Int J STTT (2001) 3: 332–352 / Digital Object Identifier (DOI) 10.1007/s100090100047

A case study in class library verification: Java's vector class

Marieke Huisman, Bart Jacobs, Joachim van den Berg

Computing Science Institute, University of Nijmegen, Toernooiveld 1, 6525 Nijmegen, The Netherlands;
E-mail: {marieke,bart,joachim}@cs.kun.nl

Published online: 10 May 2001 – © Springer-Verlag 2001

Abstract. This paper presents a verification of an invariant property for the `Vector` class from JAVA's standard library (API). The property says (essentially) that the actual size of a vector is less than or equal to its capacity. It is shown that this "safety" or "data integrity" property is maintained by all methods of the `Vector` class, and that it holds for all objects created by the constructors of the `Vector` class. The verification of the `Vector` class invariant is done with the proof tool PVS. It is based on a semantics of JAVA in higher order logic. The latter is incorporated in a special purpose compiler, the LOOP tool, which translates JAVA classes into logical theories. It has been applied to the `Vector` class for this case study. The actual verification takes into account the object-oriented character of JAVA: (non-final) methods may always be overridden, so that one cannot rely on a particular implementation. Instead, one has to reason from method specifications in such cases. This project demonstrates the feasibility of tool-assisted verification of non-trivial properties for non-trivial JAVA classes.

Keywords: Java – Invariant – Program verification – Specification

1 Introduction

One of the reasons for the popularity of object-oriented programming is the possibility it offers for reuse of code. Usually, the distribution of an object-oriented programming language comes together with a collection of ready-to-use classes, in a class library or API (application program interface). Typically, these classes contain general purpose code, which can be used as basis for many applications. Before using such classes, a programmer usually wants to know how they behave and when their methods terminate normally or throw exceptions. One way to do this is to study the actual code. This is time-consuming and requires understanding all the particular ins and outs of the implementation – which may even be absent, for native methods. Hence, this is often not the most efficient way. Another approach is to study the (informal) documentation provided. As long as this documentation is clear and concise, this works well, but otherwise a programmer is still is forced to look at the actual code.

One way to improve this situation is to formally specify suitable properties of standard classes, and add these specifications as annotations to the documentation. Examples of properties that can be specified are termination conditions (in which cases will a method terminate normally? In which cases will it throw an exception?), pre-post-condition relations, and class invariants. Once sufficiently many properties have been specified, one only has to understand these properties; and then there is no need anymore to study the actual code, in order to be able to use a class safely.

Programmers must of course be able to rely on such specifications. This introduces the obligation to actually verify that the specified properties hold for the implementation. Even stronger, specifications can exist independently of implementations, as so-called interface specifications. As such they may describe library classes in a component-oriented approach, based on interface specifications regulating the interaction between components. In such a "design by contract" scenario the provider of a class implementation has the obligation to show that the specification is met. Naturally, every next version of the implementation should still satisfy the specification, ensuring proper upgrading.

Thus, verification of class specifications is an important issue. This paper describes a case study verification of one particular library class, namely `Vector`, which is in the standard JAVA distribution [5, 10, 12]. An object belonging to the `Vector` class basically consists of an array of "element" objects. At run-time, according to needs,

Correspondence to: Bart Jacobs

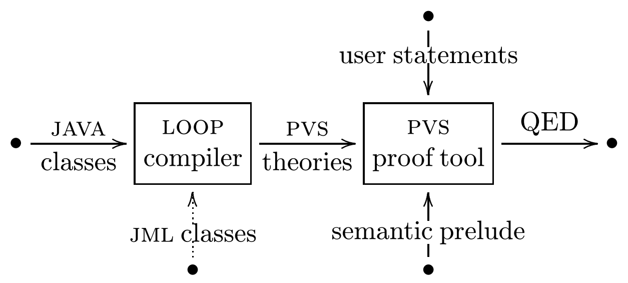
this array may be replaced by an array of a different size (but containing the same elements)[1]. The essence of the `Vector` invariant that is proven is that the size of a vector never exceeds the length of this internal array. Clearly, this is a crucial safety property.

The choice for the `Vector` class in this verification is in fact rather arbitrary: it serves our purposes well because it involves a non-trivial amount of code (including the code from its surrounding classes from the library), and gives rise to an interesting invariant. However, other classes than `Vector` could have been verified. In fact, there are many classes in the JAVA API, like `StringBuffer` using an array of characters with a count, for which a similar invariant can be formulated. Thus, the property that we consider is fairly typical as a class invariant.

1.1 Languages and tools

For the specification of the `Vector` invariant (and many pre- and post-conditions) we make use of the experimental behavioural interface specification language JML (short for JAVA modeling language) [25] (see [43]). Its syntax is much like JAVA's, and mostly self-explanatory. JML is also used for a follow-up specification and verification project focussing on the entire JAVACARD API [37] (which is much smaller than the standard JAVA API). In these projects, the JML specifications are added post hoc, after the JAVA code has already been written. It would have been much more efficient (for us, as verifiers) if the JML specifications had been written together with (or even before) the JAVA implementation. One of the main points behind JML is that writing such specifications at an early stage really pays off. It makes many of the implicit assumptions underlying the implementation explicit (e.g., in the form of invariants), and thus facilitates the use of the code and increases the reliability of software that is based on it. Furthermore, the formal specifications are amenable to tool support, for verification purposes. It is our hope that the use of specification languages like JML (and subsequent verification) becomes standard, certainly for crucial classes in standard libraries. For such library classes, the additional effort is justifiable.

This verification project makes use of two tools: the PVS [34,35] proof tool and the LOOP [17,24,28] translation tool. The latter is a compiler which translates JAVA classes[2] into logical theories in the higher order logic of PVS[3], in the following way.



The generated logical theories contain definitions, embodying the semantics of the classes, plus special lemmas that are used for automatic rewriting. These logical theories can be loaded into the proof tool, together with the so-called semantical prelude, which contains basic definitions, like in Sect. 3 below. Subsequently, the user can state desired properties about the original JAVA classes and prove these on basis of the semantical prelude and the generated theories. For example, a user may want to prove that a method terminates normally, returning a certain value.

The LOOP tool makes use of a semantics of sequential JAVA in higher order logic. This paper includes a description of a relevant part of this (sequential) semantics (see Sect. 3). The methods in JAVA's `Vector` class can be executed in a multi-threaded scenario, but we abstract away from this. Extending our semantics to concurrent JAVA would require a major change, because we would have to transform our big-step semantics into a small-step semantics. Abstracting away from concurrency does not affect the validity of our verification, because all methods that modify the vector are preceded by the keyword `synchronized`. This means that every method first has to acquire a lock before it can execute its body, and thus it is impossible to change the vector in two different ways at the same time. More information can be obtained from [7,18,19,21,22,24].

An important aspect of the verification of the `Vector` invariant is the extensive use we have made (in PVS) of a Hoare logic that can handle abrupt termination (caused, for example, by an exception or a return)(see [19] and Sect. 4). This Hoare logic has various "correctness modes", not only for normal termination as in standard Hoare logic, but also for abrupt termination caused by an exception, return, break or continue. These different modes are needed for reasoning about the frequently occurring abrupt terminations in JAVA programs. In its actual use, the extended Hoare logic is very similar to traditional Hoare logic, involving, for example, variants and invariants to handle while and for loops.

The LOOP tool is currently being extended to also translate JML specifications. They will give rise to specific proof obligations in Hoare logic. The JML specifications used in this paper have been translated by hand, and not automatically, into corresponding Hoare sentences (in PVS), which are used in verifications (see Sect. 5).

[1] Arrays in JAVA have a fixed size; vectors are thus useful if it is not known in advance how many storage positions are needed.

[2] Currently, the translation covers basically all of sequential JAVA (without threads).

[3] The LOOP tool can also produce output for the proof tool ISABELLE [36], but that is not relevant for this verification because it is done with PVS.

1.2 Positioning

This paper presents state-of-the-art work in (object-oriented) program specification and verification, using modern tools both for compilation and for reasoning. The work is not about programs written in some clean, mathematically civilised, abstract programming language, but about actual JAVA programs with all their messy (semantical) details. We consider it a challenge to be able to handle such details. This is the largest case study done so far within the LOOP project. It demonstrates the feasibility of the formal approach to software development, as advocated in the LOOP project.

There are relatively few references on formal verification for object-oriented languages. Specific logics for reasoning about abstract object-oriented programs are proposed in [1, 8, 27]. When it comes to JAVA, one can distinguish between: (1) reasoning about JAVA as a language; and (2) reasoning about programs written in JAVA. In the first category there is work on, for example, safety of the type system [33, 42], or bytecode verification [16, 39, 40]. However, the present paper falls into the second category. Related work in [38] does not, in its current state of development, cover abrupt termination (caused, for instance by exceptions). In [32], a Hoare logic for JAVA is presented which is shown sound and complete with respect to the JAVA semantics presented in [33].

In addition, also being able to reason about abrupt termination (see also [19]) is crucial for the verification in this paper.

The paper is organised as follows: it starts with a brief introduction to the (standard) type theoretic language that will be used (instead of the possibly less familiar language of PVS). Section 3 explains some basic aspects of the semantics of JAVA in this type theory. It forms the basis for our extended Hoare logic in Sect. 4. Section 5 gives a brief introduction to JML, explaining how specifications give rise to proof obligations in Hoare logic. Then, Sect. 6 starts the `Vector` class case study, by first introducing the `Vector` class in JAVA and its translation into PVS. Then it explains the invariant, and its verification by discussing several typical `Vector` methods with their JML specification in detail. Finally, Sect. 7 discusses conclusions and experiences.

2 Type theory

The semantics for JAVA on which the verification effort of this paper relies is sketched in Sect. 3. This semantics is described in a simple type theory with higher order logic. Using this general type theory and logic enables us to abstract away from the particulars of the language of PVS and makes this work more accessible to readers unfamiliar with PVS.

Our type theory is a simple type theory with types built up from:

- type variables;
- type constants nat, bool, string (and some more);
- exponent types $\sigma \to \tau$;
- labeled product (also called record) types $[\,\mathsf{lab}_1 : \sigma_1, \ldots, \mathsf{lab}_n : \sigma_n\,]$;
- labeled coproduct (or variant) types $\{\,\mathsf{lab}_1 : \sigma_1 \mid \ldots \mid \mathsf{lab}_n : \sigma_n\,\}$;

for given types $\sigma, \tau, \sigma_1, \ldots, \sigma_n$, and with all labels lab_i within one (co)product type distinct.

For exponent types, the standard notations for lambda abstraction $\lambda x\colon \sigma.\, M$ and application NL are used. Given terms $M_i\colon \sigma_i$, there exists a labeled tuple $(\,\mathsf{lab}_1 = M_1, \ldots, \mathsf{lab}_n = M_n\,)$ in the corresponding labeled product type $[\,\mathsf{lab}_1\colon \sigma_1, \ldots, \mathsf{lab}_n\colon \sigma_n\,]$. Given a term $N\colon [\,\mathsf{lab}_1\colon \sigma_1, \ldots, \mathsf{lab}_n\colon \sigma_n\,]$ in such a product, $N.\mathsf{lab}_i$ denotes the selection term of type σ_i. Terms for labeled coproducts are formed as follows: given a term $M\colon \sigma_i$ there exists a tagged term $\mathsf{lab}_i\, M$, inhabiting the labeled coproduct type $\{\,\mathsf{lab}_1\colon \sigma_1 \mid \ldots \mid \mathsf{lab}_n\colon \sigma_n\,\}$. For a term $N\colon \{\,\mathsf{lab}_1\colon \sigma_1 \mid \ldots \mid \mathsf{lab}_n\colon \sigma_n\,\}$, and n terms $L_i(x_i)\colon \tau$, where $x_i\colon \sigma_i$ is free in L_i, there is a case term

$$\mathsf{CASE}\ N\ \mathsf{OF}\ \{\,\mathsf{lab}_1\, x_1 \mapsto L_1(x_1) \mid \ldots \mid \mathsf{lab}_n\, x_n \mapsto L_n(x_n)\,\}$$

of type τ, which binds the variables x_i. It reduces to $L_i[M/x_i]$ if N is of the form $\mathsf{lab}_i\, M$. The introduction and elimination terms for exponents, labeled products, and labeled coproducts satisfy standard (β)- and (η)-conversions.

New types can be introduced via definitions, as in:

$$\mathsf{lift}[\alpha] : \mathsf{TYPE} \stackrel{\mathrm{def}}{=} \{\,\mathsf{bot}\colon \mathsf{unit} \mid \mathsf{up}\colon \alpha\,\}$$

where unit is the empty product type $[\,]$. This lift type constructor adds a bottom element to an arbitrary type, given as type variable α.

Formulas in higher order logic are terms of type bool. The connectives $\wedge$ (conjunction), $\vee$ (disjunction), $\supset$ (implication), $\neg$ (negation, used with rules of classical logic) and constants true and false are used, together with the (typed) quantifiers $\forall x\colon \sigma.\, \varphi$ and $\exists x\colon \sigma.\, \varphi$, for a formula φ. There is also a conditional term $\mathsf{IF}\ \varphi\ \mathsf{THEN}\ M\ \mathsf{ELSE}\ N$, for terms M, N of the same type.

3 Java semantics

This section presents the basic ingredients of the semantics for JAVA as used for the `Vector` invariant verification. It describes the semantics of statements and expressions, the underlying memory model, and the formalisation of references (including arrays). Inheritance does not play an important rôle in the `Vector` class, so we will not discuss its type theoretic semantics here, and refer the interested reader to [18] instead.

3.1 Statements and expressions

In classical program semantics the assumption is that statements either terminate normally, resulting in a successor state, or do not terminate at all (see, for example, [6, Chap. 3] or [41, Sect. 2.2]). In the latter case, one also says that the statement hangs, typically because of a non-terminating loop. Hence, statements may be understood as partial functions from states to states. Writing Self as a type variable for the state space, statements can be seen as "state transformer" functions of the type:

– TYPE THEORY

$$\mathsf{Self} \longrightarrow \{\, \mathsf{hang}\colon \mathsf{unit} \mid \mathsf{norm}\colon \mathsf{Self} \,\}$$

This classical view of statements turns out to be inadequate for reasoning about JAVA programs. JAVA statements may hang, or terminate normally (as above), but they may additionally "terminate abruptly" (see, for example, [5, 12]). Abrupt termination may be caused by an exception (for example, caused by a division by 0), a return, a break or a continue (inside a loop). Abrupt (or abnormal) termination is fundamentally different from non-termination: abnormalities may be temporary because they may be caught at some later stage, whereas recovery from non-termination is impossible. Abnormalities can both be thrown and be caught, basically via re-arranging coproduct options. Abrupt termination affects the flow of control: once it arises, all subsequent statements are ignored, until the abnormality is caught (see the definition of composition ";" later in this section). From that moment on, the program executes normally again.

Abrupt termination requires a modification of the standard semantics of statements and expressions, resulting in a failure semantics, as for example in [41, Sect. 5.1]. Therefore, in our approach, statements are modeled as more general state transformer functions

– TYPE THEORY

$$\begin{array}{l} \mathsf{Self} \longrightarrow \mathsf{StatResult}[\mathsf{Self}] \overset{\mathrm{def}}{=} \\ \qquad\qquad \{\, \mathsf{hang} \;\; : \mathsf{unit} \\ \qquad\qquad \mid \mathsf{norm} \;\; : \mathsf{Self} \\ \qquad\qquad \mid \mathsf{abnorm}\colon \mathsf{StatAbn}[\mathsf{Self}] \,\} \end{array}$$

The first option hang captures the situation where a statement hangs. The second option norm occurs when it terminates normally, resulting in a successor state. The final option abnorm describes abrupt termination, yielding a value of the type StatAbn (for Statement Abnormal). It can be subdivided into four parts:

– TYPE THEORY

$$\begin{array}{l} \mathsf{StatAbn}[\mathsf{Self}] : \mathsf{TYPE} \overset{\mathrm{def}}{=} \\ \quad \{\, \mathsf{excp} \;\, : [\, \mathsf{es}\colon \mathsf{Self}, \mathsf{ex}\colon \mathsf{RefType} \,] \\ \quad \mid \mathsf{rtrn} \;\; : \mathsf{Self} \\ \quad \mid \mathsf{break}\colon [\, \mathsf{bs}\colon \mathsf{Self}, \mathsf{blab}\colon \mathsf{lift}[\mathsf{string}] \,] \\ \quad \mid \mathsf{cont} \;\, : [\, \mathsf{cs}\colon \mathsf{Self}, \mathsf{clab}\colon \mathsf{lift}[\mathsf{string}] \,] \,\} \end{array}$$

These four constituents of StatAbn typically consist of a state in Self together with some extra information. An exception abnormality consists of a state together with a reference to an exception object. The reference is represented as an element of the type RefType, which is described below (see Sect. 3.3). A return abnormality only consists of a (tagged) state, and break and continue abnormalities consist of a state, possibly with a label (given as string).

A similar reasoning applies to expressions. In classical semantics, expressions are viewed as functions of the form:

– TYPE THEORY

$$\mathsf{Self} \longrightarrow \mathsf{Out}$$

The type Out is the type of the result. This view is not quite adequate for our purposes, because it does not involve non-termination, abrupt termination or side-effects. As statements, expressions in JAVA may hang, terminate normally or terminate abruptly. If an expression terminates normally, it produces an output result (of the type of the expression) together with a successor state (since it may have a side-effect). If it terminates abruptly, this can only be because of an exception (and not because of a break, continue or return (see [12, §15.5]). Hence an expression of type Out is (in our view) a function of the form:

– TYPE THEORY

$$\begin{array}{l} \mathsf{Self} \longrightarrow \mathsf{ExprResult}[\mathsf{Self}, \mathsf{Out}] \overset{\mathrm{def}}{=} \\ \qquad\qquad \{\, \mathsf{hang} \;\;\, : \mathsf{unit} \\ \qquad\qquad \mid \mathsf{norm} \;\;\, : [\, \mathsf{ns}\colon \mathsf{Self}, \mathsf{res}\colon \mathsf{Out} \,] \\ \qquad\qquad \mid \mathsf{abnorm}\colon \mathsf{ExprAbn}[\mathsf{Self}] \,\} \end{array}$$

Notice that the second option norm occurs when an expression terminates normally, resulting in a successor state together with an output result. The third option abnorm describes abrupt termination – because of an exception – for expressions:

– TYPE THEORY

$$\begin{array}{l} \mathsf{ExprAbn}[\mathsf{Self}] : \mathsf{TYPE} \overset{\mathrm{def}}{=} \\ \quad [\, \mathsf{es}\colon \mathsf{Self}, \mathsf{ex}\colon \mathsf{RefType} \,] \end{array}$$

To summarise, in our formalisation, statements are modeled as functions from Self to StatResult[Self], and expressions as functions from Self to ExprResult[Self, Out], for the appropriate result type Out. This abstract representation of statements and expressions as "one entry/multi-exit" functions (terminology of [9]) forms the basis for the work presented here. It is used to give a (compositional) meaning to basic programming constructs like composition, if-then-else, and while. For example, the statement composition operator "`;`" of JAVA is translated into ";" in type theory. Thus, for JAVA statements `s`, `t`,

$$[\![\texttt{s;t}]\!] = [\![\texttt{s}]\!] \,;[\![\texttt{t}]\!]$$

where the definition of ; in type theory is as follows:

– TYPE THEORY

$s, t\colon \mathsf{Self} \to \mathsf{StatResult[Self]} \ \vdash$

$\quad (s\,;t) \ : \ \mathsf{Self} \to \mathsf{StatResult[Self]} \ \stackrel{\mathrm{def}}{=}$

$\qquad \lambda x\colon \mathsf{Self}.\ \mathsf{CASE}\ s\,x\ \mathsf{OF}\ \{$

$\qquad\qquad \mid \mathsf{hang} \mapsto \mathsf{hang}$

$\qquad\qquad \mid \mathsf{norm}\,y \mapsto t\,y$

$\qquad\qquad \mid \mathsf{abnorm}\,a \mapsto \mathsf{abnorm}\,a\ \}$

Thus if statement s terminates normally in state x, resulting in a next state y, then $(s\,;t)\,x$ is $t\,y$. In addition, if s hangs or terminates abruptly in state x, then $(s\,;t)\,x$ is $s\,x$ and t is not executed.

A typical example of an abruptly terminating statement in JAVA is the `return` statement. When a `return` statement is executed, the program immediately exits from the current method. A `return` statement may have an expression argument; if so, this expression is evaluated and returned as the result of the method. The translation of the JAVA `return` statement (without argument) is,

$$[\![\texttt{return}]\!] = \mathsf{RETURN}$$

where RETURN is defined as:

– TYPE THEORY

$\mathsf{RETURN} \ : \ \mathsf{Self} \to \mathsf{StatResult[Self]} \ \stackrel{\mathrm{def}}{=}$

$\qquad\qquad\qquad \lambda x\colon \mathsf{Self}.\ \mathsf{abnorm}(\mathsf{rtrn}\ x)$

This statement produces an abnormal state, which will be caught at the end of a method body. The translation of a `return` statement with argument is similar, but more subtle. First, the value of the expression is stored in a special local variable, and then the state becomes abnormal, via the above RETURN.

To recover from this return abnormality, functions CATCH-STAT-RETURN and CATCH-EXPR-RETURN are used. In our translation of JAVA programs, a function CATCH-STAT-RETURN is wrapped around every method body that returns `void`. First, the method body is evaluated. This may result in an abnormal state, because of a return. In that case the function CATCH-STAT-RETURN turns the (abnormal) state back to normal again. Otherwise, it leaves the state unchanged.

– TYPE THEORY

$s\colon \mathsf{Self} \to \mathsf{StatResult[Self]} \ \vdash$

$\quad \mathsf{CATCH\text{-}STAT\text{-}RETURN}(s) \ : \ \mathsf{Self} \to \mathsf{StatResult[Self]} \ \stackrel{\mathrm{def}}{=}$

$\qquad \lambda x\colon \mathsf{Self}.\ \mathsf{CASE}\ s\,x\ \mathsf{OF}\ \{$

$\qquad\qquad \mid \mathsf{hang} \mapsto \mathsf{hang}$

$\qquad\qquad \mid \mathsf{norm}\,y \mapsto \mathsf{norm}\,y$

$\qquad\qquad \mid \mathsf{abnorm}\,a \mapsto$

$\qquad\qquad\quad \mathsf{CASE}\ a\ \mathsf{OF}\ \{$

$\qquad\qquad\qquad \mid \mathsf{excp}\,e \mapsto \mathsf{abnorm}(\mathsf{excp}\,e)$

$\qquad\qquad\qquad \mid \mathsf{rtrn}\,z \mapsto \mathsf{norm}\,z$

$\qquad\qquad\qquad \mid \mathsf{break}\,b \mapsto \mathsf{abnorm}(\mathsf{break}\,b)$

$\qquad\qquad\qquad \mid \mathsf{cont}\,c \mapsto \mathsf{abnorm}(\mathsf{cont}\,c)\ \}\ \}$

If a method returns a value, a function CATCH-EXPR-RETURN is used, instead of CATCH-STAT-RETURN. Recall that the result value of a method is stored in a special variable. This function CATCH-EXPR-RETURN turns the state back to normal and, in that case, returns the output that is held by this special variable.

Below, a similar function CATCH-CONTINUE is used, which catches an abnormal state, because of a `continue`, and turns it back to normal. Since `continue` statements can only occur in loops, with the effect that control skips the rest of the loop's body and starts re-evaluating (the update statement, in a `for` loop, and) the Boolean expression which controls the loop, this function is used in the semantics of loops.

Finally, there is one technicality that deserves some attention. Sometimes an expression has to be transformed into a statement, which is only a matter of forgetting the result of the expression. However, in our formalisation this transformation has to be done explicitly, using a function E2S.

– TYPE THEORY

$e\colon \mathsf{Self} \to \mathsf{ExprResult[Self, Out]} \ \vdash$

$\quad \mathsf{E2S}(e) \ : \ \mathsf{Self} \to \mathsf{StatResult[Self]} \ \stackrel{\mathrm{def}}{=}$

$\qquad \lambda x\colon \mathsf{Self}.\ \mathsf{CASE}\ e\,x\ \mathsf{OF}\ \{$

$\qquad\qquad \mid \mathsf{hang} \mapsto \mathsf{hang}$

$\qquad\qquad \mid \mathsf{norm}\,y \mapsto \mathsf{norm}(y.\mathsf{ns})$

$\qquad\qquad \mid \mathsf{abnorm}\,a \mapsto \mathsf{abnorm}(\mathsf{excp}(\mathsf{es} = a.\mathsf{es},$

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{ex} = a.\mathsf{ex}))\ \}$

In the last line an expression abnormality (an exception) is transformed into a statement abnormality.

A more detailed elaboration of this semantics can be found in [19, 21].

3.2 Memory model

This section starts by defining memory cells for storing JAVA objects and arrays. They are used to build up the main memory for storing arbitrarily many such items, without any garbage collection. This object memory OM comes with various operations for reading and writing. More information on this memory model is given in [7].

3.2.1 Memory cells

A single memory cell (or object cell) can store the contents of all the fields from a single object belonging to an arbitrary class. The (translated) types that the fields of objects can have are limited to byte, short, int, long, char, float, double, bool and RefType (which is defined below in Sect. 3.3). Therefore, an object cell has entries for all of these. The number of fields for a particular type is not bounded, so infinitely many are incorporated in a memory cell. This makes memory cells basically untyped in the sense that they can store the contents of the fields of an arbitrary JAVA object. The ability to do this is the main contribution of our memory model. Often this typing problem is avoided by only considering one or two types.

– TYPE THEORY

$$
\begin{array}{l}
\mathsf{ObjectCell} : \mathsf{TYPE} \stackrel{\mathrm{def}}{=} \\
\quad [\,\mathsf{bytes}\colon \mathsf{CellLoc} \to \mathsf{byte}, \\
\quad\;\; \mathsf{shorts}\colon \mathsf{CellLoc} \to \mathsf{short}, \\
\quad\;\; \mathsf{ints}\colon \mathsf{CellLoc} \to \mathsf{int}, \\
\quad\;\; \mathsf{longs}\colon \mathsf{CellLoc} \to \mathsf{long}, \\
\quad\;\; \mathsf{chars}\colon \mathsf{CellLoc} \to \mathsf{char}, \\
\quad\;\; \mathsf{floats}\colon \mathsf{CellLoc} \to \mathsf{float}, \\
\quad\;\; \mathsf{doubles}\colon \mathsf{CellLoc} \to \mathsf{double}, \\
\quad\;\; \mathsf{booleans}\colon \mathsf{CellLoc} \to \mathsf{bool}, \\
\quad\;\; \mathsf{refs}\colon \mathsf{CellLoc} \to \mathsf{RefType}\,]
\end{array}
$$

where the type CellLoc is defined separately, simply as nat, for reasons of abstraction. Our memory is organised in such a way that each memory location points to a memory cell, and each cell location to a position for a particular label inside the cell.

Storing an object belonging to a class with, for instance, two integer fields and one Boolean field in a memory cell is done by (only) using the first two values (at 0 and at 1) of the function ints: CellLoc → int and (only) the first value (at 0) of the function booleans: CellLoc → bool. Other values of these and other functions in the object cell are irrelevant, and are not used for objects belonging to this class. Enormous storage capacity is wasted in this manner, but that is unproblematic. The LOOP compiler attributes these cell locations to (static) fields of a class, local variables, and parameters. These cell locations are hidden from the user.

3.2.2 Object memory

Object cells form the main ingredient of a new type OM representing the main memory. It has a heap, stack, and static part, for storing the contents of, respectively, instance variables, local variables and parameters, and static (also called class) variables:

– TYPE THEORY

$$
\begin{array}{l}
\mathsf{OM} : \mathsf{TYPE} \stackrel{\mathrm{def}}{=} \\
\quad [\,\mathsf{heapmem}\colon \mathsf{MemLoc} \to \mathsf{ObjectCell}, \\
\quad\;\; \mathsf{heaptop}\colon \mathsf{MemLoc}, \\
\quad\;\; \mathsf{stackmem}\colon \mathsf{MemLoc} \to \mathsf{ObjectCell}, \\
\quad\;\; \mathsf{stacktop}\colon \mathsf{MemLoc}, \\
\quad\;\; \mathsf{staticmem}\colon \mathsf{MemLoc} \to [\,\mathsf{initialised}\colon \mathsf{bool}, \\
\quad\;\; \qquad\qquad\qquad\qquad\quad \mathsf{staticcell}\colon \mathsf{ObjectCell}\,]\,]
\end{array}
$$

Once again, for abstraction we define MemLoc: Type $\stackrel{\mathrm{def}}{=}$ nat. The entry heaptop (resp. stacktop) indicates the next free (unused) memory location on the heap (or stack). These numbers change during program execution (in type theory). The LOOP compiler assigns locations (in the static memory) to classes with static fields. At such locations a Boolean initialised tells whether static initialisation has taken place for this class. One must keep track of this because static initialisation should be performed at most once.

3.2.3 Reading and writing in the object memory

Accessing a specific value in an object memory x: OM, either for reading or for writing, involves the following three ingredients: (1) an indication of which memory (heap, stack, static); (2) a memory location (in MemLoc); and (3) a cell location (in CellLoc) giving the offset in the cell. These ingredients are combined in the following variant type for memory addressing:

– TYPE THEORY

$$
\begin{array}{l}
\mathsf{MemAdr} : \mathsf{TYPE} \stackrel{\mathrm{def}}{=} \\
\quad \{\,\mathsf{heap}\colon [\,\mathsf{ml}\colon \mathsf{MemLoc}, \mathsf{cl}\colon \mathsf{CellLoc}\,] \\
\quad\;\, |\,\mathsf{stack}\colon [\,\mathsf{ml}\colon \mathsf{MemLoc}, \mathsf{cl}\colon \mathsf{CellLoc}\,] \\
\quad\;\, |\,\mathsf{static}\colon [\,\mathsf{ml}\colon \mathsf{MemLoc}, \mathsf{cl}\colon \mathsf{CellLoc}\,]\,\}
\end{array}
$$

For each type typ from the collection of types byte, short, int, long, char, float, double, bool, and RefType occurring in object cells (see the definition of ObjectCell), there are two operations:

– TYPE THEORY

$$\begin{array}{rl} \mathsf{get\_typ}(x,m)\colon \mathsf{typ} & \text{for } x\colon \mathsf{OM}, m\colon \mathsf{MemAdr} \\ \mathsf{put\_typ}(x,m,u)\colon \mathsf{OM} & \text{for } x\colon \mathsf{OM}, m\colon \mathsf{MemAdr}, u\colon \mathsf{typ} \end{array}$$

These functions are described in detail only for typ = byte; the other cases are similar. Reading from the memory is easy: for $x\colon \mathsf{OM}, m\colon \mathsf{MemAdr}$

– TYPE THEORY

$$\begin{array}{l} \mathsf{get\_byte}(x,m) \stackrel{\text{def}}{=} \\ \mathsf{CASE}\ m\ \mathsf{OF}\ \{ \\ \quad \mid \mathsf{heap}\ \ell \mapsto ((x.\mathsf{heapmem}(\ell.\mathsf{ml})).\mathsf{bytes})(\ell.\mathsf{cl}) \\ \quad \mid \mathsf{stack}\ \ell \mapsto ((x.\mathsf{stackmem}(\ell.\mathsf{ml})).\mathsf{bytes})(\ell.\mathsf{cl}) \\ \quad \mid \mathsf{static}\ \ell \mapsto ((x.\mathsf{staticmem}(\ell.\mathsf{ml})).\mathsf{staticcell}.\mathsf{bytes})(\ell.\mathsf{cl})\ \} \end{array}$$

The corresponding write-operation uses updates of records and also updates of functions; both these use a ‘WITH’ notation, which is hopefully self-explanatory: for $x\colon \mathsf{OM}$, $m\colon \mathsf{MemAdr}$ and $u\colon \mathsf{byte}$

– TYPE THEORY

$$\begin{array}{l} \mathsf{put\_byte}(x,m,u) \stackrel{\text{def}}{=} \\ \mathsf{CASE}\ m\ \mathsf{OF}\ \{ \\ \quad \mid \mathsf{heap}\ \ell \mapsto x\ \mathsf{WITH} \\ \qquad [\ ((x.\mathsf{heapmem}(\ell.\mathsf{ml})).\mathsf{bytes})(\ell.\mathsf{cl}) = u\ ] \\ \quad \mid \mathsf{stack}\ \ell \mapsto x\ \mathsf{WITH} \\ \qquad [\ ((x.\mathsf{stackmem}(\ell.\mathsf{ml})).\mathsf{bytes})(\ell.\mathsf{cl}) = u\ ] \\ \quad \mid \mathsf{static}\ \ell \mapsto x\ \mathsf{WITH} \\ \qquad [\ ((x.\mathsf{staticmem}(\ell.\mathsf{ml})).\mathsf{staticcell}.\mathsf{bytes})(\ell.\mathsf{cl}) = u\ ]\ \} \end{array}$$

The various get- and put-functions (18 in total) satisfy obvious commutation equations, such as:

– TYPE THEORY

$$\begin{array}{rl} \mathsf{get\_byte}(\mathsf{put\_byte}(x,m,u),n) = & \mathsf{IF}\ m = n \\ & \mathsf{THEN}\ u \\ & \mathsf{ELSE}\ \mathsf{get\_byte}(x,n) \\ \mathsf{get\_byte}(\mathsf{put\_short}(x,m,v),n) = & \mathsf{get\_byte}(x,n). \end{array}$$

Such equations (81 in total) are used for auto-rewriting: whenever these equations can be applied, the back-end proof-tool simplifies goals automatically.

3.3 Formalising references to objects and arrays

Reference types are used in JAVA for objects and arrays. A reference may be `null`, indicating that it does not refer to anything. In our model, a non-null reference contains a pointer ‘objpos’ to a memory location (on the heap, see Sect. 3.2.2), together with a string ‘clname’ indicating the run-time type of the object, or the run-time elementtype of the array, at this location, and possibly two natural numbers describing the dimension and length of non-null array references. This leads to the following definition[4]:

– TYPE THEORY

$$\begin{array}{l} \mathsf{RefType} : \mathsf{TYPE} \stackrel{\text{def}}{=} \\ \quad \{\ \mathsf{null}\colon \mathsf{unit} \mid \mathsf{ref}\colon [\ \mathsf{objpos}\colon \mathsf{MemLoc}, \\ \qquad\qquad\qquad\qquad \mathsf{clname}\colon \mathsf{string}, \\ \qquad\qquad\qquad\qquad \mathsf{dimlen}\colon \mathsf{lift}[\ [\ \mathsf{dim}\colon \mathsf{nat}, \mathsf{len}\colon \mathsf{nat}\ ]\ ]\ ]\ \} \end{array}$$

Based on this type RefType, various operations on references can be formalised in type theory, e.g., comparing two references is translated as

$$[\![\texttt{r1 == r2}]\!] \stackrel{\text{def}}{=} [\![\texttt{r1}]\!] == [\![\texttt{r2}]\!]$$

where == is defined in type theory in Fig. 1, following [12, §§ 15.20.3].

Appropriate operations, such as accessing and storing elements in an array are formalised. A multi-dimensional array is represented as an array of arrays, e.g., a two-dimensional array of Booleans is stored as an array of references, and these references in their turn are one-dimensional arrays of Booleans. For example, the array_access function, defined below, is used for the translation of indexing an array, in the following way:

$$[\![\texttt{a[i]}]\!] \stackrel{\text{def}}{=} \mathsf{array\_access}(\mathsf{get\_typ}, [\![\texttt{a}]\!], [\![\texttt{i}]\!])$$

assuming that `a[i]` is not the left-hand side of an assignment. The function get_typ is determined by the component type of `a`, for example: if `a` is an integer array of type `int[]`, then get_typ = get_int. Furthermore, if `a` is a 2-dimensional array of, say Booleans, then get_typ = get_ref.

The JAVA evaluation strategy prescribes that, first, the array expression, and then the index expression must be evaluated. Subsequently, it must be checked first whether the array reference is non-null, and then whether the (evaluated) index is non-negative and below the length of the array. Only then can the memory be accessed (see [12, §§ 15.12.1 and §§ 15.12.2]). This is described in our setting as in Fig. 2 (omitting the details of how exceptions are thrown; for more information on exceptions the reader is referred to [22]). Notice that the function array_access has to be defined over the object memory OM, instead of over an arbitrary state space Self, because it uses the get- and put-functions defined on OM.

[4] Recently, we have changed the representation of references: the clname and dimlen entry have moved from references to object cells (see [21]). However, the verification described here has been done with references as described.

Notice that arrays, like objects, are stored on the heap. All translated non-null array references have a non-bottom dimlen field by construction, so, in the case indicated as "should not happen", we choose to use hang as output. We also could have thrown some non-standard exception. There is a similar function array_assign which is used for assigning a value at a particular index in an array. Further, there are also functions for array creation and returning the array length. The function for array creation sets up an appropriately linked number of (empty) memory cells, depending on the dimension and lengths of the array that is being created.

4 A Hoare logic for Java

Many verifications of JAVA programs can be done immediately in terms of the semantics as described in Sect. 3.

– TYPE THEORY

```
$r_1, r_2 : \mathsf{Self} \to \mathsf{ExprResult}[\mathsf{Self}, \mathsf{RefType}] \vdash$
  $r_1 == r_2 : \mathsf{Self} \to \mathsf{ExprResult}[\mathsf{Self}, \mathsf{bool}] \stackrel{\text{def}}{=}$
    $\lambda x : \mathsf{Self}.\ \mathsf{CASE}\ r_1\ x\ \mathsf{OF}\ \{$
              $|\ \mathsf{hang} \mapsto \mathsf{hang}$
              $|\ \mathsf{norm}\ y \mapsto \mathsf{CASE}\ r_2\ (y.\mathsf{ns})\ \mathsf{OF}\ \{$
                         $|\ \mathsf{hang} \mapsto \mathsf{hang}$
                         $|\ \mathsf{norm}\ z \mapsto \mathsf{norm}\ (\mathsf{ns} = z.\mathsf{ns}, \mathsf{res} = \mathsf{CASE}\ y.\mathsf{res}\ \mathsf{OF}\ \{$
                                                   $|\ \mathsf{null} \mapsto \mathsf{CASE}\ z.\mathsf{res}\ \mathsf{OF}\ \{$
                                                           $|\ \mathsf{null} \mapsto \mathsf{true}$
                                                           $|\ \mathsf{ref}\ s \mapsto \mathsf{false}\ \}$
                                                   $|\ \mathsf{ref}\ r \mapsto \mathsf{CASE}\ z.\mathsf{res}\ \mathsf{OF}\ \{$
                                                           $|\ \mathsf{null} \mapsto \mathsf{false}$
                                                           $|\ \mathsf{ref}\ s \mapsto r.\mathsf{objpos} = s.\mathsf{objpos}\ \}\ \})$
                         $|\ \mathsf{abnorm}\ b \mapsto \mathsf{abnorm}\ b\ \}$
              $|\ \mathsf{abnorm}\ a \mapsto \mathsf{abnorm}\ a\ \}$
```

Fig. 1. Formalisation of Java's equality == in type theory

– TYPE THEORY

```
$a : \mathsf{OM} \to \mathsf{ExprResult}[\mathsf{OM}, \mathsf{RefType}], i : \mathsf{OM} \to \mathsf{ExprResult}[\mathsf{OM}, \mathsf{int}] \vdash$
  $\mathsf{array\_access}(\mathsf{get\_typ}, a, i) : \mathsf{OM} \to \mathsf{ExprResult}[\mathsf{OM}, \mathsf{typ}] \stackrel{\text{def}}{=}$
    $\lambda x : \mathsf{OM}.\ \mathsf{CASE}\ a\ x\ \mathsf{OF}\ \{$
              $|\ \mathsf{hang} \mapsto \mathsf{hang}$
              $|\ \mathsf{norm}\ y \mapsto \mathsf{CASE}\ i\ (y.\mathsf{ns})\ \mathsf{OF}\ \{$
                         $|\ \mathsf{hang} \mapsto \mathsf{hang}$
                         $|\ \mathsf{norm}\ z \mapsto \mathsf{CASE}\ y.\mathsf{res}\ \mathsf{OF}\ \{$
                                 $|\ \mathsf{null} \mapsto [\![$new NullPointerException()$]\!]$
                                 $|\ \mathsf{ref}\ r \mapsto \mathsf{CASE}\ r.\mathsf{dimlen}\ \mathsf{OF}\ \{$
                                         $|\ \mathsf{bot} \mapsto \mathsf{hang}$ // should not happen
                                         $|\ \mathsf{up}\ p \mapsto \mathsf{IF}\ z.\mathsf{res} < 0 \vee z.\mathsf{res} \geq p.\mathsf{len}$
                                                 $\mathsf{THEN}\ [\![$new ArrayIndexOutOfBoundsException()$]\!]$
                                                 $\mathsf{ELSE}\ \mathsf{norm}(\mathsf{ns} = z.\mathsf{ns}, \mathsf{res} =$
                                                         $\mathsf{get\_typ}(z.\mathsf{ns}, \mathsf{heap}(\mathsf{ml} = r.\mathsf{objpos}, \mathsf{cl} = z.\mathsf{res})))\ \}\ \}$
                         $|\ \mathsf{abnorm}\ c \mapsto \mathsf{abnorm}\ c\ \}$
              $|\ \mathsf{abnorm}\ b \mapsto \mathsf{abnorm}\ b\ \}$
```

Fig. 2. Accessing array a at index i in type theory

However, "... reasoning about correctness formulas in terms of semantics is not very convenient. A much more promising approach is to reason directly on the level of correctness formulas." (quote from [3, p. 57]). Hoare logic [20] is a formalism for doing precisely this. This section describes an extension of Hoare logic which is especially tailored to JAVA. The proof rules that are discussed here are heavily used in the `Vector` case study described below.

Our Hoare logic extension is a concrete and detailed elaboration and adaptation of existing approaches to programming logics with exceptions, notably from [9, 26, 29] (which are mostly in weakest precondition form). Although the basic ideas used here are the same as in [9, 26, 29], the elaboration is different. For example, in this elaboration many forms of abrupt termination are considered, and not just one sole exception. In addition, a semantics of statements and expressions as particular functions is used (as described in Sect. 3), and not a semantics of traces.

Hoare logic, for a particular programming language, consists of a series of deduction rules for special sentences, involving constructs from the programming language, such as assignment, if-then-else and composition. In particular, (while) loops have received much attention in Hoare logic, because they involve a judicious and often non-trivial choice of a loop invariant. For more information (see, for example, [3, 4, 6, 13, 15]). There is a so-called "classical" body of Hoare logic, which applies to standard constructs from an idealised imperative programming language. This forms a well-developed part of the theory of Hoare logic. It is based on sentences of the form $\{P\}\,S\,\{Q\}$, for partial correctness, or $[P]\,S\,[Q]$, for total correctness. They involve assertions P and Q in some logic (usually predicate logic), and statements S from the programming language that one wishes to reason about. The partial correctness sentence $\{P\}\,S\,\{Q\}$ expresses that if the assertion P holds in some state x and *if* the statement S, when evaluated in state x, terminates normally, resulting in a state x', then the assertion Q holds in x'. Total correctness $[P]\,S\,[Q]$ expresses something stronger, namely: if P holds in x, *then* S in x terminates normally, resulting in a state x' where Q holds. These partial and total correctness sentences can be translated easily into our type theory.

TYPE THEORY

$$\begin{array}{l}
\mathsf{pre},\mathsf{post}\colon \mathsf{Self}\to\mathsf{bool}, \mathsf{stat}\colon \mathsf{Self}\to\mathsf{StatResult}[\mathsf{Self}] \vdash \\
\quad \mathsf{PartialNormal?}(\mathsf{pre},\mathsf{stat},\mathsf{post}) \,:\, \mathsf{bool} \overset{\mathrm{def}}{=} \\
\qquad \forall x\colon \mathsf{Self}.\, \mathsf{pre}\,x \supset \mathsf{CASE}\ \mathsf{stat}\,x\ \mathsf{OF}\ \{ \\
\qquad\qquad\qquad\qquad \mid \mathsf{hang}\mapsto\mathsf{true} \\
\qquad\qquad\qquad\qquad \mid \mathsf{norm}\,y\mapsto\mathsf{post}\,y \\
\qquad\qquad\qquad\qquad \mid \mathsf{abnorm}\,a\mapsto\mathsf{true}\,\}
\end{array}$$

$$\begin{array}{l}
\mathsf{pre},\mathsf{post}\colon \mathsf{Self}\to\mathsf{bool}, \mathsf{stat}\colon \mathsf{Self}\to\mathsf{StatResult}[\mathsf{Self}] \vdash \\
\quad \mathsf{TotalNormal?}(\mathsf{pre},\mathsf{stat},\mathsf{post}) \,:\, \mathsf{bool} \overset{\mathrm{def}}{=} \\
\qquad \forall x\colon \mathsf{Self}.\, \mathsf{pre}\,x \supset \mathsf{CASE}\ \mathsf{stat}\,x\ \mathsf{OF}\ \{ \\
\qquad\qquad\qquad\qquad \mid \mathsf{hang}\mapsto\mathsf{false} \\
\qquad\qquad\qquad\qquad \mid \mathsf{norm}\,y\mapsto\mathsf{post}\,y \\
\qquad\qquad\qquad\qquad \mid \mathsf{abnorm}\,a\mapsto\mathsf{false}\,\}
\end{array}$$

To adapt this classical body to JAVA, proof rules are described for normally terminating statements. Following Gordon [14], these proof rules are shown to be sound with respect to the semantics. In our case, the soundness of all the rules with respect to our semantics has been proven in PVS (and ISABELLE), but completeness is not our main concern, because we want to actually use these rules and not study their meta-theoretical properties (as in [32]).

This enables both semantic and axiomatic reasoning about (translated)[5] JAVA programs. These (standard) proof rules are described in more detail in [19, 21].

4.1 A Hoare logic with abrupt termination

Unfortunately, the proof rules for normal termination do not give enough power to reason about arbitrary JAVA programs. Therefore it is necessary to have a "correctness notion" for being in an abnormal state, e.g., if execution of S starts in a state satisfying P, then execution of S terminates abruptly, because of a `return`, in a state satisfying Q. To this end, the notions of abnormal correctness are introduced. They appear in four forms, corresponding to the four possible kinds of abnormalities. Rules are formulated to derive the (abnormal) correctness of a program compositionally. These rules allow the user to move back and forth between the various correctness notions.

The first notion that is introduced is partial break correctness (with notation: $\{P\}\,S\,\{\mathsf{break}(Q,l)\}$), meaning that if execution of S starts in some state satisfying P, and execution of S terminates in an abnormal state, because of a **`break`**, then the resulting abnormal state satisfies Q. If the **`break`** is labeled with `lab`, then $l = \mathsf{up}(\text{"}\mathtt{lab}\text{"})$, otherwise $l = \mathsf{bot}$.

Naturally, there exists also *total* break correctness ($[P]\,S\,[\mathsf{break}(Q,l)]$), meaning that if execution of S starts in some state satisfying P, then execution of S terminates in an abnormal state, satisfying Q, because of a **`break`**. If this **`break`** is labeled with `lab`, then $l = \mathsf{up}(\text{"}\mathtt{lab}\text{"})$, otherwise $l = \mathsf{bot}$. Continuing in this manner leads to the following eight notions of abnormal correctness:

[5] Traditionally, Hoare logic rules are syntax-based. Our rules are semantics-based, which allows us to reason about translated programs. Since the translation is compositional, we can still follow the program structure.

partial break correctness
$\{P\}\, S\, \{\mathsf{break}(Q, l)\}$
partial continue correctness
$\{P\}\, S\, \{\mathsf{continue}(Q, l)\}$
partial return correctness
$\{P\}\, S\, \{\mathsf{return}(Q)\}$
partial exception correctness
$\{P\}\, S\, \{\mathsf{exception}(Q, E)\}$
total break correctness
$[P]\, S\, [\mathsf{break}(Q, l)]$
total continue correctness
$[P]\, S\, [\mathsf{continue}(Q, l)]$
total return correctness
$[P]\, S\, [\mathsf{return}(Q)]$
total exception correctness
$[P]\, S\, [\mathsf{exception}(Q, E)]$

The formalisation of these correctness notions in type theory is straightforward. As an example, consider the predicate PartialReturn? for partial return correctness. This is used to give meaning to the notation

$$\{P\}\, [\![S]\!]\, \{\mathsf{return}(Q)\} = \mathsf{PartialReturn?}(P, [\![S]\!], Q).$$

This predicate is defined as follows:

– TYPE THEORY

$$\begin{array}{l}
\mathsf{pre}, \mathsf{post}\colon \mathsf{Self} \to \mathsf{bool}, \mathsf{stat}\colon \mathsf{Self} \to \mathsf{StatResult[Self]} \vdash \\
\quad \mathsf{PartialReturn?}(\mathsf{pre}, \mathsf{stat}, \mathsf{post}) : \mathsf{bool} \overset{\text{def}}{=} \\
\qquad \forall x\colon \mathsf{Self}.\ \mathsf{pre}\, x \supset \mathsf{CASE}\ \mathsf{stat}\, x\ \mathsf{OF}\ \{ \\
\qquad\qquad\qquad\qquad \mid \mathsf{hang} \mapsto \mathsf{true} \\
\qquad\qquad\qquad\qquad \mid \mathsf{norm}\, y \mapsto \mathsf{true} \\
\qquad\qquad\qquad\qquad \mid \mathsf{abnorm}\, a \mapsto \mathsf{CASE}\ a\ \mathsf{OF}\ \{ \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{excp}\, e \mapsto \mathsf{true} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{rtrn}\, z \mapsto \mathsf{post}\, z \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{break}\, b \mapsto \mathsf{true} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{cont}\, c \mapsto \mathsf{true}\ \}\ \}
\end{array}$$

Many straightforward proof rules can be formulated and proven, for these correctness notions. First of all, there are the analogues of the skip axiom, e.g.:

– TYPE THEORY

$$\{P\}\ \mathsf{RETURN}\ \{\mathsf{return}(P)\}$$

Then there are rules, expressing how these (partial and total) correctness notions behave with "traditional" program constructs, e.g., with statement composition.

– TYPE THEORY

$$\frac{[P]\, S\, [\mathsf{return}(R)]}{[P]\, S\,; T\, [\mathsf{return}(R)]}$$

$$\frac{[P]\, S\, [Q] \qquad [Q]\, T\, [\mathsf{return}(R)]}{[P]\, S\,; T\, [\mathsf{return}(R)]}$$

$$\frac{\{P\}\, S\, \{\mathsf{return}(R)\} \qquad \{P\}\, S\, \{Q\} \qquad \{Q\}\, T\, \{\mathsf{return}(R)\}}{\{P\}\, S\,; T\, \{\mathsf{return}(R)\}}$$

There are rules to move between two correctness notions, from normal to abnormal and vice versa. Here are some examples for the return statement again:

– TYPE THEORY

$$\frac{\{P\}\, S\, \{\mathsf{return}(Q)\} \qquad \{P\}\, S\, \{Q\}}{\{P\}\ \mathsf{CATCH\text{-}STAT\text{-}RETURN}(S)\ \{Q\}}$$

$$\frac{[P]\, S\, [\mathsf{return}(Q)]}{[P]\ \mathsf{CATCH\text{-}STAT\text{-}RETURN}(S)\ [Q]}$$

Most of these proof rules are easy and straightforward to formulate, and they provide a good framework to reason about programs in JAVA. However, proof rules for while loops with abrupt termination are more difficult to formulate.

4.2 Hoare logic of while loops with abrupt termination

Recall that in classical Hoare logic, reasoning about while loops involves the following ingredients: (1) an invariant, i.e., a predicate over the state space which is true initially and after each iteration of the while loop; (2) a condition, which is false after normal termination of the while loop; (3) a body, which is iterated a number of times; and (4) (when dealing with total correctness) a variant, i.e., a mapping from the state space to some well-founded set, which strictly decreases every time the body is executed. To see what is needed to extend this to abnormal correctness, first a silly example of an abruptly terminating while loop is discussed.

– JAVA
```
  while (true) { if (i < 10) { i++; }
                 else { break; } }
```

This loop always terminates, and a variant can be constructed to show this, but after termination it cannot be concluded that the condition has become false. Thus, proof rules have to be formulated in such a way that, in this case, it can be concluded that after termination of the while loop `i < 10` does not hold (anymore). This desire leads to the development of special rules for partial and total abnormal correctness of while loops. Below, the partial and total break correctness rules are described in full detail, the rules for the other abnormalities are basically the same. The JAVA `while` statement is formalised in type theory by a function $\mathsf{WHILE}(l)(C)(S)$, where l is a formalisation of the possible label of the `while` statement, C is a formalisation of the condition, and S of the body. This definition boils down to iterating the so-called iteration body

$$\mathsf{E2S}(C)\,;\mathsf{CATCH\text{-}CONTINUE}(l)(S)$$

an appropriate number of times. More information on the definition of WHILE can be found in [19,21].

4.2.1 Partial break while rule

Assume a while loop $\mathsf{WHILE}(l_1)(C)(S)$, which will be executed in a state satisfying P. The aim is to prove that if the while loop terminates abruptly, because of a break, then the result state satisfies Q – where P is the loop invariant and Q is the condition which holds upon abrupt termination (in the example above: $i \geq 10$). A natural condition for the proof rule is thus that if the body terminates abruptly, because of a break, then Q should hold. Furthermore, it should be shown that P is an invariant if the body terminates normally. This leads to the proof rule in Fig. 3.

This rule states the following: suppose: (1) if the iteration body $\mathsf{E2S}(C)\,;\mathsf{CATCH\text{-}CONTINUE}(l_1)(S)$ is executed in a state satisfying P and terminates normally, then P still holds; and (2) if the iteration body is executed in a state satisfying P and ends in an abnormal state, because of a break, then this state satisfies some property Q. Then, if the while statement is executed in a state satisfying P and it terminates abruptly, because of a break, then its final state satisfies Q.

The soundness of this rule is easy to see (and to prove): suppose there exists a state satisfying P, in which the statement $\mathsf{WHILE}(l_1)(C)(S)$ terminates abruptly, because of a break. This means that the iteration body involved, $\mathsf{E2S}(C)\,;\mathsf{CATCH\text{-}CONTINUE}(l_1)(S)$, terminates normally a number of times. All these times, P remains true. However, at some stage the iterated statement must terminate abruptly, because of a break, labeled l_2, and then the resulting state satisfies Q. As this is also the final state of the whole loop, $\{P\}\,\mathsf{WHILE}(l_1)(C)(S)\,\{\mathsf{break}(Q,l_2)\}$ can be concluded.

4.2.2 Total break while rule

Next, a proof rule for the total break correctness of the while statement is presented. Suppose there exists a state satisfying $P \wedge C$[6]. Notice that if C did not hold in the initial state, the loop would never terminate abruptly. The aim is to prove that execution of $\mathsf{WHILE}(l_1)(C)(S)$ in this state terminates abruptly, because of a break, resulting in a state satisfying Q. Therefore, it has to be shown that: (1) the iteration body terminates normally only a finite number of times (using a well-founded variant) keeping the condition true; and (2) if the iteration body does not

[6] The use of the (translated) JAVA condition C in here is deliberately sloppy. This C is a Boolean expression, of type $\mathsf{OM} \to \mathsf{ExprResult}[\mathsf{OM}, \mathsf{bool}]$, but occurs in $P \wedge C$, where P is a predicate $\mathsf{OM} \to \mathsf{bool}$. The latter conjunction $\wedge$ in a state $x\colon \mathsf{OM}$ should be understood as: $P(x)$ holds, and $C(x)$ terminates normally, and its result is true.

– TYPE THEORY

$$\frac{\begin{array}{c}\{P\}\,\mathsf{E2S}(C)\,;\mathsf{CATCH\text{-}CONTINUE}(l_1)(S)\,\{P\}\\ \{P\}\,\mathsf{E2S}(C)\,;\mathsf{CATCH\text{-}CONTINUE}(l_1)(S)\,\{\mathsf{break}(Q,l_2)\}\end{array}}{\{P\}\,\mathsf{WHILE}(l_1)(C)(S)\,\{\mathsf{break}(Q,l_2)\}}\ \text{[partial-break]}$$

Fig. 3. Break version of the partial while rule

– TYPE THEORY

$$\frac{\begin{array}{c}[P \wedge C]\,\mathsf{CATCH\text{-}BREAK}(l_2)(\mathsf{E2S}(C)\,;\mathsf{CATCH\text{-}CONTINUE}(l_1)(S))\,[\mathsf{true}]\\ \{P \wedge C \wedge \text{variant} = n\}\,\mathsf{E2S}(C)\,;\mathsf{CATCH\text{-}CONTINUE}(l_1)(S)\,\{P \wedge C \wedge \text{variant} < n\}\\ \{P \wedge C\}\,\mathsf{E2S}(C)\,;\mathsf{CATCH\text{-}CONTINUE}(l_1)(S)\,\{\mathsf{break}(Q,l_2)\}\end{array}}{[P \wedge C]\,\mathsf{WHILE}(l_1)(C)(S)\,[\mathsf{break}(Q,l_2)]}\ \text{[total-break]}$$

Fig. 4. Break version of the total while rule

terminate normally, it must be because of a break, resulting in an abnormal state, satisfying Q. This gives the rule in Fig. 4.

The first condition states that execution of the iteration body followed by a CATCH-BREAK, in a state satisfying $P \wedge C$, always terminates normally, thus the iteration body itself must terminate either normally, or abruptly because of a break. The second condition expresses that if the iteration body terminates normally, the invariant and condition remains true and some variant decreases. Thus, the loop will not terminate normally, because the condition remains true. Since the variant is well-founded the iteration body can only terminate normally a finite number of times. Finally, the last condition of this rule requires that when the iteration body terminates abruptly (because of a break), the resulting state satisfies Q. Soundness of this rule is easy to prove.

5 Class annotations

A behavioural interface specification language for JAVA is proposed in [25], following the tradition of Eiffel and the well-established design by contract approach [31]. This language is called JML, short for JAVA modeling language. A programmer can annotate JAVA code with specifications in JML, using the annotation markers `//@` and `/*@ ... @*/`. For a JAVA compiler these annotations are ordinary comments, so the annotated JAVA code still remains valid. In this paper we shall use JML specifications to express the properties – including the invariant – that we wish to prove about JAVA's `Vector` class.

A behavioural interface specification consists of various specification declarations. Here we will only mention invariants, and pre- and post-conditions for methods and constructors. For more information (see [25]). From a client's perspective the specifications describe properties that can be assumed, but from the provider's perspective they represent proof obligations, because the provided code is supposed to satisfy these properties. Here, we shall take the latter perspective.

5.1 Predicates in JML

The predicates used in JML are built from JAVA expressions extended with logical operators, such as equivalence, `<==>`, and implication, `==>`, and with the existential and universal quantifiers, `\exists` and `\forall`, respectively. In addition, some new expression syntax is added: `\old(`E`)` is used for evaluation of expression E in the "pre-state" of a method (i.e., in the state before method execution is started), `\result` denotes the result of a non-`void` method. These are only used in post-conditions.

Predicates in JML are required to be side-effect free, and, therefore, they are not allowed to contain assignments, including the increment and decrement operators, `++` and `--`. Methods may be invoked in predicates only if they are pure, i.e., terminate normally, and do not modify any field.

Requiring that predicates are side-effect free does not imply that predicates always terminate normally. Consider the predicate `a.length >= 0`, for `a` an array. If this predicate is evaluated in a state where `a` is a null reference, it will terminate abruptly with a `NullPointerException`. To prevent this kind of abrupt termination, an extra conjunct has to be added to the predicate: `a != null && a.length >= 0`.

5.2 Behaviour specifications

In JML behaviour specifications can be written for methods and constructors. Below, we concentrate on methods. JML supports three kinds of behaviour specifications, namely `normal_behavior`, `exceptional_behavior`, and `behavior` specifications. If a method has a `normal_behavior` specification, then it should terminate normally, assuming the pre-condition holds. Similarly, an `exceptional_behavior` prescribes that a method terminates abnormally, and a `behavior` specification that the method can terminate sometimes normally and sometimes abnormally.

Let's consider a `normal_behavior` specification for a method `m`:

– JML
```
/*@ normal_behavior
  @   requires: P;  // P is a predicate
  @   ensures : Q;  // Q is a relation,
  @     // between the method's pre-state
  @     // and post-state.
  @*/
void m();
```

The basic ingredients of `normal_behavior` are its pre-condition, in JML called the `requires` clause, and its post-condition, the `ensures` clause. This `normal_behavior` specification is a total correctness assertion: it says that if `P` holds in a state x, then method `m` executed in state x will terminate normally, resulting in state y with `Q` holding of (x, y). The pre-state x is needed in the post-condition because `Q` may involve an `\old(-)` expression for evaluation in the pre-state.

A `behavior` specification can consist of the two abovementioned clauses, extended with a `signals` clause:

– JML
```
/*@ behavior
  @   requires: P;
  @   ensures : Q;
  @   signals : (E) R;
  @*/
void m();
```

The `signals` clause is the post-condition, in case of abrupt termination of method `m`. This example specification is a conjunction of two partial correctness Hoare sentences. The first one says that if `P` holds in a state x and method `m` executed in state x terminates normally resulting in a state y, then `Q` should hold of (x, y). The second one says that if `P` holds in a state x and method `m` executed in state x terminates abruptly with an exception of type `E'` in a state y, then `R` should hold of (x, y), and `E'` should be a subclass of `E`.

5.3 Invariants

An invariant is a predicate on states which always holds, as far as an outsider can see: an invariant holds immediately after an object is created and before and after a method is executed, but during a method's execution it need not hold. To prove that a certain predicate is an invariant, one therefore proves that it holds: (1) after object creation; and (2) after (normal or abnormal) termination of a method, assuming that it holds when the method's execution starts. Note that even when a method terminates abruptly, an invariant should hold. This means that if something goes wrong, a method must throw an exception before any crucial data is corrupted. A consequence is that if the exception is caught at some later stage, the invariant still holds.

An example of a (trivial) JML invariant is:

– JML
```
class A {
  //@ invariant: true;
  ...
}
```

JML offers the possibility to write multiple invariants within one class. They can be transformed into a single invariant via conjunctions. Invariants can be preceded by the keywords public, private or protected. This means that the invariant property is preserved by all public, private or protected methods, respectively. We verified a private invariant property of `Vector`, i.e., a property that is a closure under all methods.

5.4 Proof obligations

As already mentioned, invariants and behaviour specifications give rise to proof obligations. They can be expressed in our extended Hoare logic. This requires the use of so-called logical variables (such as z below) in order to allow post-conditions to be relations. For example, the `normal_behavior` specification for `m` above, together with an invariant I, yields the following proof obligation for total correctness:

– TYPE THEORY

$$\begin{aligned}\forall z\colon \mathsf{OM}.\, &[\,\lambda x\colon \mathsf{OM}.\, I(x) \wedge P(x) \wedge z = x\,]\\ &m\\ &[\,\lambda y\colon \mathsf{OM}.\, I(y) \wedge Q(z, y)\,]\end{aligned}$$

Similarly, the `behavior` specification yields a conjunction of two partial Hoare sentences:

– TYPE THEORY

$$\begin{aligned}\forall z\colon \mathsf{OM}.\, &\{\,\lambda x\colon \mathsf{OM}.\, I(x) \wedge P(x) \wedge z = x\,\}\\ &m\\ &\{\,\lambda y\colon \mathsf{OM}.\, I(y) \wedge Q(z, y)\,\}\\ &\quad\wedge\\ &\{\,\lambda x\colon \mathsf{OM}.\, I(x) \wedge P(x) \wedge z = x\,\}\\ &m\\ &\{\,\mathsf{exception}(\lambda y\colon \mathsf{OM}.\, I(y) \wedge R(z, y), E)\,\}\end{aligned}$$

In this way the proof rules for the extended Hoare logic can be used to prove JML obligations in PVS.

6 The case study: Java's `Vector` class

6.1 Vector in Java

JAVA's `Vector` class[7] is part of the `java.util` package. It can be found in the sources of the JDK distribution. The class as a whole is too big to describe here in detail. It contains three fields, three constructors, and twenty-five methods. Most of the method bodies consist of between five and ten lines of code. The interface of the `Vector` class, and also its "surrounding" classes in the JAVA library are described. The latter are classes that are used in the `Vector` class.

6.1.1 Interface of the `Vector` class

The `Vector` class has three fields: an array `elementData` of type `Object []` in which the elements of the vector are stored, an integer `elementCount` which holds the number of elements stored in the vector, and an integer `capacityIncrement` which indicates the amount by which the vector is incremented when its size (`elementCount`) becomes greater than its capacity (the length of `elementData`). If `capacityIncrement` is greater than zero, every time the vector needs to grow, the capacity of the vector is incremented by this amount, otherwise the capacity is doubled. These fields are all protected, so that they can only be accessed in (a subclass of) `Vector`.

The `Vector` class has three constructors, which all are public and thus can be accessed in any class. The

[7] For our verification we use version number 1.38, written by Lee Boynton and Jonathan Payne, under Sun Microsystems copyright.

constructor `Vector()` creates an instance of the `Vector` class by allocating the array `elementData` with an initial capacity of ten elements, and a capacity increment of zero. The second constructor `Vector(int initialCapacity)` takes an integer argument, which is the initial capacity, and sets the capacity increment to zero. The third constructor `Vector(int initialCapacity, int capacityIncrement)` takes two integer arguments, one for the initial capacity and the other for the capacity increment. After creating an instance of the `Vector` class the field `elementCount` is implicitly set to zero.

Space restrictions prevent us from describing all methods of the `Vector` class in detail. Therefore, the reader is referred to the standard documentation [10] for more information, and only the interface of the `Vector` class is listed here, (see Fig. 5). The names and types give some idea of what these methods are supposed to do.

6.1.2 Surrounding classes

The `Vector` class implicitly extends the `Object` class. All fields and methods declared in the `Object` class are

– JAVA

```java
public class Vector implements Cloneable, java.io.Serializable {
   // fields
   protected Object elementData[];
   protected int elementCount;
   protected int capacityIncrement;

   // constructors
   public Vector(int initialCapacity, int capacityIncrement);
   public Vector(int initialCapacity);
   public Vector();

   // methods
   public final synchronized void copyInto(Object anArray[]);
   public final synchronized void trimToSize();
   public final synchronized void ensureCapacity(int minCapacity);
   private void ensureCapacityHelper(int minCapacity);
   public final synchronized void setSize(int newSize);
   public final int capacity();
   public final int size();
   public final boolean isEmpty();
   public final synchronized Enumeration elements();
   public final boolean contains(Object elem);
   public final int indexOf(Object elem);
   public final synchronized int indexOf(Object elem, int index);
   public final int lastIndexOf(Object elem);
   public final synchronized int lastIndexOf(Object elem, int index);
   public final synchronized Object elementAt(int index);
   public final synchronized Object firstElement();
   public final synchronized Object lastElement();
   public final synchronized void setElementAt(Object obj, int index);
   public final synchronized void removeElementAt(int index);
   public final synchronized void insertElementAt(Object obj, int index);
   public final synchronized void addElement(Object obj);
   public final synchronized boolean removeElement(Object obj);
   public final synchronized void removeAllElements();
   public synchronized Object clone();
   public final synchronized String toString();
}
```

Fig. 5. The interface of Java's `Vector` class

thus inherited. Of particular importance in the `Vector` class are the methods `equals`, `clone`, and `toString` from `Object`. These may be overridden in particular instantiations of the data in a vector (and the new versions are then selected via the "dynamic method look-up" or "late binding" mechanism). The `Vector` class also implements two (empty) JAVA interfaces, namely `Cloneable` and `Serializable`.

The following JAVA classes are used in the `Vector` class, in one way or another: `CloneNotSupportedException`, `InternalError`, `Object`, `StringBuffer`, `String`, `System`, `ArrayIndexOutOfBoundsException` (all from the `java.lang` package), `Enumeration`, `NoSuchElementException` (both from the package `java.util`), and `Serializable` (from the `java.io` package). These additional classes are relevant for the verification, since they also have to be translated into PVS. They are intertwined via mutual recursion.

6.2 Translation of Vector into PVS

The LOOP tool translates JAVA classes into logical theories for PVS, following the semantics as described before. In this section, some aspects of the actual translation of the `Vector` class are briefly discussed. For this project, it is not needed to translate the whole JAVA library. Only those classes that are actually used in the `Vector` class – called the "surrounding" classes – have to be translated. A further reduction has been applied: from these surrounding classes, only those methods that are actually needed have been translated. Thus, 10K of JAVA code remains, excluding documentation. The LOOP tool turns it into about 500K of PVS code[8].

JAVA's `Object` and `System` classes have several native methods. A native method lets a programmer use some already existing (non-JAVA) code, by invoking it from within JAVA. In the `Vector` class two native methods are used, namely `clone` from `Object`, and `arraycopy` from `System`. Our own PVS code has been inserted as translation of the method bodies of these native methods. An alternative approach would be to use requirements (e.g., JML specifications) for these methods, such as for `toString` and `equals` (see the next section).

The current version of our LOOP tool handles practically all of "sequential" JAVA, i.e., of JAVA without threads. The possible use of vectors in a concurrent scenario is not relevant for this invariant verification. The `synchronized` keyword in the method declarations is therefore simply ignored.

There is one point where we have cheated a bit in the `Vector` translation. Often in the `Vector` class an exception is thrown with a message, such as in the following code fragment:

– JAVA
```
public final synchronized Object
     elementAt(int index) {
  if (index >= elementCount) {
    throw new ArrayIndexOutOfBoundsException
            (index + " >= " + elementCount);
  }
  ...
}
```

Implicitly in JAVA, the integers `index` and `elementCount` are converted to strings in the exception message. Such conversion is not available in PVS. Of course, it can be defined, but that is cumbersome and totally irrelevant for the invariant. Therefore, we have reduced such exception messages in `throw` clauses to the empty string `""`, thereby avoiding the conversion issue altogether. This affects the output, but not the invariant.

6.3 The class invariant

The first step is to formulate the desired class invariant property. Finding an appropriate, provable, invariant is, in general, a non-trivial exercise. Usually one starts with some desired property, but to be able to prove that this is an invariant, it has to be strengthened in an appropriate manner[9]. As suggested by the informal documentation in the `Vector` class, a class invariant should be:

> the number of elements in the array of a vector object never exceeds its capacity.

This property alone cannot be proven to be a class invariant. Strengthening is necessary to obtain an actual invariant. This invariant has been obtained "by hand", and not via some form of automatic invariant generation. Precisely annotating all the methods in the `Vector` class with JML-specifications helps in finding the appropriate strengthening, because it brings forward the pre-conditions for normal and abrupt terminations. The strengthened version of the above property can be extracted from these pre-conditions for normal termination. During verification it turned out that the resulting property had to be strengthened only once more (in a very subtle manner). In JML, the main ingredients of the invariant are listed in Fig. 6.

One more requirement is needed that is directly related to the particular memory model that we use (see Sect. 3.2), and is not expressible in JML. It says that `elementData` refers to an "allocated" cell in the heap memory, whose position is below the `heaptop`.

[8] This may seem a formidable size multiplication, but it does not present problems in verification. Reductions in size may still be possible by making more efficient use of parametrisation in PVS code generation.

[9] This is in analogy with the informal notion of "induction loading", where a statement that one wishes to prove by induction must be strengthened in order to get the induction going.

```
– JML
  /*@ invariant:
    @   elementData != null  &&
    @   elementCount <= elementData.length  &&   // main point
    @   elementCount >= 0  &&
    @   elementData != this  &&
    @   elementData instanceof Object[]  &&
    @   (\forall (int i) 0 <= i && i < elementData.length
    @                     ==> (elementData[i] == null || elementData[i] instanceof Object));
    @*/
```

Fig. 6. Essentials of the `Vector` invariant

The resulting combined property on OM will be called VectorIntegrity?. Notice that it says nothing about the value of the `capacityIncrement` field. One would expect the value of `capacityIncrement` to be positive, but this is not needed, since the only time `capacityIncrement` is actually used (in the body of the method `ensureCapacityHelper`), it is first tested whether its value is greater than zero. The informal documentation for this field states that "if the capacity increment is 0, the capacity of the vector is doubled each time it needs to grow", but a more precise statement would be "if the capacity increment is 0 *or less*, ...".

6.4 Verification of the class invariant of Vector

After translation of the `Vector` class (and all surrounding classes), the generated theories are loaded into PVS and the verification effort starts. This means that we have to show that the predicate VectorIntegrity? is indeed an invariant. To this end, it has to be shown that: (1) VectorIntegrity? is established by the constructors; and (2) that VectorIntegrity? is preserved by all public methods of class `Vector` (see Sect. 5.3). Notice that it is essential that the fields of the `Vector` class are protected, so that they cannot be accessed directly from the outside, and the VectorIntegrity? predicate cannot be corrupted in this manner.

Point (1) is relatively easy. Point (2) is handled by assuming an arbitrary state x, satisfying VectorIntegrity?; for each method m, say with arguments $\vec{a}$, the cases where $m(\vec{a})(x)$ terminates normally, and where it throws an exception are distinguished. This is done via JML behaviour specifications. In all the cases, it has to be shown that the predicate VectorIntegrity? still holds in the resulting state, (see Sect. 5.4).

Before going into some proof details, we illustrate that detecting all possible exceptions is a non-trivial, but useful exercise. Therefore, we consider in Fig. 7 a fragment from the `Vector` class, which describes the method `copyInto` together with its informal documentation.

This method throws an exception in each of the following cases:

– the field `elementCount` is greater than zero, and the argument array `anArray` is a null reference;
– `elementCount` is greater than zero, `anArray` is a non-null reference, and its length is less than `elementCount`;
– `elementCount` is greater than zero, `anArray` is a non-null reference, its length is at least `elementCount`, and there is an index `i` below `elementCount` such that the (run-time) class of `elementData[i]` is not assignment compatible with the (run-time) class of `anArray`.

The first of these three cases produces a `NullPointerException`, the second one an `ArrayIndexOutOfBoundsException`, the third one an `ArrayStoreException`[10]. This last case is subtle, and not documented at all; it can easily be overlooked. However, in all three cases, no data in `Vector` is corrupted, and the predicate VectorIntegrity? still holds in the resulting (abnormal) state.

Below, the verification in PVS of several methods will be discussed in some detail, namely of `setElementAt`, `toString`, and `indexOf`. These methods are exemplary: the method `setElementAt` is a typical example of a method for which the invariant is verified automatically. The verification of `toString` shows how we deal with late binding and `indexOf` demonstrates the use of the extended Hoare logic for JAVA. The verifications make extensive use of automatic rewriting to increase the level of automation. For instance, the low-level memory manipulations (involving the get- and put-operations from Sect. 3.2) require no user interaction at all. Automatic rewriting is also very useful in verifications using Hoare logic, because it simplifies the application of the rules.

Verification of setElementAt

The first method that is discussed in more detail is `setElementAt`. This method takes a parameter `obj` belong-

[10] See the explanation in [12], Sect. 15.25.1, second paragraph on page 371. This exception occurs for example during execution of the following (compilable, but silly) code fragment.

```
Vector v = new Vector();
v.addElement(new Object());
v.copyInto(new Integer[1]);
```

– JAVA
```
/**
 * Copies the components of this vector into the specified
 * array. The array must be big enough to hold all the
 * objects in this vector.
 *
 * @param   anArray   the array into which the components get copied.
 * @since   JDK1.0
 */
public final synchronized void copyInto(Object anArray[]) {
    int i = elementCount;
    while (i-- > 0) { anArray[i] = elementData[i]; }
}
```

Fig. 7. The `copyInto` method from `Vector` with its informal documentation

ing to class `Object` and an integer `index`, and replaces the element at position `index` in the vector with `obj`. A possible JML specification for this method is given in Fig. 8.

Notice that we have given a "functional" specification by describing post-conditions for this method. These post-conditions can be strengthened further, e.g., by including that the fields `elementCount` and `capacityIncrement` are not changed. However, for our invariant verification, these post-conditions are usually not relevant, and so we shall simply write `true` in the `ensures:` clause, giving so-called lightweight specifications (such as in [37]). In contrast, the pre-conditions are highly relevant.

Ignoring the post-conditions, the proof obligations (as Hoare sentences, see Sect. 5.4) for this method are:

– TYPE THEORY

$$\begin{aligned}
&\forall \mathsf{obj}\colon \mathsf{RefType}.\, \forall \mathsf{index}\colon \mathsf{int}.\\
&\quad [\, \lambda x\colon \mathsf{OM}.\, \mathsf{VectorIntegrity?}(x) \wedge\\
&\qquad\qquad \mathsf{index} \geq 0 \wedge\\
&\qquad\qquad \mathsf{index} < \mathsf{elementCount}(x)\,]\\
&\quad \mathsf{setElementAt}(\mathsf{obj}, \mathsf{index})\\
&\quad [\, \mathsf{VectorIntegrity?}\,]
\end{aligned}$$

$$\begin{aligned}
&\forall \mathsf{obj}\colon \mathsf{RefType}.\, \forall \mathsf{index}\colon \mathsf{int}.\\
&\quad [\, \lambda x\colon \mathsf{OM}.\, \mathsf{VectorIntegrity?}(x) \wedge\\
&\qquad\qquad (\mathsf{index} < 0 \vee\\
&\qquad\qquad \mathsf{index} \geq \mathsf{elementCount}(x))\,]\\
&\quad \mathsf{setElementAt}(\mathsf{obj}, \mathsf{index})\\
&\quad [\, \mathsf{exception}(\mathsf{VectorIntegrity?},\\
&\qquad\qquad \text{“}\texttt{ArrayIndexOutOfBoundsException}\text{”})\,]
\end{aligned}$$

The proofs of these properties proceed mainly by automatic rewriting in PVS. For the first proof obligation, regarding normal termination, we do explicitly have to make the case distinction whether the argument obj is a null reference or not.

Verification of toString

Unfortunately, the correctness of the methods in `Vector` is not always as easy to prove as for the above example `setElementAt`. Several methods in the `Vector` class invoke other methods, or contain `while` or `for` loops. Above, we already have seen `copyInto` as an example of such a method. We now concentrate on the method invocations in `Vector`'s `toString` method.

Recall that each class in JAVA inherits the `toString` method from the root class `Object`. In a specific class this method is usually overridden to give a suitable string representation for instances of that class. For a vector object the `toString` method in the `Vector` class yields a string representation of the form $[\, s_0, \ldots, s_{n-1}\,]$, where n is the vector's size `elementCount`, and s_i is the string obtained by applying the `toString` method to the ith element in the vector's array. The particular implementations that get executed as a result of these `toString` invocations are determined by the actual (run-time) types of the elements in the array (via the late binding mechanism). Thus, they cannot be determined statically. This is a key issue in object-oriented verification.

The annotated JAVA code of `toString` in `Vector` is given in Fig. 9. It reveals an undocumented possible source of abrupt termination: when one of the elements of a vector's array is a null reference, invoking `toString` on it yields a `NullPointerException`.

The "behavioural subtyping" approach to late binding that we take here, following [30], involves writing down requirements on the method `toString` in `Object` and using these requirements in reasoning. In our verification, we thus assume that the definition of `toString` that is actually used at run-time satisfies these requirements, i.e., that the classes of the component objects are a behavioural subtypes of `Object`. Thus, we prove that `toString` in `Vector` works correctly, assuming that we have a reasonable implementation of `toString`, without unexpected behaviour.

```
– JML
  /*@
    @ normal_behavior
    @   requires: index >= 0 && index < elementCount;
    @   ensures: \forall (int i) 0 <= i && i < elementCount ==>
    @     ((i == index && elementData[i] == obj) || (i != index && elementData[i] ==
    @                                                              \old(elementData[i])));
    @ also
    @ exceptional_behavior
    @   requires: index < 0 || index >= elementCount;
    @   signals: (ArrayIndexOutOfBoundsException)
    @     \forall (int i) 0 <= i && i < elementCount ==> elementData[i] == \old(elementData[i];
    @*/
  public final synchronized void setElementAt(Object obj, int index) {
    if (index >= elementCount) { throw new ArrayIndexOutOfBoundsException(index +
                                                                 " >= " + elementCount); }
    elementData[index] = obj;
  }
```

Fig. 8. The `setElementAt` method with its JML annotation

```
– JML
  /*@
    @ normal_behavior
    @   requires: \forall (int i) 0 <= i && i < elementCount ==> elementData[i] != null;
    @   ensures: true;
    @ also
    @ exceptional_behavior
    @   requires: elementCount > 0 &&
    @             ! \forall (int i) 0 <= i && i < elementCount ==> elementData[i] != null;
    @   signals: (NullPointerException) true;
    @*/
  public final synchronized String toString() {
    int max = size() - 1;
    StringBuffer buf = new StringBuffer();
    Enumeration e = elements();
    buf.append("[");
    for (int i = 0 ; i <= max ; i++) {
      String s = e.nextElement().toString();
      buf.append(s);
      if (i < max) { buf.append(", "); }
    }
    buf.append("]");
    return buf.toString();
  }
```

Fig. 9. The `toString` method with its JML annotation

In ordinary language, the requirements on `toString` say that:

– it terminates normally, and has no side-effects;
– it returns a non-null reference to a memory location in newly allocated memory, i.e., between the heaptop in the pre-state and the heaptop in the post-state (the state after execution of `toString`);
– this reference has run-time type `String`, and points to a memory cell with integer fields `offset` and `count` (from class `String`), which are non-negative, and an array field `value` (also from `String`), which

– is a non-null reference with a cell position which is above the heaptop in the pre-state, below the heaptop in the post-state, and different from the previously mentioned `String` reference;
– has run-time elementtype `char` and a length exceeding the sum of `offset` and `count`.

The verification of the `toString` method from `Vector` is then not difficult, but very laborious. This is because it uses (indirectly via `append` from `StringBuffer`) several different methods from other classes, such as `extendCapacity` from `StringBuffer`, and `getChars`, `valueOf` from `String`. For all these methods appropriate "modifiable" results – describing which cells and positions therein can be modified – are needed to prove that the methods do not affect the VectorIntegrity? predicate.

Verification of indexOf

Next, we consider the verification of a `for` loop, namely in the method `indexOf`. It makes extensive use of the Hoare logic rules as described in Sect. 4.

First we consider the specification and implementation of `indexOf` in Fig. 10. The method `indexOf` takes a parameter `elem` belonging to class `Object` and an integer parameter `index`, and checks whether `elem` occurs in the segment of the vector between `index` and `elementCount`. If so, the position at which it occurs is returned, otherwise -1 is returned.

Notice that the `equals` method in the condition of the `if` statement is invoked on the parameter `elem`. Since we cannot know `elem`'s run-time type, we also have to use the behavioural subtype approach here, and assume that certain requirements hold for `equals`, such as for `toString` in the previous example. We shall not elaborate on this point, but concentrate on the `for` loop.

To show that `indexOf` maintains VectorIntegrity?, several cases are distinguished. If the parameter `elem` is non-null and `index` is non-negative, the Hoare logic rules for abruptly terminating loops, as described in Sect. 4, are needed for the verification. A distinction is made between the case that `elem` is found, and that it is not found (because in the first case the for loop terminates abruptly, because of a return, and in the second case it terminates normally, thus different rules have to be used). In both cases it is shown that the method preserves VectorIntegrity?. To this end, the rule in Fig. 11 for total return correctness of a `for` loop, is used[11].

Notice the similarity with the rule for total break correctness of the `while` statement, as described in Fig. 4. However, the `for` loop has a different iteration body, namely $\mathsf{E2S}(C)\,;\mathsf{CATCH\text{-}CONTINUE}(l)(S)\,;U$, where U is the formalisation of the update statement of the `for` loop. Recall that for `while` loops the iteration body is $\mathsf{E2S}(C)\,;\mathsf{CATCH\text{-}CONTINUE}(l)(S)$.

This rule is instantiated as follows:

l	bot
C	$\llbracket$`i < elementCount`$\rrbracket$
U	$\llbracket$`i++`$\rrbracket$
S	$\llbracket$`if (elem.equals(elementData[i])) {return i;}`$\rrbracket$
variant	$\llbracket$`elementCount - i`$\rrbracket$
P	$\lambda x\colon \mathsf{OM}.\ \mathsf{VectorIntegrity?}\,(x) \wedge \mathtt{i} \geq \mathtt{index} \wedge \mathtt{i} \leq \mathtt{elementCount} \wedge (\exists j.\mathtt{index} \leq j < \mathtt{elementCount} \wedge j \geq \mathtt{i} \wedge \mathtt{elem.equals(elementData[}j\mathtt{])}) \wedge (\forall k.\mathtt{index} \leq k < \mathtt{i} \supset \neg\mathtt{elem.equals(elementData[}k\mathtt{])})$
Q	VectorIntegrity?

Notice that the loop invariant (P) implies that the condition `i` $<$ `elementCount` remains true, because if `i` would be equal to `elementCount`, the last two clauses of the invariant would be contradicting.

In the case that `elem` is not found in the vector, the rule for total (normal) correctness of the `for` loop is used, with a similar instantiation, to show that in that case the loop always terminates normally (returning -1).

In the case that `index` $\geq$ `elementCount`, or in the case of abrupt termination (i.e., `index` < 0 or `elem` is a null reference), it can be shown that the condition of the `for`-loop immediately evaluates to false or throws an exception, respectively. Since no changes are made to the fields of `Vector`, the property VectorIntegrity? is preserved.

Actually, we have proved a bit more about the `indexOf` method than stated here. More is needed because the method is used in another `Vector` method, namely in `contains`. With these stronger results, the `contains` method can be verified by automatic rewriting in PVS. In this case, late binding cannot occur because the `indexOf` method is declared as `final`, so that it cannot be overridden.

7 Conclusions and experiences

We have formally proved with PVS a non-trivial safety property for the `Vector` class from JAVA's standard library. The verification is based on careful (lightweight) specifications of all `Vector` methods, using the experimental behavioural interface specification language JML. It makes many non-trivial and poorly documented (normal and abnormal) termination conditions explicit (see also [43]).

[11] The JAVA statement `for(init, cond, update){body}` is translated into $\llbracket$`init`$\rrbracket$; FOR($\llbracket$`cond`$\rrbracket$, $\llbracket$`update`$\rrbracket$, $\llbracket$`body`$\rrbracket$), where FOR is defined in type theory similar to WHILE (see [21]). The proof rule only deals with this FOR function.

– JML
```
/*@
  @ normal_behavior
  @   requires: index >= elementCount || (elem != null && index >= 0);
  @   ensures: true;
  @ also
  @ exceptional_behavior
  @   requires: elem == null && index < elementCount;
  @   signals: (NullPointerException) true;
  @ also
  @ exceptional_behavior
  @   requires: elem != null && index < 0;
  @   signals: (ArrayIndexOutOfBoundsException) true;
  @*/
public final synchronized int indexOf(Object elem, int index) {
    for (int i = index; i < elementCount; i++) {
        if (elem.equals(elementData[i])) { return i; } }
    return -1;
}
```

Fig. 10. The `indexOf` method with its JML annotation

– TYPE THEORY

$$\frac{\begin{array}{c}[P \wedge C]\,\mathsf{CATCH\text{-}STAT\text{-}RETURN}(\mathsf{E2S}(C)\,;\mathsf{CATCH\text{-}CONTINUE}(l)(S)\,;U)\,[\mathsf{true}]\\ \{P \wedge C \wedge \mathrm{variant} = n\}\,\mathsf{E2S}(C)\,;\mathsf{CATCH\text{-}CONTINUE}(l)(S)\,;U\,\{P \wedge C \wedge \mathrm{variant} < n\}\\ \{P \wedge C\}\,\mathsf{E2S}(C)\,;\mathsf{CATCH\text{-}CONTINUE}(l)(S)\,;U\,\{\mathsf{return}(Q)\}\end{array}}{[P \wedge C]\,\mathsf{FOR}(l)(C)(U)(S)\,[\mathsf{return}(Q)]}\;[\text{total-return}]$$

Fig. 11. Return version of the total for rule

The whole invariant verification presented here was a lot of work. In total, it involved 13,193 proof commands (atomic interactions) in PVS. Some methods required only a few proof commands – and could be verified entirely by automatic rewriting – but others required more interaction. The `toString` method was most labour intensive, requiring 4,922 proof commands, about one-third of the total number. Quantifying the time it took is more difficult, because much of the work was done for the first time in such a large project, and could be done faster given more experience. However, 3–4 months full-time work (for a single, experienced person) seems a reasonable estimate.

In the end one should ask: is it worthwhile to do these kinds of formal specifications and verifications, and do they scale up? We think that writing (lightweight) specifications (even without formal verifications) is certainly worthwhile, because it can make many implicit assumptions explicit, at relatively little cost. Such specifications facilitate the use of the code and increase the reliability. Extended static checking of such specifications is becoming possible [11]. Actual verification of the specifications is far more labour intensive. It may be worthwhile to do this for library classes (such as `Vector`) which are intensively used, but not for classes which are specific for a particular application. However, the entire JAVA class library has become so large that it would be an unrealistically large investment to fully verify it in the way that we have done for one single class. However, it may still be worthwhile to do this for certain central and crucial parts of the library.

On the basis of the experiences in this project we have chosen to concentrate next on the JAVACARD [23] class library [37]. It is much smaller (about 45 classes), and is used in smaller applications (namely JAVACARD applets, which are small programs for smart cards with modest resources). There, both specification and verification are more easily justified, not only because of the smaller investment due to smaller size, but also because there is a great need for reliability in this area, since smart cards are being used in large numbers in often security sensitive environments.

References

1. Abadi, M., Leino, K.R.M.: A logic of object-oriented programs. In: Bidoit, M., Dauchet, M. (eds.): Theory and practice of software development (TAPSOFT '97). LNCS 1214. Berlin, Heidelberg, New York: Springer-Verlag, 1997, pp. 682–696
2. Alves-Foss, J. (ed.): In: Formal syntax and semantics of Java. LNCS 1523. Berlin, Heidelberg, New York: Springer-Verlag, 1999
3. Apt, K.R., Olderog, E.-R.: Verification of Sequential and Concurrent Programs. Berlin, Heidelberg, New York: Springer-Verlag, 2nd. rev. edn., 1997
4. Apt, K.R.: Ten years of Hoare's logic: a survey–part I. ACM Trans. Program. Lang. Syst. 3(4): 431–483, 1981
5. Arnold, K., Gosling, J.: The Java Programming Language. Addison-Wesley, Reading, Mass., USA, 2nd. edn., 1997
6. de Bakker, J.W.: Mathematical Theory of Program Correctness. Prentice Hall, Englewood Cliffs, N.J., USA, 1980
7. van den Berg, J., Huisman, M., Jacobs, B., Poll, E.: A type-theoretic memory model for verification of sequential Java programs. In: Bert, D., Choppy, C., Mosses, P.D. (eds.): Recent trends in algebraic development techniques. LNCS 1827. Berlin, Heidelberg, New York: Springer-Verlag, 2000, pp. 1–21
8. de Boer, F.S.: A WP-calculus for OO. In: Thomas, W. (ed.): Foundations of software science and computation structures. LNCS 1578. Berlin, Heidelberg, New York: Springer-Verlag, 1999, pp. 135–149
9. Christian, F.: Correct and robust programs. IEEE Trans. Software Eng. 10(2): 163–174, 1984
10. Chan, P., Lee, R., Kramer, D.: The Java Class Libraries, vol. 1. Addison-Wesley, Reading, Mass., USA, 2nd. edn., 1998
11. Extended static checker ESC/Java: Compaq System Reserch Center, `http:// www.research.digital.com / SRC / esc / Esc.html`
12. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, Second Edition. In: The Java Series. Addison-Wesley, Reading, Mass., USA, 2000
13. Gordon, M.J.C.: Programming Language Theory and Its Implementation. Prentice Hall, Englewood Cliffs, N.J., USA, 1988
14. Gordon, M.J.C.: Mechanizing programming logics in higher order logic. In: Birtwistle, G.M., Subrahmanyam, P.A. (eds.): Current Trends in Hardware Verification and Automated Theorem Proving. Berlin, Heidelberg, New York: Springer-Verlag, 1989
15. Gries, D.: The Science of Programming. Berlin, Heidelberg, New York: Springer-Verlag, 1981
16. Hartel, P.H., Butler, M.J., Levy, M.: The operational semantics of a Java Secure Processor. In: [2], pp. 313–352
17. Hensel, U., Huisman, M., Jacobs, B., Tews, H.: Reasoning about classes in object-oriented languages: logical models and tools. In: Hankin, C. (ed.), Proc. European Symposium on Programming (ESOP '98). LNCS 1578. Berlin, Heidelberg, New York: Springer-Verlag, 1998, pp. 105–121
18. Huisman, M., Jacobs, B.: Inheritance in higher order logic: modeling and reasoning. In: Harrison, J., Aagaard, M. (eds.): Theorem proving in higher order logics: 13th International Conference (TPHOLs 2000). LNCS 1869. Berlin, Heidelberg, New York: Springer-Verlag, 2000, pp. 301–319
19. Huisman, M., Jacobs, B.: Java program verification via a Hoare logic with abrupt termination. In: Maibaum, T. (ed.), Fundamental approaches to software engineering (FASE 2000). LNCS 1783. Berlin, Heidelberg, New York: Springer-Verlag, pp. 284–303
20. Hoare, C.A.R.: An axiomatic basis for computer programming. Comm. ACM 12: 576–583, 1969
21. Huisman, M.: Reasoning about Java programs in higher-order logic, using PVS and Isabelle/HOL. Forthcoming PhD thesis, Univ. of Nijmegen, 2001
22. Jacobs, B.: A formalisation of Java's exception mechanism. Techn. Rep. CSI-R0015, Computing Science Inst., Univ. of Nijmegen, 2000
23. The Java Card 2.1 Application Programming Interface (API): Sun Microsystems, 1999. URL: `http:// java.sun.com / products / javacard / htmldoc/`
24. Jacobs, B., van den Berg, J., Huisman, M., van Berkum, M., Hensel, U., Tews, H.: Reasoning about classes in Java (preliminary report). In: Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '98), ACM, 1998, pp. 329–340
25. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. Techn. Rep. 98-06, Dep. Computer Science, Iowa State Univ., 1998
26. Leino, K.R.M.: Toward Reliable Modular Programs. PhD thesis, California Inst. of Technol., 1995
27. Leino, K.R.M.: Data groups: specifying the modification of extended state. In: Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '98), ACM, 1998, pp. 144–153
28. The LOOP project, Computing Science Institute, University of Nijmegen: URL: `http:// www.cs.kun.nl / ~bart / LOOP / index.html`
29. Leino, K.R.M., van de Snepscheut, J.: Semantics of exceptions. In: Olderog, E.-R. (ed.), Programming Concepts, Methods and Calculi, North-Holland, 1994, pp. 447–466
30. Liskov, B.H., Wing, J.M.: A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. 16(1): 1811–1841, 1994
31. Meyer, B.: Object-Oriented Software Construction. Prentice Hall, Englewood Cliffs, N.J., 2nd. rev. edn., 1997
32. von Oheimb, D.: Axiomatic semantics for Java$^{\ell ight}$. In: Drossopoulou, S., Eisenbach, S., Jacobs, B., Leavens, G.T., Müller, P., Poetzsch-Heffter, A. (eds.): Formal Techniques for Java Programs, Techn. Rep. 269 - 5/2000 FernUniversität Hagen, 2000, pp. 88–95
33. von Oheimb, D., Nipkow, T.: Machine-checking the Java specification: proving type-safety. In: [2] pp. 119–156
34. Owre, S., Rajan, S., Rushby, J.M., Shankar, N., Srivas, M.K.: PVS: Combining specification, proof checking, and model checking. In: Alur, R., Henzinger, T.A. (eds.): Computer-aided verification (CAV '96). LNCS 1102. Berlin, Heidelberg, New York: Springer-Verlag, 1996, pp. 411–414
35. Owre, S., Rushby, J., Shankar, N., von Henke, F.: Formal verification for fault-tolerant architectures: prolegomena to the design of PVS. IEEE Trans. Software Eng. 21(2): 107–125, 1995
36. Paulson, L.C.: Isabelle – a generic theorem prover. In: LNCS 828. Berlin, Heidelberg, New York: Springer-Verlag, 1994
37. Poll, E., van den Berg, J., Jacobs, B.: Specification of the JavaCard API in JML. In: Domingo-Ferrer, J., Chan, D., Watson, A. (eds.): Fourth Smart Card Research and Advanced Application Conference (IFIP Cardis 2000), Kluwer Academic, 2000, pp. 135–154
38. Poetzsch-Heffter, A., Müller, P.: A programming logic for sequential Java. In: Swierstra, S.D.(ed.), Programming languages and systems (ESOP '99). LNCS 1576. Berlin, Heidelberg, New York: Springer-Verlag, 1999, pp. 162–176
39. Pusch, C.: Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. In: Cleaveland, W.R.(ed.), Tools and algorithms for the construction and analysis of systems (TACAS '99). LNCS 1578. Berlin, Heidelberg, New York: Springer-Verlag, 1999, pp. 89–103
40. Qian, Z.: A formal specification of Java$^{\mathrm{TM}}$ Virtual Machine instructions for objects, methods and subroutines. In: [2] pp. 271–311
41. Reynolds, J.C.: Theories of Programming Languages. Cambridge University, Cambridge, UK, 1998
42. Syme, D.: Proving Java type soundness. In: [2] pp. 83–118
43. `Vector` class (copyright Sun Microsystems, version 1.38, 1997), with JML annotations: Loop web pages, URL: `http:// www.cs.kun.nl / ~bart / LOOP/ Vector_annotated.java`