The version of the following full text has not yet been defined or was untraceable and may differ from the publisher's version.

For additional information about this publication click this link.  
http://hdl.handle.net/2066/18939

Please be advised that this information was generated on 2019-08-24 and may be subject to change.
A Logic for the Java Modeling Language JML

E.Poll, B.P.F. Jacobs

Computing Science Institute/

CSI-R0018 November 2000
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) [14,13] 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 [17]. JML has been integrated with the specification language used for E$\cdot$SC/Java, the extended static checker developed at Compaq System Research Center [16,25].

At Nijmegen, a formal semantics has been developed for essentially all of sequential Java. A compiler has been built, the LOOP tool, which translates a Java program into logical theories describing its semantics [12,4,9,8,11,7,24]. 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 [19] and Isabelle [20]. 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 [10]. The current main application area is JavaCard [1], see [23]. 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 it becomes possible to provide tool support. Current work in this direction for JML focuses on the generation of run-time checks on preconditions for testing, at Iowa State University [5] 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) [9] 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 [13]. 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 distinction. The soundness of the while rule, see Subsection 5.7, 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 [21,18]. We do not need these rules because
we can always fall back on our low level semantics where these issues are handled automatically [8]. 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 [3,15,6,21]—but on the translated Java code in the back-end theorem prover. 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.

This combination of high level proof rules and low level automatic rewriting on the basis of the underlying semantics forms the strength of our approach. It means that we do not have to care about completeness of our higher level rules (like in [18]), because we can always fall back on the semantical level (technically, by expanding the definition of the correctness formulas SB and EB from Section 4 below). We only introduce logical rules when this really suits us, in order to achieve a higher level of abstraction in proofs. This pragmatic attitude allows us to avoid some of the complicated rules found in Hoare logics for object-oriented languages.

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 [22]) 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 [9]—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 [14,13]. 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 the 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.

```java
/**@ behavior
  @ diverges: <pre-condition for non-termination>
  @ requires: <precondition>
  @ modifiable: <fields 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) de­
scribes a situation where a method must terminate normally (or exceptionally),
assuming that the precondition holds. For example, the class below describes
an annotated method (from [9]) that searches for a certain pattern in an array
using a single while loop. It has a non-trivial postcondition.

## 3 Java statements and expressions in higher order logic

This section introduces the (coalgebraic) representation 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 ex­
pressions act, as functions OM → StatResult and OM → ExprResult[\alpha],
for a
suitable result type \alpha. These result types are introduced as labeled coproduct
(also called variant or sum) types:

```
StatResult : TYPE def = { hang: unit | norm: OM | abnorm: StatAbn }
```

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;
    int [] 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
    @     \imp pattern[i] == base[\result+i]) &&
    @ /// \pattern does not occur earlier
    @ (forall (int j)
    @     0 <= j && j < \result
    @     \imp (\exists (int i)
    @         0 <= i && i < pattern.length
    @         \imp pattern[i] != base[j+i])))
    @ ||
    @ ( //\ pattern does not occur
    @ \result == -1 &&
    @ (forall (int j)
    @     0 <= j && j < base.length - pattern.length
    @     \imp (\exists (int i)
    @         0 <= i && i < pattern.length
    @         \imp 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 \[ns: OM, res: \alpha\]. The result types StatAbn and ExprAbn for abrupt
termination are subdivided differently for statements and expressions:

\[
\text{StatAbn} : \text{TYPE} \overset{\text{def}}{=} \\
\{ \text{excp: [es: OM, ex: RefType]} \}
\]

\[
\text{ExprAbn} : \text{TYPE} \overset{\text{def}}{=} \\
\{ \text{es: OM, ex: RefType} \}
\]

The type \text{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 \text{lift} type constructor adds a bottom element \text{bot} to an arbitrary type, and keeps all original elements \text{a} as \text{up a}. It is used because \text{break} and \text{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 LOO P tool). For instance, the composition of two statements is defined as:

\[
s, t : \text{OM} \rightarrow \text{StatResult} \vdash \\
(s ; t) : \text{OM} \rightarrow \text{StatResult} \overset{\text{def}}{=} \\
\lambda x : \text{OM}. \text{CASES } s \cdot x \text{ OF } \\
\mid \text{hang} \mapsto \text{hang} \\
\mid \text{norm } y \mapsto t \cdot y \\
\mid \text{abnorm } a \mapsto \text{abnorm } a
\]

where \text{·} is used for function application, and \text{CASES} for pattern matching on the labels of the \text{StatResult} coproduct type. What is important to note is that if the statement \text{s} hangs or terminates abruptly, then so does the composition \text{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 \text{E2S}, defined as:

\[
e : \text{OM} \rightarrow \text{ExprResult}[\alpha] \vdash \\
\text{E2S} \cdot e : \text{OM} \rightarrow \text{StatResult} \overset{\text{def}}{=} \\
\lambda x : \text{OM}. \text{CASES } e \cdot x \text{ OF } \\
\mid \text{hang} \mapsto \text{hang} \\
\mid \text{norm } y \mapsto \text{norm}(y . n s) \\
\mid \text{abnorm } a \mapsto \text{abnorm}(\text{excp}(\text{es} = a . e s, \text{ex} = a . e x))
\]

where the notation \text{y.ns} describes field selection associated with \text{y} in the labeled product \text{[ns: OM, res: a]}. In the last line an expression abnormality (an exception) is transformed into a statement abnormality. Java’s \text{if-then-else}
becomes:

\[
\begin{align*}
\text{c}: \text{OM} & \rightarrow \text{ExprResult}[\text{bool}], \ s, t: \text{OM} \rightarrow \text{StatResult} \\
\text{IF-THEN-ELSE} \cdot c \cdot s \cdot t: \text{OM} \rightarrow \text{StatResult} & \overset{\text{def}}{=} \\
\lambda x: \text{OM}. \ \text{CASES} \ c \cdot x \ \text{OF} \{ \\
\ & \text{hang} \mapsto \text{hang} \\
\ & \text{norm} \ y \mapsto \text{IF} \ y.\text{res} \ \text{THEN} \ s \cdot (y.\text{ns}) \ \text{ELSE} \ t \cdot (y.\text{ns}) \\
\ & \text{abnorm} \ a \mapsto \text{abnorm}(\text{excp}(\text{es} = a.\text{es}, \text{ex} = a.\text{ex})) \} \\
\end{align*}
\]

The formalisation of Java’s \texttt{return} statement (without argument) is:

\[
\text{RETURN} : \text{OM} \rightarrow \text{StatResult} \overset{\text{def}}{=} \lambda x: \text{OM}. \ \text{abnorm}(\text{rtrn} \ x)
\]

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 \texttt{CATCH-RETURN} is wrapped around every method body that returns \texttt{void}. First the method body is executed. This may result in an abnormal state, because of a return. In that case the function \texttt{CATCH-RETURN} turns the state back to normal again. Otherwise, it leaves everything unchanged.

\[
\begin{align*}
\text{s}: \text{OM} & \rightarrow \text{StatResult}[\text{OM}] \\
\text{CATCH-RETURN} \cdot s: \text{OM} \rightarrow \text{StatResult}[\text{OM}] & \overset{\text{def}}{=} \\
\lambda x: \text{OM}. \ \text{CASES} \ s \cdot x \ \text{OF} \{ \\
\ & \text{hang} \mapsto \text{hang} \\
\ & \text{norm} \ y \mapsto \text{norm} \ y \\
\ & \text{abnorm} \ a \mapsto \text{CASES} \ a \ \text{OF} \{ \\
\ & \text{excp} \ e \mapsto \text{abnorm}(\text{excp} \ e) \\
\ & \text{rtrn} \ z \mapsto \text{norm} \ z \\
\ & \text{break} \ b \mapsto \text{abnorm}(\text{break} \ b) \\
\ & \text{cont} \ c \mapsto \text{abnorm}(\text{cont} \ c) \} \}
\end{align*}
\]

The formalisation of creating and catching break and continue abnormalities works similarly, via function \texttt{CATCH-BREAK} and \texttt{CATCH-CONTINUE}. 7
4 Method specifications in type theory

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

\[
\text{StatBehaviorSpec} : \text{TYPE} \overset{\text{def}}{=} \left[ \begin{array}{l}
\text{diverges} : \text{OM} \to \text{boolean}, \\
\text{requires} : \text{OM} \to \text{boolean}, \\
\text{statement} : \text{OM} \to \text{StatResult}, \\
\text{ensures} : \text{OM} \to \text{boolean}, \\
\text{signals} : \text{OM} \to \text{RefType} \to \text{boolean}, \\
\text{return} : \text{OM} \to \text{boolean}, \\
\text{break} : \text{OM} \to \text{lift}[^\text{string}] \to \text{boolean}, \\
\text{continue} : \text{OM} \to \text{lift}[^\text{string}] \to \text{boolean}
\end{array} \right]
\]

\[
\text{ExprBehaviorSpec}[\alpha] : \text{TYPE} \overset{\text{def}}{=} \left[ \begin{array}{l}
\text{diverges} : \text{OM} \to \text{boolean}, \\
\text{requires} : \text{OM} \to \text{boolean}, \\
\text{expression} : \text{OM} \to \text{ExprResult}[\alpha], \\
\text{ensures} : \text{OM} \to \alpha \to \text{boolean}, \\
\text{signals} : \text{OM} \to \text{RefType} \to \text{boolean}
\end{array} \right]
\]

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.

\[
v \circ \text{StatBehaviorSpec} \vDash \]

\[
\text{SB} \cdot v : \text{boolean} \overset{\text{def}}{=} \\
\forall x \in \text{OM}. \ x.\text{requires} \rightarrow x \implies \text{CASES } x \circ \text{statement} \circ x \text{ OF } \{
\begin{array}{l}
| \text{hang} \rightarrow v.\text{diverges} \circ x \\
| \text{norm } y \rightarrow v.\text{ensures} \circ y \\
| \text{abnorm } a \rightarrow \text{CASES } a \circ \text{expression} \circ (f) \text{ OF } \{
\begin{array}{l}
| \text{excp } c \rightarrow v.\text{signals} \circ (c).\text{es} \circ (c).\text{ex} \\
| \text{rtm } z \rightarrow v.\text{return} \circ z \\
| \text{break } b \rightarrow v.\text{break} \circ (b).\text{bs} \circ (b).\text{blab} \\
| \text{cont } c \rightarrow v.\text{continue} \circ (c).\text{cs} \circ (c).\text{clab}\}
\end{array}\}
\}
\]

\[
v \circ \text{ExprBehaviorSpec}[\alpha] \vDash \]

\[
\text{EB} \cdot v : \text{boolean} \overset{\text{def}}{=} \\
\forall x \in \text{OM}. \ x.\text{requires} \rightarrow x \implies \text{CASES } x \circ \text{expression} \circ x \text{ OF } \{
\begin{array}{l}
| \text{hang} \rightarrow v.\text{diverges} \circ x \\
| \text{norm } y \rightarrow v.\text{ensures} \circ (y).\text{ns} \circ (y).\text{res} \\
| \text{abnorm } a \rightarrow v.\text{signals} \circ (a).\text{es} \circ (a).\text{ex}\}
\}
\]

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 5.5.

For example, consider a JML method specification

```java
/*@ 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:

```
∀z: OM. SB • ( diverges = \[[d]\],
  requires = λx: OM. \[[I]\] • x ∧ \[[p]\] • x ∧ (z = x),
  statement = \[[m]\],
  ensures = λx: OM. \[[I]\] • x ∧ \[[q]\] • x ∧ (z ≈_{mod} x),
  signals = λx: OM. λa: RefType. \[[I]\] • x ∧
    \[[a instanceof E]\] ∧
    \[[r]\] • x • z • a ∧ (z ≈_{mod} x),
  return = λx: OM. false,
  break = λx: OM. λI: lift[string].false,
  continue = λx: OM. λI: lift[string].false )
```

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 \void(e), requiring evaluation of e in the pre-state. The term z ≈_{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.

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 rules that are used for verifying JML method specifications. Some of these rules are bureaucratic, but most of them are “syntax driven”. The rules are presented in natural deduction style. 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.

```
SB • (diverges = \lambda x: OM. true, requires = \lambda x: OM. p \cdot x \land d \cdot x, statement = s, ensures = q, signals = r, return = R, break = B, continue = C)  
```

```
SB • (diverges = \lambda x: OM. false, requires = \lambda x: OM. p \cdot x \land \neg d \cdot x, statement = s, ensures = q, signals = r, return = R, break = B, continue = C)  
```

```
SB • (diverges = d, requires = p, statement = s, ensures = q, signals = r, return = R, break = B, continue = C)  
```

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

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 \( x : OM \), 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.
- \( s \cdot x \) 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 Modifiable

The translation of JML method specifications at the end of Section 4 involves an equation \( z = x \) in the precondition and a modifiable relation \( z \approx_{\text{mod}} x \) in the
normal and exceptional postconditions. During actual proofs this is not conve-
nient, because it forces us to keep track of all effects on the state space in the
equation \( z = x \), as \( f(z) = x \) where \( f \) describes this effect. The trick to avoid this
is to replace \( z = x \) in the precondition by \( z \approx_{\text{mod}} x \), giving a more symmetric
correctness formula. This replacement imposes two proof obligations:

- \( z \approx_{\text{mod}} x \), saying that \( x \) is almost the same as \( x \); this should obviously hold
  for any modifiable clause.
- \( z \approx_{\text{mod}} x \) and \( p \cdot z \) implies \( p \cdot x \), saying that the possible modifications do not
  affect the precondition. Again this should be easy to prove.

5.3 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.

\[
\begin{align*}
\text{SB} \cdot ( \text{diverges} = \lambda x: \text{OM}. b, & \quad \text{SB} \cdot ( \text{diverges} = \lambda x: \text{OM}. b, \\
\text{requires} = p, & \quad \text{requires} = p_1, \\
\text{statement} = s_1, & \quad \text{statement} = s_2, \\
\text{ensures} = p_1, & \quad \text{ensures} = q, \\
\text{signals} = r, & \quad \text{signals} = r, \\
\text{return} = R, & \quad \text{return} = R, \\
\text{break} = B, & \quad \text{break} = B, \\
\text{continue} = C ) & \quad \text{continue} = C )
\end{align*}
\]

Fig. 3. Composition rule

A special case of this rule which is often useful in practice has the intermediate
condition \( p_1 \) of the form \( \lambda x: \text{OM}. p \cdot x \land p_2 \cdot x \), where \( p \) is the precondition of the
goal, and \( p_2 \) is an addition to the precondition which holds because of the first
statement \( s_1 \).
5.4 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.

\[ \forall x : OM. \; p \cdot x \implies R \cdot x \]

\[ SB \cdot ( \text{diverges} = d, \]
\[ \text{requires} = p, \]
\[ \text{statement} = \text{RETURN}, \]
\[ \text{ensures} = q, \]
\[ \text{signals} = r, \]
\[ \text{return} = R, \]
\[ \text{break} = B, \]
\[ \text{continue} = C ) \]

Fig. 4. Rule for the return statement

5.5 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 5 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.

5.6 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 6. It explicitly takes the possible side-effect of the condition into account via the E2S function from Section 3.

5.7 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 [9] for a detailed description of
SB • ( diverges = d,
  requires = p,
  statement = s,
  ensures = q,
  signals = r,
  return = q,
  break = B,
  continue = C )

SB • ( diverges = d,
  requires = p,
  statement = CATCH-RETURN • s,
  ensures = q,
  signals = r,
  return = Ä,
  break = B,
  continue = C )

\text{Fig. 5. Rule for catch-return}

SB • ( diverges = d,
  requires = \lambda x : OM.p \land x
  CASES c : x OF {
    | hang \mapsto \text{true}
    | norm y \mapsto y.\text{res}
    | abnorm a \mapsto \text{true} },
  statement = (E2S c) ; s_1,
  ensures = q,
  signals = r,
  return = R,
  break = B,
  continue = C )

SB • ( diverges = d,
  requires = \lambda x : OM.p \land x
  CASES c : x OF {
    | hang \mapsto \text{true}
    | norm y \mapsto \neg y.\text{res}
    | abnorm a \mapsto \text{true} },
  statement = (E2S c) ; s_2,
  ensures = q,
  signals = r,
  return = R,
  break = B,
  continue = C )

\text{Fig. 6. Rule for if-then-else}

13
Fig. 7. Rule for total reasoning with while

the type theoretic formalisation of the while statement, and restrict ourselves to the relevant rule, see Figure 7.

The parameter \( \ell \) : \text{lift}[\text{string}] in the goal statement \( \text{WHILE} \cdot \ell \cdot c \cdot s \) is up(“lab”) if there is label \text{lab} immediately before the while statement in Java, and \text{bot} otherwise. If a statement \text{continue} or \text{continue} \text{lab} is executed within the loop body \( s \), the resulting “continue” abnormality is caught by the wrapper \text{CATCH-CONTINUE} : \ell \cdot s \), so that the next cycle can start normally. The LOOP tool puts a \text{CATCH-BREAK} function around every while statement, in order to catch any breaks within this statement\(^1\). The variant is a function OM \( \rightarrow A \) to some well-founded order \( A \), which is required to decrease with every normally

\(^1\) The effect of these \text{CATCH-BREAK} and \text{CATCH-CONTINUE} functions can be incorporated into the while rule in Figure 7, by adapting the break and continue predicates in the assumptions, but this complicates this rule even further.
executed cycle\textsuperscript{2}. The soundness proof for this rule involves subtle reasoning about chains in $A$.

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 in proofs in PVS of JML specifications. The translations of these specifications are Boolean expressions of the form $SB \cdot (\text{diverges} = d, \cdots)$ or $EB \cdot (\text{diverges} = d, \cdots)$ involving suitable labeled tuples. These tuples can become really 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 statement or expression in the labeled tuple is sufficiently simple to proceed with the proof at a purely semantical level (like in the rule for \texttt{return} in Subsection 5.4).

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 \texttt{return} statements inside a \texttt{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 \texttt{return} statements but also because of the non-trivial (in)variant involved. The proof makes essential use of the rule for \texttt{while (once)} and for \texttt{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 [9], where it was verified with the special Hoare logic (from [9]) 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

\textsuperscript{2} 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 \texttt{diverges} clause is constantly false.
(and those from [9]) 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


