A Logic for the Java Modeling Language JML

Bart Jacobs and Erik Poll

Dept. Computer Science, Univ. Nijmegen,
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands.
`{bart,erikpoll}@cs.kun.nl` `http://www.cs.kun.nl/~{bart,erikpoll}`

Abstract. This paper describes a specialised logic for proving specifications in the Java Modeling Language (JML). JML is an interface specification language for Java. It allows assertions like invariants, constraints, pre- and post-conditions, and modifiable clauses as annotations to Java classes, in a design-by-contract style. Within the LOOP project at the University of Nijmegen JML is used for specification and verification of Java programs. A special compiler has been developed which translates Java classes together with their JML annotations into logical theories for a theorem prover (PVS or Isabelle). The logic for JML that will be described here consists of tailor-made proof rules in the higher order logic of the back-end theorem prover for verifying translated JML specifications. The rules efficiently combine partial and total correctness (like in Hoare logic) for all possible termination modes in Java, in a single correctness formula.

1 Introduction

JML (for Java Modeling Language) [15, 14] is a specification language tailored to Java, primarily developed at Iowa State University. It allows assertions to be included in Java code, specifying for instance pre- and postconditions and invariants in the style of Eiffel and the design-by-contract approach [18]. JML has been integrated with the specification language used for ESC/Java, the extended static checker developed at Compaq System Research Center [17, 27].

At Nijmegen, a formal denotational semantics has been developed for essentially all of sequential Java. A compiler has been built, the LOOP tool [5], which translates a Java program into logical theories describing its semantics [13, 4, 10, 9, 12, 8, 26]. These logical theories are in a format that can serve as input for theorem provers, which can then be used to prove properties of the Java program, thus achieving a high level of reliability for this program. The LOOP tool supports output for the theorem provers PVS [20] and Isabelle [21]. This approach to verification of Java has demonstrated its usefulness for instance with the proof of a non-trivial invariant for the Vector class in the standard Java API [11]. The current main application area is JavaCard [1], see [24, 25]. The LOOP tool is being extended to JML, so that it can be used to verify JML-annotated Java source code. At the moment this works for a kernel of JML.

One advantage of using a formal specification language is that tool support becomes possible. Work on tool support for JML focuses on the generation of run-time checks on preconditions for testing, at Iowa State University [6] extended static checking, at Compaq System Research Center, and verification using the LOOP tool, at the University of Nijmegen. This offers a wide range of validation options—a key advantage of JML.

This paper presents a logic for reasoning about (sequential) Java programs which is the result of several years of experience in this area. The semantical and logical approach to Java within the LOOP project is bottom-up: it starts from an (automatic) translation of Java programs into what is ultimately a series of low level get- and put-operations on a suitable memory model [4]. From this point onwards, several steps have been taken up the abstraction ladder.

1. At first, the results to be proved (about the Java program under consideration) were formulated in the higher order logic of the back-end theorem prover (PVS or Isabelle), and proved by fully unpacking their meaning in terms of the low level (get and put) operations on the memory. Only relatively small programs can be handled like this, despite the usefulness of automatic rewriting.
2. Next a suitable Hoare logic for Java was introduced (in PVS and Isabelle) [10] for compositional reasoning about programs at a higher level of abstraction. This logic has different Hoare triples, corresponding to partial / total correctness for each of the possible termination modes of Java statements and expressions (normal / exception / return / break / continue). In theory this logic is appropriate, but in practice it involves too many rules and leads to too many duplications of proof obligations (for the different termination modes).
3. In a further abstraction step, the results to be proved were no longer formulated in PVS or Isabelle, but in a suitable specification language for Java, namely JML [14]. JML makes it possible to write specifications for Java programs without the need to know the details of these programs in PVS/Isabelle. Again, the translation from (a kernel of) JML to PVS/Isabelle is done automatically.
4. In a final step—the topic of this paper—a tailor-made logic is developed for proving (in PVS/Isabelle) these JML specifications. This logic involves syntax-driven rules (in PVS/Isabelle), supported by appropriate proof strategies, enabling the user to go step-by-step through a method body. The logic combines partial and total correctness together with the five different termination modes in a single correctness formula, resembling JML method specifications. This makes the logic both powerful and efficient in its use. Soundness of all these rules has been proved on the basis of the underlying semantics for Java. Most of the proofs are easy and just involve many case distinctions. The soundness of the while rule, see Subsection 5.6, is non-trivial.

The rules we describe below only handle the standard, imperative part of (sequential) Java, and not its typically object-oriented features (dealing for example with dynamic binding), as in [22, 19]. We do not need these rules because we can always fall back on our low level semantics where these issues are handled automatically [9]. This is a crucial point. Our logic for JML is not used directly at the Java source code level—as is standard in Hoare logics, see [3, 16, 7, 22]—but at the translated Java code in the back-end theorem prover, *i.e.* on the semantical level. But since the translation performed by the LOOP tool is compositional, there is not much difference: during proofs in the logic for JML one still follows the original code structurally. In a forward approach (following the execution order) one typically peels off the leading statements step-by-step, adapting the precondition in a suitable way. In every step one has to prove this adaptation of the precondition, as a result of the leading statement. In our approach the latter is typically done *without* the logic for JML, by going down to the lowest semantical level (as in 1 above), making efficient use of automatic rewriting. As mentioned, this works well for small programs. Note that an important consequence of working at the semantic level

This combination of high level proof rules and low level automatic rewriting on the basis of the underlying semantics forms the strength of our pragmatic approach, where we only introduce logical rules when this really suits us, in order to achieve a higher level of abstraction in proofs. A consequence of working at the semantical level is that we cannot really define a notion of completeness for our higher level rules (like in [19]), because completeness can only be defined for a syntactic level w.r.t. some lower semantic level.

In this paper we shall only talk about proving JML specifications for certain Java implementations. We shall not use this here, but in certain cases these proofs may actually rely on other JML specifications, for example for methods which are native (implemented in some other language than Java), or which may be overridden. In the latter case one cannot rely on a specific implementation, because it may be different in subclasses. In a behavioural approach to subtyping [2] (see also [23]) one then assumes that all implementations in subclasses satisfy the specification in the class in which the method is first introduced. This specification will form the basis for verifications.

In order to explain our logic for JML, the paper will have to introduce quite a few languages: Java and its JML annotations (Section 2), higher order logic (as used in PVS and Isabelle) and the representation of Java statements and expressions therein (Section 3), the meaning of JML method specifications in logic (Section 4), and finally the rules themselves. Necessarily we cannot describe all details, and are forced to concentrate on the essentials. The paper involves an example specification in JML, verified in PVS using the logic for JML. It is the same example as in [10]—this time not on abstraction level 2 but on level 4, as described above.

2 Class and method specifications in JML

This section gives a brief impression of JML, concentrating on method specifications. For more information, see [15, 14]. JML adds assertions to Java by writing them as special comments (`/*@ ... @*/` or `//@ ...`). These assertions are Java Boolean expressions extended with special operators, like `\forall`, `\exists`, `\result` or `\old(-)`. Classes can be enriched with invariants (predicates that should be preserved by all methods) or history constraints (relations that should hold between all pre- and post-states of all methods). Methods can be annotated with behaviour specifications which can be either `normal_behavior`, `exceptional_behavior` or simply `behavior`. The latter is typically used as follows for specifying a method `m`.

```
/*@ behavior
  @   diverges: <pre-condition for non-termination>
  @   requires: <precondition>
  @ modifiable: <items that can be modified>
  @    ensures: <postcondition for normal termination>
  @    signals: <postcondition for exceptional termination>
  @*/
void m() { ... }
```

Roughly, this says that if the precondition holds, then if the method `m` hangs / terminates normally / terminates abruptly, then the diverges / ensures / signals clause holds (respectively). When the `diverges` is `true` (resp. `false`) we have partial (resp. total) correctness. But note that when it is `false`, the method can still terminate abruptly. A `normal_behavior` (or `exceptional_behavior`) describes a situation where a method *must* terminate normally (or exceptionally), assuming that the precondition holds. For example, the class in Figure 1 contains an annotated method (from [10]) that searches for a certain pattern in an array using a single while loop. It has a non-trivial postcondition.

3 Semantics of Java statements and expressions

This section introduces a denotational semantics of Java statements and expressions in higher order logic. This logic is a common abstraction of the logics used by PVS and Isabelle/HOL, and will be introduced as we proceed.

First, there is a complicated type OM, for object memory, with various get- and put-operations, see [4]. In this paper the internal structure of OM is not relevant. The type OM serves as our state space on which statements and expressions act, as functions $\mathsf{OM} \to \mathsf{StatResult}$ and $\mathsf{OM} \to \mathsf{ExprResult}[\alpha]$, for a suitable result type α. These result types are introduced as labeled coproduct (also called variant or sum) types:

$$\begin{array}{ll}
\mathsf{StatResult} : \mathsf{TYPE} \stackrel{\mathrm{def}}{=} & \qquad \mathsf{ExprResult}[\alpha] : \mathsf{TYPE} \stackrel{\mathrm{def}}{=} \\
\quad \{\, \mathsf{hang} \colon \mathsf{unit} & \qquad \quad \{\, \mathsf{hang} \colon \mathsf{unit} \\
\quad \mid \mathsf{norm} \colon \mathsf{OM} & \qquad \quad \mid \mathsf{norm} \colon [\, \mathsf{ns} \colon \mathsf{OM}, \mathsf{res} \colon \alpha \,] \\
\quad \mid \mathsf{abnorm} \colon \mathsf{StatAbn} \,\} & \qquad \quad \mid \mathsf{abnorm} \colon \mathsf{ExprAbn} \,\}
\end{array}$$

with labels hang, norm and abnorm corresponding to the three termination modes in Java: non-termination, normal termination and abrupt termination. Notice

```
class Pattern {

    int [] base, pattern;

    /*@ normal_behavior
      @   requires: base != null && pattern != null &&
      @             pattern.length <= base.length;
      @ modifiable: \nothing;
      @    ensures: ( /// pattern occurs;
      @             \result >= 0 &&
      @             \result <= base.length - pattern.length &&
      @               /// \result gives the start position
      @             (\forall (int i)
      @                 0 <= i && i < pattern.length
      @                    ==> pattern[i] == base[\result+i]) &&
      @               /// pattern does not occur earlier
      @             (\forall (int j)
      @                 0 <= j && j < \result
      @                    ==> (\exists (int i)
      @                            0 <= i && i < pattern.length
      @                               && pattern[i] != base[j+i])))
      @             ||
      @             ( /// pattern does not occur
      @             \result == -1 &&
      @             (\forall (int j)
      @                 0 <= j && j < base.length - pattern.length
      @                    ==> (\exists (int i)
      @                            0 <= i && i < pattern.length
      @                               && pattern[i] != base[j+i])));
      @*/
    int find_first_occurrence () {
        int p = 0, s = 0;
        while (true)
            if (p == pattern.length) return s;
            else if (s + p == base.length) return -1;
            else if (base[s + p] == pattern[p]) p++;
            else { p = 0; s++; }
    }
}
```

Fig. 1. A pattern search method in Java with JML annotation

that a normally termination expression returns both a state (incorporating the possible side-effect) and a result value. This is indicated by a labeled product (record) type $[\,\mathsf{ns}\colon \mathsf{OM}, \mathsf{res}\colon \alpha\,]$. The result types $\mathsf{StatAbn}$ and $\mathsf{ExprAbn}$ for abrupt termination are subdivided differently for statements and expressions:

$$\begin{array}{ll}\mathsf{StatAbn} : \mathsf{TYPE} \stackrel{\mathrm{def}}{=} & \\ \quad \{\, \mathsf{excp}\colon [\,\mathsf{es}\colon \mathsf{OM}, \mathsf{ex}\colon \mathsf{RefType}\,] & \qquad \mathsf{ExprAbn} : \mathsf{TYPE} \stackrel{\mathrm{def}}{=} \\ \quad \mid \mathsf{rtrn}\colon \mathsf{OM} & \qquad\quad [\,\mathsf{es}\colon \mathsf{OM}, \mathsf{ex}\colon \mathsf{RefType}\,] \\ \quad \mid \mathsf{break}\colon [\,\mathsf{bs}\colon \mathsf{OM}, \mathsf{blab}\colon \mathsf{lift}[\mathsf{string}]\,] & \\ \quad \mid \mathsf{cont}\colon [\,\mathsf{cs}\colon \mathsf{OM}, \mathsf{clab}\colon \mathsf{lift}[\mathsf{string}]\,]\,\} & \end{array}$$

The type $\mathsf{RefType}$ is used for references, containing either the null-reference or a pointer to a memory location. It describes the reference to an exception object, in case an exception is thrown. The lift type constructor adds a bottom element bot to an arbitrary type, and keeps all original elements a as $\mathsf{up}\, a$. It is used because `break` and `continue` statements in Java can be used both with and without label (represented as string).

On the basis of this representation of statements and expressions all language constructs from (sequential) Java are formalised in type theory (and used in the translation performed by the LOOP tool). For instance, the composition of two statements $s, t\colon \mathsf{OM} \to \mathsf{StatResult}$ is defined as:

$$(s\,;t) \stackrel{\mathrm{def}}{=} \lambda x\colon \mathsf{OM}.\, \mathsf{CASES}\ s \cdot x\ \mathsf{OF}\ \{ \begin{array}{l} \ \mathsf{hang} \mapsto \mathsf{hang} \\ \mid \mathsf{norm}\, y \mapsto t \cdot y \\ \mid \mathsf{abnorm}\, a \mapsto \mathsf{abnorm}\, a\,\} \end{array}$$

where $\cdot$ is used for function application, and CASES for pattern matching on the labels of the $\mathsf{StatResult}$ coproduct type. What is important to note is that if the statement s hangs or terminates abruptly, then so does the composition $s\,;t$.

There is no space to describe all these constructs in detail. We mention some of them that will be used later. Sometimes we need to execute an expression only for its side-effect (if any). This is done via the function $\mathsf{E2S}$, defined as:

$$\begin{array}{l} \mathsf{E2S} \cdot e\ :\ \mathsf{OM} \to \mathsf{StatResult} \stackrel{\mathrm{def}}{=} \\ \quad \lambda x\colon \mathsf{OM}.\ \mathsf{CASES}\ e \cdot x\ \mathsf{OF}\ \{ \\ \qquad\qquad \ \mathsf{hang} \mapsto \mathsf{hang} \\ \qquad\qquad \mid \mathsf{norm}\, y \mapsto \mathsf{norm}(y.\mathsf{ns}) \\ \qquad\qquad \mid \mathsf{abnorm}\, a \mapsto \mathsf{abnorm}(\mathsf{excp}(\mathsf{es} = a.\mathsf{es}, \mathsf{ex} = a.\mathsf{ex}))\,\} \end{array}$$

for $e\colon \mathsf{OM} \to \mathsf{ExprResult}[\alpha]$. The notation $y.\mathsf{ns}$ describes field selection associated with y in the labeled product $[\,\mathsf{ns}\colon \mathsf{OM}, \mathsf{res}\colon \alpha\,]$. In the last line an expression abnormality (an exception) is transformed into a statement abnormality. Java's `if-then-else` becomes:

$$\begin{array}{l} \mathsf{IF\text{-}THEN\text{-}ELSE} \cdot c \cdot s \cdot t\ :\ \mathsf{OM} \to \mathsf{StatResult} \stackrel{\mathrm{def}}{=} \\ \quad \lambda x\colon \mathsf{OM}.\ \mathsf{CASES}\ c \cdot x\ \mathsf{OF}\ \{ \\ \qquad\qquad \ \mathsf{hang} \mapsto \mathsf{hang} \\ \qquad\qquad \mid \mathsf{norm}\, y \mapsto \mathsf{IF}\ y.\mathsf{res} \mathsf{THEN}\ s \cdot (y.\mathsf{ns}) \mathsf{ELSE}\ t \cdot (y.\mathsf{ns}) \\ \qquad\qquad \mid \mathsf{abnorm}\, a \mapsto \mathsf{abnorm}(\mathsf{excp}(\mathsf{es} = a.\mathsf{es}, \mathsf{ex} = a.\mathsf{ex}))\,\} \end{array}$$

for $c\colon \mathsf{OM} \to \mathsf{ExprResult[bool]}$ and $s, t\colon \mathsf{OM} \to \mathsf{StatResult}$. The formalisation of Java's `return` statement (without argument) is:

$$\mathsf{RETURN} \ :\ \mathsf{OM} \to \mathsf{StatResult} \stackrel{\text{def}}{=} \lambda x\colon \mathsf{OM}.\, \mathsf{abnorm}(\mathsf{rtrn}\, x)$$

This statement produces an abnormal "return" state. Such a return abnormality can be undone, via appropriate catch-return functions. In our translation of Java programs, such a function CATCH-RETURN is wrapped around every method body that returns `void`. First the method body is executed. This may result in an abnormal state, because of a return. In that case the function CATCH-RETURN turns the state back to normal again. Otherwise, it leaves everything unchanged.

$$\begin{array}{l}
\mathsf{CATCH\text{-}RETURN} \cdot s \ :\ \mathsf{OM} \to \mathsf{StatResult[OM]} \stackrel{\text{def}}{=} \\
\quad \lambda x\colon \mathsf{OM}.\ \mathsf{CASES}\ s \cdot x\ \mathsf{OF}\ \{ \\
\qquad\qquad \mathsf{hang} \mapsto \mathsf{hang} \\
\qquad\qquad \mid \mathsf{norm}\, y \mapsto \mathsf{norm}\, y \\
\qquad\qquad \mid \mathsf{abnorm}\, a \mapsto\ \mathsf{CASES}\ a\ \mathsf{OF}\ \{ \\
\qquad\qquad\qquad\qquad \mathsf{excp}\, e \mapsto \mathsf{abnorm}(\mathsf{excp}\, e) \\
\qquad\qquad\qquad\qquad \mid \mathsf{rtrn}\, z \mapsto \mathsf{norm}\, z \\
\qquad\qquad\qquad\qquad \mid \mathsf{break}\, b \mapsto \mathsf{abnorm}(\mathsf{break}\, b) \\
\qquad\qquad\qquad\qquad \mid \mathsf{cont}\, c \mapsto \mathsf{abnorm}(\mathsf{cont}\, c)\ \}\ \}
\end{array}$$

The formalisation of creating and catching break and continue abnormalities works similarly, via function CATCH-BREAK and CATCH-CONTINUE.

4 Semantics of method specifications

To start we define two labeled product types incorporating appropriately typed predicates for the various termination modes of statements and expressions.

$$\begin{array}{l}
\mathsf{StatBehaviorSpec}\ :\ \mathsf{TYPE} \stackrel{\text{def}}{=} \\
\quad [\ \mathsf{diverges}\colon \mathsf{OM} \to \mathsf{boolean}, \\
\quad\ \ \mathsf{requires}\colon \mathsf{OM} \to \mathsf{boolean}, \\
\quad\ \ \mathsf{statement}\colon \mathsf{OM} \to \mathsf{StatResult}, \\
\quad\ \ \mathsf{ensures}\colon \mathsf{OM} \to \mathsf{boolean}, \\
\quad\ \ \mathsf{signals}\colon \mathsf{OM} \to \mathsf{RefType} \to \mathsf{boolean}, \\
\quad\ \ \mathsf{return}\colon \mathsf{OM} \to \mathsf{boolean}, \\
\quad\ \ \mathsf{break}\colon \mathsf{OM} \to \mathsf{lift[string]} \to \mathsf{boolean}, \\
\quad\ \ \mathsf{continue}\colon \mathsf{OM} \to \mathsf{lift[string]} \to \mathsf{boolean}\ ]
\end{array}$$

$$\begin{array}{l}
\mathsf{ExprBehaviorSpec}[\alpha]\ :\ \mathsf{TYPE} \stackrel{\text{def}}{=} \\
\quad [\ \mathsf{diverges}\colon \mathsf{OM} \to \mathsf{boolean}, \\
\quad\ \ \mathsf{requires}\colon \mathsf{OM} \to \mathsf{boolean}, \\
\quad\ \ \mathsf{expression}\colon \mathsf{OM} \to \mathsf{ExprResult}[\alpha], \\
\quad\ \ \mathsf{ensures}\colon \mathsf{OM} \to \alpha \to \mathsf{boolean}, \\
\quad\ \ \mathsf{signals}\colon \mathsf{OM} \to \mathsf{RefType} \\
\qquad\qquad\qquad \to \mathsf{boolean}\ ]
\end{array}$$

Notice that the StatBehaviorSpec type has more entries than ExprBehaviorSpec precisely because a statement in Java can terminate abruptly for more reasons than an expression.

There are associated predicates which give the "obvious" meaning.

$$
\begin{array}{l}
\mathsf{SB} \cdot sbs \;:\; \mathsf{boolean} \stackrel{\mathrm{def}}{=} \\
\quad \forall x \in \mathsf{OM}.\; sbs.\mathsf{requires} \cdot x \implies \\
\qquad \mathsf{CASES}\; sbs.\mathsf{statement} \cdot x \;\mathsf{OF}\; \{ \\
\qquad\quad\;\; \mathsf{hang} \mapsto sbs.\mathsf{diverges} \cdot x \\
\qquad\quad \mid \mathsf{norm}\, y \mapsto sbs.\mathsf{ensures} \cdot y \\
\qquad\quad \mid \mathsf{abnorm}\, a \mapsto \mathsf{CASES}\; a \;\mathsf{OF}\; \{ \\
\qquad\qquad\qquad\qquad\;\; \mathsf{excp}\, e \mapsto sbs.\mathsf{signals} \cdot (e.\mathsf{es}) \cdot (e.\mathsf{ex}) \\
\qquad\qquad\qquad\qquad \mid \mathsf{rtrn}\, z \mapsto sbs.\mathsf{return} \cdot z \\
\qquad\qquad\qquad\qquad \mid \mathsf{break}\, b \mapsto sbs.\mathsf{break} \cdot (b.\mathsf{bs}) \cdot (b.\mathsf{blab}) \\
\qquad\qquad\qquad\qquad \mid \mathsf{cont}\, c \mapsto sbs.\mathsf{continue} \cdot (c.\mathsf{cs}) \cdot (c.\mathsf{clab}) \;\}\;\} \\
\mathsf{EB} \cdot ebs \;:\; \mathsf{boolean} \stackrel{\mathrm{def}}{=} \\
\quad \forall x \in \mathsf{OM}.\; ebs.\mathsf{requires} \cdot x \implies \\
\qquad \mathsf{CASES}\; ebs.\mathsf{expression} \cdot x \;\mathsf{OF}\; \{ \\
\qquad\quad\;\; \mathsf{hang} \mapsto ebs.\mathsf{diverges} \cdot x \\
\qquad\quad \mid \mathsf{norm}\, y \mapsto ebs.\mathsf{ensures} \cdot (y.\mathsf{ns}) \cdot (y.\mathsf{res}) \\
\qquad\quad \mid \mathsf{abnorm}\, a \mapsto ebs.\mathsf{signals} \cdot (a.\mathsf{es}) \cdot (a.\mathsf{ex}) \;\}
\end{array}
$$

for sbs: StatBehaviorSpec and ebs: ExprBehaviorSpec[α]. Notice that the diverges predicate is evaluated in the pre-state, in case the statement/expression hangs, because in that case there is simply no post-state. All other predicates are evaluated in the post-state.

The LOOP compiler translates JML method specifications into elements of StatBehaviorSpec and ExprBehaviorSpec, depending on whether the method produces a result or not. The additional entries in StatBehaviorSpec which do not occur in JML specifications (the three last ones) are filled with default values. They may be filled with other values during proofs, typically because of catching of abnormalities, see Subsection 5.4.

For example, consider a JML method specification

```
 /*@ behavior
   @   diverges: d;
   @   requires: p;
   @ modifiable: mod;
   @    ensures: q;
   @    signals: (E e) r;
   @*/
void m() { ... }
```

in a class with invariant I. This specification gets translated (by the LOOP compiler) into:

$$
\begin{aligned}
\forall z\colon \mathsf{OM}.\ \mathsf{SB}\cdot(\ & \mathsf{diverges} = [\![\,\mathtt{d}\,]\!],\\
& \mathsf{requires} = \lambda x\colon \mathsf{OM}.\ [\![\,\mathtt{I}\,]\!]\cdot x \wedge [\![\,\mathtt{p}\,]\!]\cdot x \wedge z = x,\\
& \mathsf{statement} = [\![\,\mathtt{m}\,]\!],\\
& \mathsf{ensures} = \lambda x\colon \mathsf{OM}.\ [\![\,\mathtt{I}\,]\!]\cdot x \wedge [\![\,\mathtt{q}\,]\!]\cdot x\cdot z \wedge z \approx_{\mathtt{mod}} x,\\
& \mathsf{signals} = \lambda x\colon \mathsf{OM}.\,\lambda a\colon \mathsf{RefType}.\ [\![\,\mathtt{I}\,]\!]\cdot x \wedge\\
& \qquad\qquad [\![\,a\ \mathtt{instanceof}\ \mathtt{E}\,]\!] \wedge\\
& \qquad\qquad [\![\,\mathtt{r}\,]\!]\cdot x\cdot z\cdot a \wedge z \approx_{\mathtt{mod}} x,\\
& \mathsf{return} = \lambda x\colon \mathsf{OM}.\,\mathsf{false},\\
& \mathsf{break} = \lambda x\colon \mathsf{OM}.\,\lambda l\colon \mathsf{lift[string]}.\,\mathsf{false},\\
& \mathsf{continue} = \lambda x\colon \mathsf{OM}.\,\lambda l\colon \mathsf{lift[string]}.\,\mathsf{false}\ )
\end{aligned}
$$

The variable z is a logical variable which records the pre-state. It is needed because the normal and exceptional postconditions `q` and `r` may involve an operator `\old(e)`, requiring evaluation of `e` in the pre-state. The term $z \approx_{\mathtt{mod}} x$ is an appropriate translation of the modifiable clause, expressing that x and z are almost the same, except for the fields that are mentioned in the modifiable clause[1].

When translating a `normal_behavior` the diverges and signals fields are set to the constant predicate false; similarly, in an `exceptional_behavior` the diverges and ensures fields become false.

5 Rules for proving method specifications

This section discusses a representative selection of the inference rules that are used for verifying JML method specifications. Some of these rules are bureaucratic, but most of them are "syntax driven". In a goal-oriented view they should be read up-side-down.

5.1 Diverges

Usually, the `diverges` clause in JML is constant, *i.e.* either true or false. Some of the rules below—for example, the composition rule in Figure 3—actually require it to be constant. This can always be enforced via the following rule—at the expense of duplication of the number of proof obligations, see Figure 2.

We illustrate the soundness of this rule. We assume therefore that the assumptions above the line hold. In order to prove the conclusion, we have to distinguish three main cases, for an arbitrary state $x\colon \mathsf{OM}$, satisfying $p\cdot x$:

- $s\cdot x$ hangs. According to the definition of SB, we have to prove $d\cdot x$. But $\neg d\cdot x$, leads to false by the second assumption.

[1] Often it is convenient to weaken the precondition to $\lambda x\colon \mathsf{OM}.\ [\![\,\mathtt{I}\,]\!]\cdot x \wedge [\![\,\mathtt{p}\,]\!]\cdot x \wedge z \approx_{\mathtt{mod}} x$, to obtain a more symmetric correctness formula.

$$\frac{\begin{array}{ll}\mathsf{SB}\cdot(\ \mathsf{diverges} = \lambda x\colon \mathsf{OM}.\,\mathsf{true}, & \mathsf{SB}\cdot(\ \mathsf{diverges} = \lambda x\colon \mathsf{OM}.\,\mathsf{false},\\ \quad\mathsf{requires} = \lambda x\colon \mathsf{OM}.\, p\cdot x \wedge d\cdot x, & \quad\mathsf{requires} = \lambda x\colon \mathsf{OM}.\, p\cdot x \wedge \neg d\cdot x,\\ \quad\mathsf{statement} = s, & \quad\mathsf{statement} = s,\\ \quad\mathsf{ensures} = q, & \quad\mathsf{ensures} = q,\\ \quad\mathsf{signals} = r, & \quad\mathsf{signals} = r,\\ \quad\mathsf{return} = R, & \quad\mathsf{return} = R,\\ \quad\mathsf{break} = B, & \quad\mathsf{break} = B,\\ \quad\mathsf{continue} = C\ ) & \quad\mathsf{continue} = C\ )\end{array}}{\begin{array}{l}\mathsf{SB}\cdot(\ \mathsf{diverges} = d,\\ \quad\mathsf{requires} = p,\\ \quad\mathsf{statement} = s,\\ \quad\mathsf{ensures} = q,\\ \quad\mathsf{signals} = r,\\ \quad\mathsf{return} = R,\\ \quad\mathsf{break} = B,\\ \quad\mathsf{continue} = C\ )\end{array}}$$

Fig. 2. Rule to force diverges predicates to be constant

- $s \cdot x$ terminates normally. The normal postcondition q follows in both cases $d \cdot x$ and $\neg d \cdot x$ from both the assumptions.
- $s \cdot x$ terminates abruptly. Similarly, one gets the appropriate postcondition from both the assumptions.

The soundness of most of the rules below (except for while) is similarly easy. Soundness of all the rules has been proved in PVS.

5.2 Composition

The rule that is most often used is the composition rule. It makes it possible to step through a piece of code by handling single statements one at a time, by introducing appropriate intermediate conditions, namely the p_1 in Figure 3.

A special case of this rule which is often useful in practice has the intermediate condition p_1 of the form $\lambda x\colon \mathsf{OM}.\, p \cdot x \wedge p_2 \cdot x$, where p is the precondition of the goal, and p_2 is an addition to the precondition which holds because of the first statement s_1.

5.3 Return

Recall from Section 3 that the RETURN statement immediately terminates abruptly, by creating a "return" abnormality. The associated rule is much like a skip rule, see Figure 4.

$$
\frac{
\begin{array}{ll}
\begin{array}{l}
\mathsf{SB} \cdot (\ \mathsf{diverges} = \lambda x : \mathsf{OM}.\, b, \\
\qquad \mathsf{requires} = p, \\
\qquad \mathsf{statement} = s_1, \\
\qquad \mathsf{ensures} = p_1, \\
\qquad \mathsf{signals} = r, \\
\qquad \mathsf{return} = R, \\
\qquad \mathsf{break} = B, \\
\qquad \mathsf{continue} = C\ )
\end{array}
&
\begin{array}{l}
\mathsf{SB} \cdot (\ \mathsf{diverges} = \lambda x : \mathsf{OM}.\, b, \\
\qquad \mathsf{requires} = p_1, \\
\qquad \mathsf{statement} = s_2, \\
\qquad \mathsf{ensures} = q, \\
\qquad \mathsf{signals} = r, \\
\qquad \mathsf{return} = R, \\
\qquad \mathsf{break} = B, \\
\qquad \mathsf{continue} = C\ )
\end{array}
\end{array}
}{
\begin{array}{l}
\mathsf{SB} \cdot (\ \mathsf{diverges} = \lambda x : \mathsf{OM}.\, b, \\
\qquad \mathsf{requires} = p, \\
\qquad \mathsf{statement} = s_1\, ;\, s_2, \\
\qquad \mathsf{ensures} = q, \\
\qquad \mathsf{signals} = r, \\
\qquad \mathsf{return} = R, \\
\qquad \mathsf{break} = B, \\
\qquad \mathsf{continue} = C\ )
\end{array}
}
$$

Fig. 3. Composition rule

5.4 Catching returns

Recall that the LOOP compiler wraps a CATCH-RETURN function around each translated method body, in order to turn possible return abnormalities into normal termination. The associated rule in Figure 4 therefore puts the normal postcondition of the goal into the return position.

Notice that via a rule like this an entry which is not used in JML specifications (namely return) can get a non-default value during proofs. This is the reason for including such additional entries in the definition of the type StatBehaviorSpec in Section 4.

5.5 If-then-else

Java has the `if-then` and `if-then-else` conditional statements. We only describe the relevant rule for the latter, see Figure 5. It deals with the possible side-effect and with the result of the condition c via the intermediate predicate qc.

5.6 While

In a final rule, we consider Java's `while(c){s}` statement. It involves a condition `c` and a statement `s` which is iterated until the condition becomes false, or a form of abrupt termination arises. Especially, a `break` or `continue` statement, possibly with a label, may be used within a `while` statement (to jump out of the loop, or to jump to the next cycle). We refer to [10] for a detailed description of

$$
\frac{\forall x\colon \mathsf{OM}.\ p \cdot x \implies R \cdot x}{\begin{array}{l}\mathsf{SB} \cdot (\ \mathsf{diverges} = d,\\ \quad \mathsf{requires} = p,\\ \quad \mathsf{statement} = \mathsf{RETURN},\\ \quad \mathsf{ensures} = q,\\ \quad \mathsf{signals} = r,\\ \quad \mathsf{return} = R,\\ \quad \mathsf{break} = B,\\ \quad \mathsf{continue} = C\ )\end{array}}
\qquad
\frac{\begin{array}{l}\mathsf{SB} \cdot (\ \mathsf{diverges} = d,\\ \quad \mathsf{requires} = p,\\ \quad \mathsf{statement} = s,\\ \quad \mathsf{ensures} = q,\\ \quad \mathsf{signals} = r,\\ \quad \mathsf{return} = q,\\ \quad \mathsf{break} = B,\\ \quad \mathsf{continue} = C\ )\end{array}}{\begin{array}{l}\mathsf{SB} \cdot (\ \mathsf{diverges} = d,\\ \quad \mathsf{requires} = p,\\ \quad \mathsf{statement} = \mathsf{CATCH\text{-}RETURN} \cdot s,\\ \quad \mathsf{ensures} = q,\\ \quad \mathsf{signals} = r,\\ \quad \mathsf{return} = R,\\ \quad \mathsf{break} = B,\\ \quad \mathsf{continue} = C\ )\end{array}}
$$

Fig. 4. Rules for the return and catch-return statements

$$
\frac{\begin{array}{l}\mathsf{EB} \cdot (\ \mathsf{diverges} = \lambda x\colon \mathsf{OM}.\, b,\\ \quad \mathsf{requires} = p,\\ \quad \mathsf{expression} = c,\\ \quad \mathsf{ensures} = qc,\\ \quad \mathsf{signals} = r\ )\end{array}
\quad
\begin{array}{l}\mathsf{SB} \cdot (\ \mathsf{diverges} = \lambda x\colon \mathsf{OM}.\, b,\\ \quad \mathsf{requires} = \lambda x\colon \mathsf{OM}.\\ \qquad qc \cdot x \cdot \mathsf{true}\\ \quad \mathsf{statement} = s_1,\\ \quad \mathsf{ensures} = q,\\ \quad \mathsf{signals} = r,\\ \quad \mathsf{return} = R,\\ \quad \mathsf{break} = B,\\ \quad \mathsf{continue} = C\ )\end{array}
\quad
\begin{array}{l}\mathsf{SB} \cdot (\ \mathsf{diverges} = \lambda x\colon \mathsf{OM}.\, b,\\ \quad \mathsf{requires} = \lambda x\colon \mathsf{OM}.\\ \qquad qc \cdot x \cdot \mathsf{false}\\ \quad \mathsf{statement} = s_2,\\ \quad \mathsf{ensures} = q,\\ \quad \mathsf{signals} = r,\\ \quad \mathsf{return} = R,\\ \quad \mathsf{break} = B,\\ \quad \mathsf{continue} = C\ )\end{array}}{\begin{array}{l}\mathsf{SB} \cdot (\ \mathsf{diverges} = \lambda x\colon \mathsf{OM}.\, b,\\ \quad \mathsf{requires} = p,\\ \quad \mathsf{statement} = \mathsf{IF\text{-}THEN\text{-}ELSE} \cdot c \cdot s_1 \cdot s_2,\\ \quad \mathsf{ensures} = q,\\ \quad \mathsf{signals} = r,\\ \quad \mathsf{return} = R,\\ \quad \mathsf{break} = B,\\ \quad \mathsf{continue} = C\ )\end{array}}
$$

Fig. 5. Rule for if-then-else

the formalisation of the `while` statement, and restrict ourselves to the relevant rule, see Figure 6.

$$\frac{\begin{array}{ll} \mathsf{EB}\cdot(\,\mathsf{diverges} = \lambda x\colon \mathsf{OM}.\, b, & \mathsf{SB}\cdot(\,\mathsf{diverges} = \lambda x\colon \mathsf{OM}.\, b, \\ \quad \mathsf{requires} = \lambda x\colon \mathsf{OM}. & \quad \mathsf{requires} = \lambda x\colon \mathsf{OM}.\, qc\cdot x \wedge \\ \quad\quad \mathsf{invariant}\cdot x \wedge & \quad\quad \mathsf{variant}\cdot x = a_2, \\ \quad\quad \mathsf{variant}\cdot x = a_1, & \quad \mathsf{statement} = \mathsf{CATCH\text{-}CONTINUE}\cdot \ell \cdot s, \\ \quad \mathsf{expression} = c, & \quad \mathsf{ensures} = \lambda x\colon \mathsf{OM}. \\ \quad \mathsf{ensures} = \lambda x\colon \mathsf{OM}.\, \lambda b\colon \mathsf{bool}. & \quad\quad \mathsf{invariant}\cdot x \wedge \\ \quad\quad \mathsf{IF}\ b & \quad\quad \mathsf{variant}\cdot x < a_1, \\ \quad\quad \mathsf{THEN}\ qc\cdot x \wedge \mathsf{variant}\cdot x = a_2 & \quad \mathsf{signals} = r, \\ \quad\quad \mathsf{ELSE}\ q\cdot x, & \quad \mathsf{return} = R, \\ & \quad \mathsf{break} = B, \\ \quad \mathsf{signals} = r\,) & \quad \mathsf{continue} = C\,) \end{array}}{\begin{array}{l} \mathsf{SB}\cdot(\,\mathsf{diverges} = \lambda x\colon \mathsf{OM}.\, b, \\ \quad \mathsf{requires} = \mathsf{invariant}, \\ \quad \mathsf{statement} = \mathsf{WHILE}\cdot \ell \cdot c \cdot s, \\ \quad \mathsf{ensures} = q, \\ \quad \mathsf{signals} = r, \\ \quad \mathsf{return} = R, \\ \quad \mathsf{break} = B, \\ \quad \mathsf{continue} = C\,) \end{array}}$$

Fig. 6. Rule for total reasoning with while

The parameter $\ell\colon \mathsf{lift}[\mathsf{string}]$ in the goal statement $\mathsf{WHILE}\cdot\ell\cdot c\cdot s$ is $\mathsf{up}(\text{“}\mathtt{lab}\text{”})$ if there is label `lab` immediately before the while statement in Java, and bot otherwise. If a statement `continue` or `continue lab` is executed within the loop body s, the resulting “continue” abnormality is caught by the wrapper $\mathsf{CATCH\text{-}CONTINUE}\cdot\ell\cdot s$, so that the next cycle can start normally. The LOOP tool puts a CATCH-BREAK function around every while statement, in order to catch any breaks within this statement[2]. The variant is a function $\mathsf{OM} \to A$ to some well-founded order A, which is required to decrease with every normally executed cycle[3]. Notice how an auxiliary predicate qc and values $a_1, a_2 \in A$ are used to pass on the effect of the condition to the statement—in the case where the condition evaluates to true. In this way the variant can decrease during execution of either the condition c or the statement s.

[2] The effect of these CATCH-BREAK and CATCH-CONTINUE functions can be incorporated into the while rule in Figure 6, by adapting the break and continue predicates in the assumptions, but this complicates this rule even further.

[3] Note that requiring the existence of the variant restricts the use of this rule to terminating while loops. Therefore, this “total” while rule only really make sense when the divergence clause is constantly false.

6 Example verification in PVS

The rules from the previous section (plus some more rules) have all been formulated in PVS, and proven correct. This makes it possible to use these rules to prove that Java methods meet their JML specifications in PVS. The translations of these specifications are Boolean expressions of the form $\mathsf{SB} \cdot (\, \mathsf{diverges} = d, \cdots)$ or $\mathsf{EB} \cdot (\, \mathsf{diverges} = d, \cdots)$ involving suitable labeled tuples. These tuples can become very big during proofs, but the explicit labels keep them reasonably well-structured and manageable. The proof rules allow us to rewrite these labeled tuples into adapted tuples, following the structure of the Java code (of the body of the method whose correctness should be proved). This rewriting is continued until the $\mathsf{statement}$ or $\mathsf{expression}$ in the labeled tuple is sufficiently simple to proceed with the proof at a purely semantical level (like in the rule for RETURN in Subsection 5.3).

In this way the example JML specification of the pattern-search from Figure 1 has been proved for the given Java implementation. The latter involves `return` statements inside a `while` loop, leading to abrupt termination and a break out of the loop, both when it becomes clear that the pattern is present and that it is absent. This presents a non-trivial verification challenge, not only because of these `return` statements but also because of the non-trivial (in)variant involved, see [10]. The proof makes essential use of the rule for while (once) and for if-then-else (three times), and also for composition (several times), following the structure of the Java code.

The same example has been used in [10], where it was verified with the special Hoare logic (from [10]) with separate triples for the different termination modes in Java. It is re-used here to enable a comparison. Such a comparison is slightly tricky because when the proof was re-done with the proof rules for JML, both the variant and invariant were already known. Also, no time had to be spent on formulating the required correctness property in PVS, because this could all be done (more conveniently) in JML. Taking this into account, the new rules still give a considerable speed-up of the proof. The verification is no longer a matter of days, but has become a matter of hours. The main reason is that the correctness formulas in the new logic for JML combine all termination modes in a single formula, and thus requires only one rule per language construct, with fewer assumptions.

7 Conclusion

In this paper JML method specification have been transformed into correctness formulas in an associated logic. These formulas extend standard Hoare triples (and those from [10]) by combining all possible termination modes for Java, naturally following the (coalgebraic) representation of statements and expressions. The correctness formulas capture all essential ingredients for an axiomatic semantics for Java. In combination with the underlying low-level, memory-based semantics of Java, these rules for JML provide an efficient, powerful and flexible setting for tool-assisted verification of Java programs with JML annotations.

Acknowledgements

Thanks are due to Joachim van den Berg and Marieke Huisman for discussing various aspects of the rules for JML.

References

1. JavaCard API 2.1. `http://java.sun.com/products/javacard/htmldoc/`.
2. P. America. Designing an object-oriented language with behavioural subtyping. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, number 489 in Lect. Notes Comp. Sci., pages 60–90. Springer, Berlin, 1990.
3. K.R. Apt. Ten years of Hoare's logic: A survey—part I. *ACM Trans. on Progr. Lang. and Systems*, 3(4):431–483, 1981.
4. 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.
5. J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. Techn. Rep. CSI-R0019, Comput. Sci. Inst., Univ. of Nijmegen. To appear at TACAS'01., 2000.
6. A. Bhorkar. A run-time assertion checker for Java using JML. Techn. Rep. 00-08, Dep. of Comp. Science, Iowa State Univ. (`http://www.cs.iastate.edu/~leavens/JML.html`), 2000.
7. F.S. de Boer. A WP-calculus for OO. In W. Thomas, editor, *Foundations of Software Science and Computation Structures*, number 1578 in Lect. Notes Comp. Sci., pages 135–149. Springer, Berlin, 1999.
8. M. Huisman. *Reasoning about JAVA Programs in higher order logic with PVS and Isabelle*. PhD thesis, Univ. Nijmegen, 2001.
9. 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.
10. 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.
11. M. Huisman, B. Jacobs, and J. van den Berg. A case study in class library verification: Java's Vector class. Techn. Rep. CSI-R0007, Comput. Sci. Inst., Univ. of Nijmegen. To appear in *Software Tools for Technology Transfer*, 2001.
12. B. Jacobs. A formalisation of Java's exception mechanism. Techn. Rep. CSI-R0015, Comput. Sci. Inst., Univ. of Nijmegen. To appear at ESOP'01., 2000.
13. 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.
14. 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.

15. G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Techn. Rep. 98-06, Dep. of Comp. Sci., Iowa State Univ. (`http://www.cs.iastate.edu/~leavens/JML.html`), 1999.
16. K.R.M. Leino. *Toward Reliable Modular Programs.* PhD thesis, California Inst. of Techn., 1995.
17. K.R.M. Leino, J.B. Saxe, and R. Stata. Checking java programs via guarded commands. In B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs. Proceedings of the ECOOP'99 Workshop.* Techn. Rep. 251, Fernuniversität Hagen, 1999. Also as Technical Note 1999-002, Compaq Systems Research Center, Palo Alto.
18. B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 2^{nd} rev. edition, 1997.
19. D. von Oheimb. Axiomatic semantics for Java^{light} in Isabelle/HOL. Technical Report CSE 00-009, Oregon Graduate Inst., 2000. TPHOLS 2000 Supplemental Proceedings.
20. 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.
21. L.C. Paulson. *Isabelle: A Generic Theorem Prover.* Number 828 in Lect. Notes Comp. Sci. Springer, Berlin, 1994.
22. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S.D. Swierstra, editor, *Programming Languages and Systems*, number 1576 in Lect. Notes Comp. Sci., pages 162–176. Springer, Berlin, 1999.
23. E. Poll. A coalgebraic semantics of subtyping. In H. Reichel, editor, *Coalgebraic Methods in Computer Science*, number 33 in Elect. Notes in Theor. Comp. Sci. Elsevier, Amsterdam, 2000.
24. E. Poll, J. van den Berg, and B. Jacobs. Specification of the JavaCard API in JML. In J. Domingo-Ferrer, D. Chan, and A. Watson, editors, *Smart Card Research and Advanced Application*, pages 135–154. Kluwer Acad. Publ., 2000.
25. E. Poll, J. van den Berg, and B. Jacobs. Formal specification of the JavaCard API in JML: the APDU class. *Comp. Networks Mag.*, 2001. To appear.
26. Loop Project. `http://www.cs.kun.nl/~bart/LOOP/`.
27. Extended static checker ESC/Java. Compaq System Research Center. `http://www.research.digital.com/SRC/esc/Esc.html`.