Java Program Verification at Nijmegen: Developments and Perspective

Bart Jacobs and Erik Poll

University of Nijmegen,
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands.
{bart,erikpoll}@cs.kun.nl

Abstract. This paper presents a historical overview of the work on Java program verification at the University of Nijmegen (the Netherlands) over the past six years (1997–2003). It describes the development and use of the LOOP tool that is central in this work. Also, it gives a perspective on the field.

1 Introduction

The LOOP project started out as an exploration of the semantics of object-oriented languages in general, and Java in particular. It has evolved to become what we believe is one of the largest attempts to date at formalising a real programming and using this formalisation as a basis for program verification. It is probably also one of the largest attempts to date at using mechanical theorem provers. This paper attempts to give an overview of the whole project. It is unavoidable that we have to resort to a high level of abstraction to do this in the limited space here. Therefore, our main aim is to convey the general principles and we will frequently refer to other papers for much more of the technical details.

From the outset, a goal of the project has been to reason about a *real* programming language, and not just a toy object-oriented language. Apart from leaving out threads, all the complications of real Java are covered, incl.

- side-effects in expressions (something often omitted in the toy languages studied in theoretical computer science),
- exceptions and all other forms of abrupt control flow (including the more baroque constructs that Java offers, such as labelled breaks and continues),
- static and non-static field and methods,
- overloading,
- all the complications of Java's inheritance mechanism, including late binding for methods, early binding for fields, overriding of methods, and shadowing (or hiding) of fields.

Apart from threads, the only major feature of Java not supported is inner classes.

The LOOP tool What we call the LOOP tool is effectively a compiler, written in O‘Caml. Fig. 1 illustrates roughly how it is used. As input, the LOOP tool takes sequential Java programs, and specifications written (as annotations in the Java source files) in the Java Modeling Language JML [24]. Fig. 2 gives an example of a Java class with a JML specification. As output, the LOOP tool generates several files which, in the syntax of the theorem prover PVS [31], describe the meaning of the Java program and its JML specification. These files can be loaded into PVS, and then one can try to prove that the Java program meets its JML specification.

In addition to the automatically generated PVS files, there are also several hand-written PVS files, the so-called prelude. These files define the basic building blocks for the Java and JML semantics, and define all the machinery needed, in the form of PVS theories and lemmas, to support the actual work of program verification.

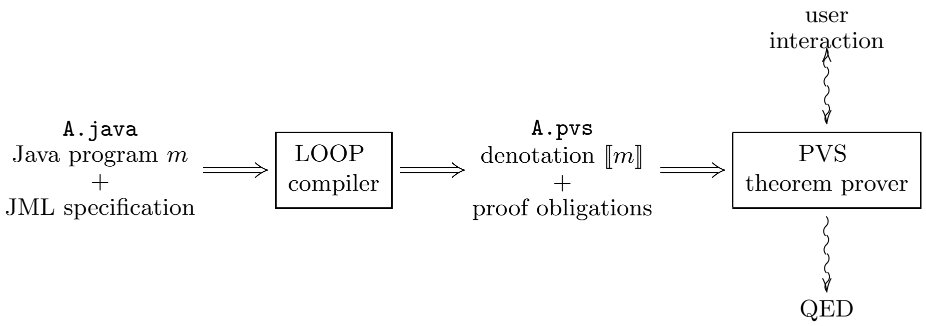

Fig. 1. The LOOP tool as pre-processor for PVS

Organisation of this paper The next section begins by giving an example of a Java program with JML specification to illustrate the kind of program verification we are doing. The organisation of the rest of this paper is then more or less chronological, and follows the bottom-up approach that we have taken over the years, starting at the detailed representation of the Java semantics at a low level, on top of which further abstractions are built. The LOOP project started with definition of a formal semantics for Java in higher-order logic, by giving a so-called shallow embedding, and using coalgebras as a means of organising the semantics of objects. This is described in Sect. 3. The project then evolved to also provide a formal semantics of the Java specification language JML in PVS, as discussed in Sect. 4, and to provide techniques for the verification Java programs with JML specifications on the basis of these formal semantics, which we discuss in Sect. 5. Sect. 6 compares the LOOP projects with other work on providing theorem prover supported program verification for Java.

2 Specification and verification example

Fig. 2 gives an example of a simple Java method arrayCopy preceded by a specification written in JML. It illustrates some of the complications that may arise in actual specification and verification. Although the idea of copying part of one array into another is quite simple, an accurate specification turns out to be surprisingly subtle. The specification makes many possibilities explicit, and serves as a precise documentation that can help a programmer who wants to use this arrayCopy method.

```
class ArrayCopy {

   /*@    requires src != null && dest != null &&
     @             destOff+length >= 0; // no overflow
     @
     @  assignable dest[destOff..destOff+length-1];
     @
     @     ensures (length > 0 ==>
     @          (srcOff >= 0  && srcOff+length <= src.length &&
     @              srcOff+length >= 0 &&
     @           destOff >= 0 && destOff+length <= dest.length))
     @          &&
     @          (\forall int i; 0 <= i && i < length ==>
     @               dest[destOff+i] == \old(\old(src)[\old(srcOff)+i]));
     @
     @     signals (ArrayIndexOutOfBoundsException)
     @          length > 0 &&
     @          (srcOff < 0  || srcOff+length > src.length ||
     @              srcOff+length < 0 // caused by overflow
     @           || destOff < 0 || destOff+length > dest.length);
     @*/
    public static void arrayCopy(byte[] src,   int srcOff,
                                 byte[] dest,  int destOff,
                                 int    length)
    {   if (length <= 0) return;
        if (srcOff > destOff) {
            for (int i = 0; i < length; i++)
                dest[destOff+i] = src[srcOff+i];
        }
        else {
            for (int i = length-1; i >= 0 ; i--)
                dest[destOff+i] = src[srcOff+i]; }
    }
}
```

Fig. 2. Example JML specification, proven correct

This JML specification consists of

- a precondition, the `requires` clause;
- a so-called frame property, the `assignable` clause, that restricts the possible side effects of the method;
- a postcondition, the `ensures` clause;
- an "exceptional" postcondition, the `signals` clause.

The meaning of the specification is that if the precondition is met, then either the method terminates normally making the postcondition true, or the method terminates abnormally by throwing an `ArrayIndexOutOfBoundsException` making the associated exceptional postcondition true, and the method will not change any fields other than those listed the `assignable` clause.

The use of `\old` in the postcondition allows us to refer to the value of an expression in the pre-state of the method invocation. We need to use `\old` here in the postcondition to allow for the possibility that the two arrays `src` and `dest` are aliases. This possibility is also reason for the case distinction in the implementation. Omitting `\old` in the postcondition is an easy mistake to make, as is not making the case distinction in the implementation. Either mistake, or both at the same time, would make it impossible to verify correctness of the implementation wrt. the specification.

A subtle point that has to be taken into account is overflow of the additions that are used. If `destOff+length` yields an overflow, the resulting value becomes negative. This is excluded in the precondition (and is needed for the `assignable` clause to make sense). However, the addition `srcOff+length` may also cause an overflow. This possibility is not excluded in the precondition, but handled explicitly in the (normal and exceptional) postconditions.

Fig. 2 gives only one possible spec for `arrayCopy`, but many variations are possible. Eg., we could strengthen the precondition to exclude the possibility of an `ArrayIndexOutOfBoundsException`, or we could weaken the precondition to include the possibility of a `NullPointerException`.

We have used the LOOP tool and PVS to prove that the Java code in Fig. 2 satisfies the JML specification given there. This verification can be done both with Hoare logic, and with weakest precondition reasoning, see Section 4. In both cases it requires the specification of the loop invariants for the two for-loops, which we have omitted from Fig. 2 to save space.

Similar verification challenges may be found in [19].

3 Semantical phase

This first phase of the project concentrated on the semantical set-up for the sequential part of Java. This grew out of earlier work on so-called coalgebraic specification. Important issues at this stage were the underlying memory model [1] and the semantics of inheritance [14]. This semantics is fairly stable since about 2000, and has undergone only relatively minor, modular changes such as the move from unbounded to bounded semantics for integral types [17], see Sect. 3.5.

We shall briefly review the main points. A more detailed description of the material presented in this section can be found in Chapter 2 of [13].

3.1 Coalgebraic origins

During the mid-nineties Horst Reichel [33] was the first who clearly described the view that the semantics of objects and classes can be phrased in terms of coalgebras, soon followed by others [16, 25]. Coalgebras are the formal duals of algebras. Algebras describe data, but coalgebras describe dynamical systems. Coalgebras consist of a set S together a function acting on S. The elements of S are usually called states, and therefore S is often referred to as the state space. The function acting on S is typically of the form $S \to \boxed{\;\cdots\;}$ and gives more information about states. For instance, it may involve a map of the form $S \to \texttt{int}$ which represents an integer-valued field. In each state it tells the value of the field. But also the function may involve a transition function $S \to S$, or $S \to \{\perp\} \cup S$, or $S \to \mathcal{P}(S)$ describing ways to move to successor states. Hence the result type $\boxed{\;\cdots\;}$ tells us what kind of functions we have on our state space S[1].

A class in an object-oriented programming language (like Java) combines data (in its fields) with associated operations (in its methods). The key point is that such a class may be considered as a coalgebra, and an object of the class as a state of this coalgebra (*i.e.* as an element of the state space). Given such an object/state `o`, field access `o.i` yields the value of the field function `i` in the state (referred to by) `o`. Similarly, a method invocation `o.m` yields a successor state resulting from the application of the function `m` to the state `o`.

Suppose a method in Java is declared as `boolean m(int j){...}`. Such a method usually terminates normally, producing an integer result value, and implicitly, a new state. But also it may hang, if it gets into an infinite loop or recursion. There is a third option, since it may throw an exception of some sort, caused for instance by a division by zero. Semantically, we shall interpret such a method `m` as a function $[\![\texttt{m}]\!]$ of the coalgebraic form:

$$S \xrightarrow{[\![\texttt{m}]\!]} \Big(\{\perp\} + (S \times \texttt{boolean}) + (S \times \texttt{Exception})\Big)^{\texttt{int}} \qquad (1)$$

where $+$ describes disjoint union. This says that `m` takes a state/object $x \in S$ and an integer $j \in \texttt{int}$ and produces a result $[\![\texttt{m}]\!]\,(x)(j)$ which is either:

- $\perp$, in case $[\![\texttt{m}]\!]\,(x)(j)$ hangs;
- (s', b) in case $[\![\texttt{m}]\!]\,(x)(j)$ terminates normally with successor state x' and boolean result value b;
- (x', e) in case $[\![\texttt{m}]\!]\,(x)(j)$ terminates abruptly because of exception e, with successor state x'.

[1] Formally, in categorical terminology, this box is an endofunctor acting on a suitable category of state spaces, see [21].

The modeling of a `void` method is similar. The normal termination case (the second one) is then without result value, and the abrupt termination case (the third one) involves besides exceptions also a possible return, break or continue.

The semantics of a Java method is defined by induction over the structure of the methods body. To do this we define the semantics of all Java statements and expression, by induction over their structure, which amounts to defining the semantics of all of Java's language constructs for statements and expressions. For example, $[\![s_1; s_2]\!]$ is defined as $[\![s_1]\!] \, ; [\![s_2]\!]$, where ; is the function that maps functions s_1 and s_2 of type $S \to (\{\bot\} + S + (S \times \texttt{Exception}))$ to the function

$$\lambda x : S. \begin{cases} \bot, & \text{if } s_1(x) = \mathit{in}_1(\bot) & \text{ie. if } s_1(x) \text{ hangs,} \\ s_2(x') & \text{if } s_1(x) = \mathit{in}_2(x') & \text{if } s_1(x) \text{ terminates normally} \\ & & \text{in state } x', \\ \mathit{in}_3(x', e), & \text{if } s_1(x) = \mathit{in}_3(x', e), & \text{ie. if } s_1(x) \text{ terminates abruptly by} \\ & & \text{throwing exception } e \text{ in state } x' \end{cases}$$

of the same type.

The main point is that a class with fields $f_1 \colon A_1, \ldots, f_n \colon A_n$, and methods $m_1, \ldots, m_k$ is interpreted as a single coalgebra of the form:

$$S \xrightarrow{\quad c \quad} A_1 \times \cdots A_n \times M_1(S) \times \cdots M_k(S) \tag{2}$$

where M_j is the result type corresponding to method m_j, like in (1). Hence c combines data and operations in a single function. The individual fields and methods can be reconstructed from c via appropriate projections[2].

In work on coalgebraic specification [16, 12, 11, 34, 35] coalgebras are studied together with certain assertions that constrain the possible behaviour. A format has been defined, in a language called CCSL for Coalgebraic Class Specification Language, in which such specifications could be written. A special compiler translates these specifications to the language of the theorem prover PVS [31]. The compiler is called LOOP, for Logic of Object-Oriented Programs. Coalgebraic notions like invariance, bisimilarity and refinements can then be used to reason about models of class specifications.

Soon after the start of this work on coalgebraic class specifications it was realised that the assertions could also be used to bind method names to method bodies (implementations) and fields to particular access functions. This led to an extension of the LOOP compiler that works on Java programs and also produces output for PVS.

For some time the LOOP compiler was developed jointly for CCSL and for Java. Internally, a representation of an abstract formulation of higher-order logic is used. A pretty printer is available for PVS, and also one for Isabelle/HOL [29]. For various practical and organisational reasons the two translations of the tools (for CCSL and for Java) were split. The version for CCSL is now publicly available[3]. It is maintained in Dresden and still supports output for both PVS and

[2] For reasons of simplicity, we have omitted constructors. But they can be understood as special methods.

[3] At `wwwtcs.inf.tu-dresden.de/~tews/ccsl/`.

Isabelle, see [35]. The version of the LOOP tool for Java is maintained in Nijmegen. A comparison between its performance in PVS and Isabelle is given in [10, 13]. But currently, the translation to Isabelle is no longer supported.

When we talk about the LOOP tool in the remainder of this paper we refer to the one that translates Java (and JML) to PVS. It is described in [2].

The theory of coalgebras has gone through rapid development during the last few years. However, coalgebras are only used as convenient representational device for classes in the LOOP translation from Java to PVS. The associated theory is not really used in any depth, and is not needed to understand the translation.

3.2 Memory model

The discussion above mentions a state space S that methods act on, without going into detail of what this state space consists of. The coalgebraic representation of classes (2) takes a functional view in which each class has its own state space and in which there is no object identity. Clearly, this does not work at all for an imperative language such as Java. Here the state space that programs act on is the entire memory of the computer, i.e. the heap and the stack. The heap records all the objects in existence, their states, ie. values of all fields, and their run-time types. The representation of all this in PVS is what we call the memory model or object memory. For a more detailed description of the memory model we refer to [1]. Here we just explain the main idea.

The memory model is represented in PVS as a complicated type called OM, for Object Memory. It consists of three infinite series of memory cells, one for the heap, one for the stack, and one for static data. Each cell can store the data of an arbitrary object (including its run-time type, represented as a string). An object is represented as either the null-reference, or a reference to a particular cell on the heap. This reference consists simply of a natural number n, pointing to the n-th cell. Associated with the memory are various put and get operations. The LOOP compiler binds these to the variables occurring in the programs that are translated. For instance, for integer fields `i,j`, the translation of an assignment `i=5` involves the put operation associated with `i`. It is a functional operation which maps the memory $x\colon \mathsf{OM}$ before the assignment to an entirely new memory $x' = \mathsf{put}(x, \text{“position of } \mathtt{i}\text{”}, 5)\colon \mathsf{OM}$ after the assignment. The value of `j` is obtained via its get operation. There are obvious put-get rules ensuring that the value of `j` in x' is the same as in x. During verifications these rules are loaded in PVS as so-called auto-rewrite rules and applied automatically. Hence the reader does not have to worry about these low-level memory issues, including references and aliasing.

One difficulty in defining a suitable type OM in PVS is the constraints imposed by the PVS type system. Intuitively, the state of an individual object can be seen as a record value—eg, the state of an object with a field `x` of type `int` and a field `b` of type `boolean` would be a record value of a record type {`x:int, b:boolean`}—and the heap can be regarded as a list of such record values. However, dealing with Java's hiding (or shadowing) of superclass fields will require some additional machinery in such an approach. Also, such a representation is not that convenient in PVS: it would require a heterogeneous list of record values, since objects on the heap have different types, and PVS doesn't support such heterogeneous lists. A way around this would be to have separate heap for each class, but then interpreting subtyping causes complications.

Despite the absence of the horrible pointer arithmetic à la C(++), references remain the main complication in reasoning about Java programs, as the basic "pointer spaghetti" difficulties associated with references, like aliasing and leakage of references, remain. One would really like Java to offer some notion of encapsulation or confinement and alias control, such as the universes [28], which could then be reflected in the memory model.

3.3 Inheritance

The way that inheritance is handled by the LOOP tool is fairly complicated—because inheritance itself is fairly complicated. Here we shall only describe the main ideas, and we refer to [14] for the details.

When Java class `B` inherits from class `A`, all the methods and field from `A` are in principle accessible in `B`. If the current object `this` belongs to class `B`, then `(super)this` belongs to `A`, and allows access to the fields and methods of `A`. Class `B` may thus add extra fields and methods to those of its superclass `A`. This is unproblematic. But `B` may also re-introduce methods and fields that already occur in `A`. In that case one speaks of *hiding* of fields and of *overriding* of methods. The terminology differs for fields and methods because the underlying mechanisms differ. Field selection `o.i` is based on the compile-time type of the the receiving object `o`, whereas method invocation `o.m()` uses the run-time type of `o`.

The basis for our handling of inheritance can be explained in terms of the coalgebraic representation. Suppose the class `A` is represented as a coalgebra $S \to \widehat{A}(S)$, like in (2). The representation of class `B` is then of the form $S \to [\cdots] \times \widehat{A}(S)$, where the part $[\cdots]$ corresponds to the fields and methods of `B`. In this the operations from `A` are accessible in `B`: the `super` operation involves a projection.

But there is also a second (semantical) mapping possible from `B` to `A`, namely the one which replaces in $\widehat{A}$ those methods that are overridden in `B`. These two different mappings allow us to model the difference between hiding and overriding.

The LOOP tool inserts the appropriate mapping from subclasses to superclasses. As a result, the selection of appropriate get and put operations for fields, and of the appropriate method body for methods is not a concern for the user. It is handled automatically.

3.4 Executability of the semantics

Although our Java semantics is denotational rather than operational, it is still executable in PVS to a degree, in the sense that PVS can try to rewrite $[\![\mathtt{s}]\!]$ to some normal form using a given set of equalities. PVS will not always produce a readable result, or, indeed, any result at all, as the attempted evaluation might not terminate (notably, if the program diverges), but in many cases PVS can symbolically execute programs in this way.

In fact, one could use the LOOP tool as a normal Java compiler, which produces binaries that can be executed inside PVS instead of class files which can be executed on a virtual machine. Of course, this is not very practical, because such executions are extremely slow and use huge amounts of memory, and because we do not have an implementation of the entire Java API in PVS.

This possibility of symbolic execution has been extremely useful in testing and debugging our formal semantics. By comparing the results of the normal execution of a Java program, i.e. the result of executing its bytecode on a Java VM, and the symbolically execution of its semantics in PVS, we can check if there are no mistakes in our semantics.

It is somewhat ironic that we have to rely on the down-to-earth method of testing to ensure the correctness of our formal semantics, when this formal semantics is used as the basis for the more advanced method of program verification. But given there is nothing that we can formally verify our semantics against—this would presuppose another formal semantics—such testing is the only way to to ensure the absence of mistakes in our semantics, apart from careful, but informal, verification against the official Java Language Specification [9].

Symbolic execution is also extremely useful in the verification of Java programs as discussed later in Sect. 5: for relatively simple fragments of code, and relatively simple specifications, PVS can often fully automatically decide correctness by symbolic execution of the code.

3.5 Java arithmetic

Initially, the Java semantics simply interpreted all of Java's numeric types—`byte` (8 bits), `short` (16 bits), `int` (32 bits) and `long` (64 bits)—as PVS integers. This was just done to keep things simple; our main interest was the semantics of Java features such as object-orientation, inheritance, exceptions, etc., and interpretation of the base types is orthogonal to the semantics of these. Later, when this became relevant for the Java Card smart card programs we wanted to verify, a correct formalisation of the semantics of Java numeric types, with all the peculiarities of the potential overflow during arithmetic operations, was included [17]. It is used in the verification example in Section 2.

4 Specification phase

The semantics of sequential Java described in the previous section was developed with the aim to do program verification. This requires specification of properties that we want to verify. One option is to specify such properties directly in PVS, i.e. at the semantical level. This approach was used initially, e.g. in [22]. However, this approach quickly becomes impractical, as specifications become very complicated and hard to read.

This is why we decided to adopt JML as our specification language, and extended the LOOP tool to provide not just a formal semantics of Java, but also a formal semantics of JML. The fact that JML is a relatively small extension of Java has the pleasant consequence that much of our Java semantics could be re-used—or extended—to provide a semantics for JML.

The LOOP tool provides the semantics of a JML specification for an individual method as a proof obligation in PVS. Taking some liberties with the syntax, and ignoring the exceptional postconditions expressed by signals clauses, the proof obligation corresponding with a JML specification of the form

```
/*@ requires    Pre;
  @ assignable Assign;
  @ ensures    Post;
  @*/
  public void m()  ...
```

is of the following form

$$\begin{aligned} \forall x : \mathsf{OM}.\, \llbracket Pre \rrbracket\,(x) \Rightarrow \llbracket Post \rrbracket\,(x, \llbracket m \rrbracket\,(x)) \\ \wedge\, \forall l \notin \llbracket Assign \rrbracket\,.\; x.l = \llbracket m \rrbracket\,(x).l \end{aligned} \tag{3}$$

Here $\llbracket m \rrbracket$ is the semantics of the method involved, $\llbracket Pre \rrbracket$ the semantics of its precondition, a predicate on OM, $\llbracket Post \rrbracket$ the semantics of its postcondition, a relation on OM, and $\llbracket Assign \rrbracket$ the meaning of the assignable clause, a subset of OM. Just like the LOOP tool generates PVS file defining $\llbracket m \rrbracket$, it generates PVS files defining $\llbracket Pre \rrbracket$, $\llbracket Post \rrbracket$ and $\llbracket Assign \rrbracket$.

To produce such proof obligations, the LOOP tool translates any `assignable` clause to a set $\llbracket A \rrbracket$ of locations in the object memory OM, and translates any pre- and postconditions to a PVS predicates or relations on the object memory OM.

JML's syntax for preconditions, postconditions, and invariants extends Java's boolean expressions, for example with a logical operator for implication, `==>`, as well as universal and existential quantification, `\forall` and `\exists`. So, to provide a formal semantics for JML, the LOOP semantics for Java had to be extended to support these additional operations.

One complicating factor here is that Java boolean expression may not terminate, or throw an exception, but we want our interpretations of pre- and postconditions to be proper two-valued PVS booleans. This is a drawback of using Java syntax in a specification language, as is discussed in [23]. We deal with this by interpreting any Java boolean expression that does not denote `true` or `false` by an unspecified boolean value in PVS.

Some PVS definitions are used to express the proof obligations in a more convenient form, which is closer to the original JML format and closer to the conventional notion of Hoare triple:

$$\begin{array}{rl}\forall z\colon \mathsf{OM}. \quad \mathsf{SB}\cdot (& \mathsf{requires} = \lambda x\colon \mathsf{OM}.\ [\![Pre]\!]\,(x) \wedge x = z, \\ & \mathsf{statement} = [\![m]\!]\,, \\ & \mathsf{ensures} = \lambda x\colon \mathsf{OM}.\ [\![Post_{norm}]\!]\,(x, z) \\ & \qquad\qquad\qquad \wedge\ \forall l \notin [\![Assign]\!]\,.\ x.l = z.l, \\ & \mathsf{signals} = \lambda x\colon \mathsf{OM}.\ [\![Post_{excp}]\!]\,(x, z) \\ & \qquad\qquad\qquad \wedge\ \forall l \notin [\![Assign]\!]\,.\ x.l = z.l\ ) \end{array} \tag{4}$$

Here SB, an abbreviation of statement behaviour, is a function acting on a labelled record with four fields. A so-called logical variable z is used to relate pre- and post-state and to give a meaning to uses of `\old` in postconditions.

If we omit the signals clause, we effectively have a traditional Hoare triple, consisting of a precondition, a statement, and a postcondition. But note that unlike in conventional Hoare logics, $[\![m]\!]$ is not a statement in Java syntax, but rather its denotation, so we have a Hoare logic at the semantic rather than the syntactic level. However, given that the semantics is compositional, $[\![m]\!]$ has the same structure as m in Java syntax, and reasoning with these 'semantic' Hoare tuples is essentially the same as reasoning with conventional 'syntactic' Hoare tuples.

We have omitted some further complications in the discussion above. Firstly, rather than having Hoare 4-tuples as in (4), giving a precondition, statement, postcondition, and exceptional postcondition, we actually have Hoare 8-tuples, to cope with the other forms of abrupt control flow in Java, by return, break, and continue statements, and to specify whether or not the statement is allowed to diverge, allowing us to specify both total and partial correctness. Secondly, in addition to have Hoare n-tuples for statements, we also have them for expressions.

The format for our proof obligations in (4) is convenient for several reasons. It is more readable, as it closely resembles the original JML specification. Moreover, it allows the formulation of suitable PVS theories, i.e. collections of PVS theorems, that can be used to prove proof obligations of this forms in a convenient way. How we go about proving these Hoare n-tuples is the subject of the next section.

5 Verification phase

Using the semantics of Java and JML discussed in the previous sections, the LOOP tool translates JML-annotated Java code to proof obligations in PVS. This is by no means the end of the story. A lot of PVS infrastructure, in the form of theories and PVS proof strategies, is needed to make proving these proof obligations feasible.

This section describes the three different techniques that we have developed for this. Each of these techniques involves a collection of PVS lemmas, in the hand-written prelude or the PVS theories generated by the LOOP tool, and associated PVS strategies, which can automate large parts of proofs.

5.1 Semantical proofs by symbolic execution

In the early, semantical phase of the LOOP project—described in Sect. 3—the proofs were "semantical". What this means is that these proofs worked by using the underlying semantics: the method bodies involved are expanded (in a controlled fashion) until nothing remains but a series of put and get operations on the memory model, for which the required properties are established.

These semantical proofs can be very efficient, but they only work well for very small programs, without complicated control structures like (while or for) loops. To verify the correctness of larger programs we use the program logics described in the next sections, which provide Hoare logics and weakest precondition calculi for Java. These logics are used to break complicated method bodies iteratively to simple parts, for which semantical reasoning can be used. Hence semantical proofs are not abolished, but are postponed to the very end, namely to the leafs of the proof trees.

5.2 Hoare logic

As we already emphasised, Java programs have various termination modes: they can hang, terminate normally, or terminate abruptly (typically because of an exception). An adaptation of Hoare logic for Java with different termination modes has been developed in [15]. It involves separate rules for the different termination modes. Although this logic has been used successfully for several examples, it is not very convenient to use because of the enormous number of rules involved.

With the establishment of JML as specification language, it became clear that the most efficient logic would not use Hoare triples, but the "JML n-tuples" already discussed Sect. 4. The proof obligation resulting for a JML specification for a Java method is expressed as a Hoare n-tuple, and for these n-tuples we we can formulate and prove deduction rules like those used in conventional Hoare logic, one for every programming language construct. For example, for composition we have the deduction rule

$$\frac{\begin{array}{ll}\mathsf{SB}\cdot(\, \mathsf{requires} = Pre, & \mathsf{SB}\cdot(\, \mathsf{requires} = Q, \\ \quad\quad \mathsf{statement} = m_1, & \quad\quad \mathsf{statement} = m_2, \\ \quad\quad \mathsf{ensures} = Q, & \quad\quad \mathsf{ensures} = Post_{norm}, \\ \quad\quad \mathsf{signals} = Post_{excp}\ ) & \quad\quad \mathsf{signals} = Post_{excp}\ ) \end{array}}{\begin{array}{l}\mathsf{SB}\cdot(\, \mathsf{requires} = Pre, \\ \quad\quad \mathsf{statement} = m_1; m_2, \\ \quad\quad \mathsf{ensures} = Post_{norm}, \\ \quad\quad \mathsf{signals} = Post_{excp}\ ) \end{array}}$$

NB. the rule above can be *proved* as a lemma inside PVS, ie. we have proved that this rule is sound our Java semantics. All the deduction rules have been proved correct in this way, establishing soundness of the Hoare logic. Because we have a shallow embedding rather than a deep embedding of Java in PVS, proving completeness of the rules is not really possible. For more details about this Hoare logic for for Java and JML, see [18].

The bottleneck in doing Hoare logic proofs is, as usual, providing the intermediate assertions, such as Q in the proof rule for composition above. This is aggravated by the fact that our Hoare logic works at the semantical level, which means that these intermediate assertions have to be expressed at semantical level and are therefore less readable that they would be in a Hoare logic at the syntactic level. One way to alleviate this is to specify intermediate assertions in the Java program; JML provides the `assert` keyword to do this, and the LOOP tool could parse these assertions to provide the (less readable) semantical counterparts, which can then serve as the intermediate predicate in the Hoare logic proof. Another way, discussed in the next subsection, is the use of a weakest precondition calculus.

5.3 Weakest precondition

The latest stage in the development of PVS infrastructure for easing the job of verifying Java programs has been the development of a weakest precondition (wp) calculus. Here we just discuss the main ideas; for more details see [18].

The fact that JML distinguishes normal and exceptional postconditions means a wp-calculus for Java and JML is slightly different than usual. Where normally the wp function acts on a statement s and a postcondition $Post$ to provide the weakest precondition $\mathsf{wp}(s, Post)$, our wp function will act on a statement s and a pair of postconditions $(Post_{norm}, Post_{excp})$, the postcondition for normal termination and exceptional termination, respectively[4].

We can actually define the weakest precondition function inside PVS, as follows:

$$\mathsf{wp}(s, Post_{norm}, Post_{excp}) \stackrel{\text{def}}{=} \lambda x.\ \exists P.\ P(x) \wedge \mathsf{SB} \cdot (\begin{array}{l} \mathsf{requires} = P, \\ \mathsf{statement} = m, \\ \mathsf{ensures} = Post_{norm}, \\ \mathsf{signals} = Post_{excp}\ ) \end{array} \tag{5}$$

The definition above is not useful in the sense that we can ask PVS to compute it. Instead, we prove suitable lemmas about the wp function as defined above, one for every language construct, and then use these lemmas as rewrite rules in PVS to compute the weakest preconditions. Eg., for the composition we prove that

$$\begin{array}{l} \mathsf{wp}(m_1; m_2, Post_{norm}, Post_{excp}) \\ = \mathsf{wp}(m_1, \mathsf{wp}(m_2, Post_{norm}, Post_{excp}), Post_{excp}) \end{array}$$

and we can use this lemma as a rewriting rule in PVS to compute (or rather, to let PVS compute) the weakest precondition.

[4] As before, we oversimplify here; instead of 2 postconditions, our wp function actually works on 5 postconditions, to cope not just with normal termination and exceptions, but also with (labelled) breaks and continues and with return statements.

The definition of $\mathsf{wp}($, i)n terms of the notion of Hoare n-tuple above in (5) allows us to prove the correctness of these lemma in PVS, thus establishing the correctness of our wp-calculus wrt. our Java semantics. So as for soundness of our Hoare logic, for correctness of our wp-calculus wrt. the Java semantics we have a completely formal proof, inside PVS.

It turns out that there are several ways to go about computing weakest preconditions. The traditional, 'backward', approach is to peel of statements at the back,

$$\begin{aligned}&\mathsf{wp}(m_1; m_2; \ldots; m_n, Post_{norm}, Post_{excp})\\ &= \mathsf{wp}(m_1; \ldots; m_{n-1}, \mathsf{wp}(m_n, Post_{norm}, Post_{excp}), Post_{excp})\end{aligned}$$

and then evaluate the inner call to wp, ie. $\mathsf{wp}(m_n, Post_{norm}, Post_{excp})$, before peeling of the next statement m_{n-1}. But an alternative, 'forward', approach is to begin by peeling of statements at the front

$$\begin{aligned}&\mathsf{wp}(m_1; m_2; \ldots; m_n, Post_{norm}, Post_{excp})\\ &= \mathsf{wp}(m_1, \mathsf{wp}(m_2; \ldots; m_n, Post_{norm}, Post_{excp}), Post_{excp})\end{aligned}$$

and then to evaluate the outer call to wp. Of course, the approaches are logically equivalent, and will ultimately produce the same result. But computationally the approaches are different, and the costs of letting PVS compute the wp-function, measured in the time and memory PVS needs for the computation, turn out to be very different for the two approaches.

Two strategies for letting PVS computing weakest preconditions have been implemented in PVS, in a forward or backward style sketched above. Each of these implementation relies on its own collection of lemmas serving as rewrite rules and relies on a a different PVS proof tactic. For a discussion of these strategies we refer to [18].

Using the wp-calculi, program verification can be completely automated in PVS. However, there are limitations to the size of program fragment that one can verify using the wp-calculi. PVS can run for hours without completing the proof, or it can crash because the proof state becomes too big. The most effective approach for larger programs seems to a combination of Hoare logic to break up proof obligation for a large program into chunks that are small enough to handle with the wp-calculus.

6 Related Work

There are quite a number of groups working on the tool-supported verification of Java programs that use theorem provers, such as ESC/Java [8], Bali [30], JACK [5], Krakatoa [26], Jive [27], and Key [7]. With more of such tools now appearing, comparing them and understanding their fundamental differences and similarities becomes more important.

In a way, ESC/Java is the odd one out in the list above, in that its developers have deliberately chosen an approach that is unsound and incomplete, as their aim was to maximise the number of (potential) bugs that the tool can spot bugs fully automatically. Still, ESC/Java does use a theorem prover and a wp-calculus to find these potential bugs, and it probably the most impressive tool to date when it comes to showing the potential of program verification.

Of the tools listed above, ESC/Java, JACK, Krakatoa, and LOOP use JML as specification language. Jive will also start supporting JML in the near future. The KeY system uses OCL as specification language. The fact that several tools support JML has its advantages of course: it makes it easy to compare tools, to work on common case studies, to reuse each other's specifications (especially for APIs), or to use different tools on different parts of the same program.

The tools differ a lot in the fragment of Java that they support, and how complicated a specification language they support. Currently, the LOOP tool and ESC/Java probably cover the largest subset of Java, and the LOOP tool probably supports the most complicated specification language.

One distinguishing feature of the LOOP project is that it uses a shallow embedding of Java in PVS. This has both advantages and disadvantages.

An advantage is that is has allowed us to give a completely formal proof of the soundness of all the programming logic we use, inside the theorem prover PVS. Of the projects listed above, only in Bali and LOOP has the soundness of the programming logic been formally proved using a theorem prover. In fact, in the Bali project, where a deep embedding is used, completeness has also been proved, which is hard for a shallow embedding.

A disadvantage of the use of a shallow embedding is that much of the reasoning takes places at the semantic level, rather than the syntactical level, which means that during the proof we have an uglier and, at least initially, less familiar syntax to deal with. Using the LOOP tool and PVS to verify programs requires a high level of expertise in the use of PVS, and an understanding of the way the semantics of Java and JML has been defined.

A difference between LOOP and many of the others approaches (with the possible exception of Bali, we suspect) is in the size of the proof obligations that are produced as input to the theorem prover. The LOOP tool produces a single, big, proof obligation in PVS for every method, and then relies on the capabilities of PVS to reduce this proof obligation into ever smaller ones which we can ultimately prove. Most of the other tools already split up the proof obligation for a single method into smaller chunks (verification conditions) before feeding them to the theorem prover, for instance by using wp-calculi. A drawback of the LOOP approach in that the capabilities of theorem prover become a bottleneck sooner than in the other approaches. After all, there is a limit to the size of proofs that PVS—or any other theorem prover for that matter—can handle before becoming painfully slow or simply crashing. Note that this is in a way a consequence of the use of a shallow embedding, and of formalising and proving the correctness of the entire programming logic inside the theorem prover.

7 Perspective

Most of the case studies for which the LOOP tool has been used are so-called Java Card programs designed to run on smart cards. Java Card programs are an excellent target for trying out formal methods, because they are small, their correctness is of crucial importance, and they use only a very limited API (for much of which we have developed a formal specification in JML [32]). Many of the groups working on formal verification for Java are targeting Java Card. Indeed, the JACK tool discussed above has been developed by a commercial smartcard manufacturer.

For examples of such case studies see [3] or [20]. The largest case study to date that has been successfully verified using the LOOP tool and PVS is a Java Card applet of about several hundred lines of code provided by one of the industrial partners in the EU-sponsored VerifiCard project (`www.verificard.com`). What is interesting about this case study is that it is a real commercial application (so it might be running on your credit card ...), and that verification has revealed bugs, albeit bugs that do not compromise the security.

Even if program verification for small programs such as smart card applets is becoming technically possible, this does not mean that it will become feasible to do it an industrial practice. Any industrial use of formal methods will have to be economically justified, by comparing the costs (the extra time and effort spent, not just for verification, but also for developing formal specifications) against the benefits (improvements in quality, number of bugs found). Here one of the benefits of JML as a specification language is that there is a range of tools that support JML that can be used to develop and check specifications. For example, the JML runtime assertion checker [6] *tests* whether programs meet specifications. This clearly provides less assurance than program verification, but requires a lot less effort, and may provide a very cost-effective way of debugging of code and formal specifications. Similarly, ESC/Java can be used to automatically check for bugs in code and annotations. We believe that having a range of tools, providing different levels of assurance at different costs, is important if one really wants to apply formal methods in practice. For an overview of the different tools for JML, see [4].

Despite the progress in the use of theorem-provers for program verification in the past years, there are still some deep, fundamental, open research problems in the field of program verification for Java or object-oriented programs in general.

Firstly, as already mentioned in Sect. 3.2, the problems caused by references, such as aliasing and the leakage of references, remains a bottleneck in formal verification. One would hope that these problems can be mitigated by introducing suitable notions of ownership or encapsulation.

Another, and somewhat related issue, is how to deal with class invariants. In the paper we have only mentioned pre- and postconditions, and we have ignored the notion of class invariant that JML provides. Class invariants are properties that have to be preserved by all the methods; in other words, all class invariants are implicitly included in the pre- and postcondition of every method. However, there is a lot more to class invariants than this, and class invariants are *not* just a useful abbreviation mechanism. For instance, for more complex object structures, ruling out that that methods disturb invariants of other objects is a serious challenge.

Finally, even though security-sensitive smart cards programs provide an interesting application area for program verification, it is usually far from clear what it really means for an application to be 'secure', let alone how to specify this formally. How to specify relevant security properties in a language like JML remains to a largely open research question.

Acknowledgements Thanks to all the former and current members of the LOOP group—Joachim van den Berg, Cees-Bart Breunesse, Ulrich Hensel, Engelbert Hubbers, Marieke Huisman, Joe Kiniry, Hans Meijer, Martijn Oostdijk, Hendrik Tews, and Martijn Warnier—for their contributions.

References

1. J. van den Berg, M. Huisman, B. Jacobs, and E. Poll. A type-theoretic memory model for verification of sequential Java programs. In D. Bert, C. Choppy, and P. Mosses, editors, *Recent Trends in Algebraic Development Techniques*, number 1827 in Lect. Notes Comp. Sci., pages 1–21. Springer, Berlin, 2000.
2. J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, number 2031 in Lect. Notes Comp. Sci., pages 299–312. Springer, Berlin, 2001.
3. C.-B. Breunesse, J. van den Berg, and B. Jacobs. Specifying and verifying a decimal representation in Java for smart cards. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology and Software Technology*, number 2422 in Lect. Notes Comp. Sci., pages 304–318. Springer, Berlin, 2002.
4. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An overview of JML tools and applications. In Th. Arts and W. Fokkink, editors, *Formal Methods for Industrial Critical Systems (FMICS 03)*, volume 80 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 73–89. Elsevier, 2003.
5. N. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: A developer-oriented approach. In *Formal Methods Europe (FME)*, LNCS. Springer Verlag, 2003.
6. Y. Cheon and G.T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In Hamid R. Arabnia and Youngsong Mun, editors, *the International Conference on Software Engineering Research and Practice (SERP '02)*, pages 322–328. CSREA Press, 2002.
7. W. Ahrendt et al. The key tool. Technical report in computing science no. 2003-5, Dept. of Computing Science, Chalmers University and Göteborg University, Göteborg, Sweden, 2003.
8. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'2002)*, pages 234–245, 2002.

9. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, 2000. `http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html`.
10. D. Griffioen and M. Huisman. A comparison of PVS and Isabelle/HOL. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics*, number 1479 in Lect. Notes Comp. Sci., pages 123–142. Springer, Berlin, 1998.
11. U. Hensel. *Definition and Proof Principles for Data and Processes*. PhD thesis, Techn. Univ. Dresden, Germany, 1999.
12. U. Hensel, M. Huisman, B. Jacobs, and H. Tews. Reasoning about classes in object-oriented languages: Logical models and tools. In Ch. Hankin, editor, *European Symposium on Programming*, number 1381 in Lect. Notes Comp. Sci., pages 105–121. Springer, Berlin, 1998.
13. M. Huisman. *Reasoning about JAVA Programs in higher order logic with PVS and Isabelle*. PhD thesis, Univ. Nijmegen, 2001.
14. M. Huisman and B. Jacobs. Inheritance in higher order logic: Modeling and reasoning. In M. Aagaard and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, number 1869 in Lect. Notes Comp. Sci., pages 301–319. Springer, Berlin, 2000.
15. M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, number 1783 in Lect. Notes Comp. Sci., pages 284–303. Springer, Berlin, 2000.
16. B. Jacobs. Objects and classes, co-algebraically. In B. Freitag, C.B. Jones, C. Lengauer, and H.-J. Schek, editors, *Object-Orientation with Parallelism and Persistence*, pages 83–103. Kluwer Acad. Publ., 1996.
17. B. Jacobs. Java's integral types in PVS. In *FMOODS 2003 Proceedings*, Lect. Notes Comp. Sci. Springer, Berlin, 2003, to appear.
18. B. Jacobs. Weakest precondition reasoning for Java programs with JML annotations. *Journ. of Logic and Algebraic Programming*, To appear.
19. B. Jacobs, J. Kiniry, and M. Warnier. Java program verification challenges. In *FMCO 2002 Proceedings*, Lect. Notes Comp. Sci. Springer, Berlin, 2003, to appear.
20. B. Jacobs, M. Oostdijk, and M. Warnier. Formal verification of a secure payment applet. *Journ. of Logic and Algebraic Programming*, To appear.
21. B. Jacobs and E. Poll. Coalgebras and monads in the semantics of Java. *Theor. Comp. Sci.*, 291(3):329–349, 2002.
22. B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about classes in Java (preliminary report). In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 329–340. ACM Press, 1998.
23. G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. Technical Report 03-04, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, March 2003. To appear in the proceedings of FMCO 2002.
24. G.T. Leavens, A.L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov and B. Rumpe, editors, *Behavioral Specifications of Business and Systems*, pages 175–188. Kluwer, 1999.
25. G. Malcolm. Behavioural equivalence, bisimulation and minimal realisation. In M. Haveraaen, O. Owe, and O.J. Dahl, editors, *Recent Trends in Data Type Specification*, number 1130 in Lect. Notes Comp. Sci., pages 359–378. Springer, Berlin, 1996.

26. C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for JML/Java program certification. *Journal of Logic and Algebraic Programming*, 2003. To appear.
27. J. Meyer and A. Poetzsch-Heffter. An architecture for interactive program provers. In S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, number 1785 in Lect. Notes Comp. Sci., pages 63–77. Springer, Berlin, 2000.
28. P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular specification of frame properties in JML. *Concurrency and Computation: Practice and Experience*, 15(2):117–154, 2003.
29. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
30. D. von Oheimb and T. Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. In L.-H. Eriksson and P. Lindsay, editors, *Formal Methods – Getting IT Right (FME'02)*, volume 2391 of *LNCS*, pages 89–105. Spinger Verlag, 2002.
31. S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification*, number 1102 in Lect. Notes Comp. Sci., pages 411–414. Springer, Berlin, 1996.
32. E. Poll, J. van den Berg, and B. Jacobs. Formal specification of the JavaCard API in JML: the APDU class. *Computer Networks*, 36(4):407–421, 2001.
33. H. Reichel. An approach to object semantics based on terminal co-algebras. *Math. Struct. in Comp. Sci.*, 5:129–152, 1995.
34. J. Rothe, H. Tews, and B. Jacobs. The coalgebraic class specification language CCSL. *Journ. of Universal Comp. Sci.*, 7(2), 2001.
35. H. Tews. *Coalgebraic Methods for Object-Oriented Specification.* PhD thesis, Techn. Univ. Dresden, Germany, 2002.