast to do so, we insert an **open** ghost command after the call of beginTransaction:

```
//@ open valid();
```

The program now verifies.