Handling Java's Abrupt Termination in a Sequent Calculus for Dynamic Logic

Bernhard Beckert and Bettina Sasse

University of Karlsruhe
Institute for Logic, Complexity and Deduction Systems
D-76128 Karlsruhe, Germany
beckert@ira.uka.de, sasse@ira.uka.de

Abstract. In JAVA, the execution of a statement can terminate abruptly (besides terminating normally and terminating not at all). Abrupt termination either leads to a redirection of the control flow after which the program execution resumes (for example if an exception is caught), or the whole program terminates abruptly (if an exception is not caught). Within the KeY project, a Dynamic Logic for Java Card as well as a sequent calculus for that logic have been developed and implemented, which can be used to verify JAVA CARD programs. In this paper, we describe how abrupt termination is handled in that calculus. The ideas behind the rules we present can easily be adapted to other program logics (in particular Hoare logic) for JAVA.

1 Introduction

In JAVA, the execution of a statement can terminate *abruptly* (besides terminating normally and terminating not at all). Possible reasons for an abrupt termination are for instance (a) that an exception has been thrown, (b) that a loop or a single loop iteration is terminated with the `break` resp. the `continue` statement, and (c) that the execution of a method is terminated with the `return` statement. Abrupt termination of a statement either leads to a redirection of the control flow after which the program execution resumes (for example if an exception is caught), or the whole program terminates abruptly (if an exception is not caught).

In [2], a Dynamic Logic for JAVA CARD (JAVA CARD DL) has been presented, as well as the basic rules of a sequent calculus for JAVA CARD DL that can be used to verify JAVA CARD programs (it has been implemented as part of the KeY system, see below). In this paper, we give a detailed description of how abrupt termination is handled in that calculus. The basic principles of the rules we present can easily be adapted to other program logics (in particular Hoare logic) for JAVA.

The basic idea of our approach, which helps to keep the calculus's rules simple, is to give an *abruptly* terminating statement the same semantics as that of a *non-terminating* statement. As usual in Dynamic Logics, the semantics of a program is a partial functions between states. Neither the fact that an abrupt termination has occurred nor the reason for the abrupt termination are made part of the states. Thus, to define the semantics of DL formulas, we do not need to provide additional constructs for handling abrupt termination. Nevertheless, our calculus can handle programs that make use of abrupt termination to redirect control flow during execution.

We work according to the principle that the program states should not include information about control flow: they do not contain a program counter, nor the value of the condition in an `if-else` statement that has just been evaluated, *nor the reason for the termination of a statement.*

A different approach is used in [3], where the semantics of a program is not a function between states but from states to pairs consisting of a state and a reason for termination, making the reason for completion effectively part of the final state of a statement. Other related work includes [6] and [8], where program logics for (subsets of) JAVA are described.

The structure of this paper is as follows: In Section 2, we shortly describe the background and motivation of our work. Syntax and semantics of JAVA CARD DL are introduced in Section 3; for details, the reader is referred to [2]. The rules for handling abrupt termination are given in Section 4. In Section 5, we present an example for the application of these rules.

2 Background

The work reported here has been carried out as part of the KeY project [1]. The goal of KeY is to enhance a commercial CASE tool with functionality for formal specification and deductive verification and, thus, to integrate formal methods into real-world software development processes. Accordingly, the design principles for the software verification component of the KeY system are:

- The programs that are verified should be written in a "real" object-oriented programming language (we decided to use JAVA CARD).
- The logical formalism should be as easy as possible to use for software developers (who do not have years of training in formal methods).

The first prototype of the KeY system—as well as additional information on KeY—is available at i12www.ira.uka.de.[1]

Since JAVA CARD is a "real" object-oriented language, it has features which are difficult to handle in a software verification system, such as dynamic binding, aliasing, object initialisation, and—the topic of this paper—abrupt termination. On the other hand, JAVA CARD lacks some crucial complications of the full JAVA language such as threads and dynamic loading of classes. Moreover, JAVA smart cards are an extremely suitable target for software verification, as the applications are typically security-critical but rather small.

[1] A version of the KeY system that includes an implementation of the calculus described here is available from the authors (at the time of writing, this implementation is not yet part of the publicly available version of KeY).

We use an instance of Dynamic Logic (DL) [5]—which can be seen as an extension of Hoare logic—as the logical basis of the KeY system's software verification component, because deduction in DL is based on symbolic program execution and simple program transformations and is close to a programmer's understanding of JAVA CARD. Also, DL has successfully been applied in practice to verify software systems of considerable size. It is used in the software verification systems KIV [7] and VSE [4] (for a programming language that is not object-oriented).

3 Dynamic Logic for Java Card

3.1 Overview

Dynamic Logic can be seen as a modal predicate logic with a modality $\langle p \rangle$ for every program p (we allow p to be any sequence of legal JAVA CARD statements); $\langle p \rangle$ refers to the successor worlds (called states in the DL framework) that are reachable by running the program p. In standard DL there can be several of these states (worlds) because the programs can be non-deterministic; but here, since JAVA CARD programs are deterministic, there is exactly one such world (if p terminates) or there is no such world (if p does not terminate). The formula $\langle p \rangle\phi$ expresses that the program p terminates in a state in which ϕ holds. A formula $\phi \to \langle p \rangle\psi$ is valid if for every state s satisfying the pre-condition ϕ, a run of the program p starting in s terminates, and in the terminating state the post-condition ψ holds.

Thus, the formula $\phi \to \langle p \rangle\psi$ is similar to the Hoare triple $\{\phi\}p\{\psi\}$. But in contrast to Hoare logic, the set of formulas of DL is closed under the usual logical operators: In Hoare logic, the formulas ϕ and ψ are pure first-order formulas. DL allows to involve programs in the descriptions ϕ resp. ψ of states. For example, using a program, it is easy to specify that a data structure is not cyclic, which is impossible in pure first-order logic. Because all JAVA constructs are available in DL for the description of states (including `while` loops and recursion) it is not necessary to define an abstract data type *state* and to represent states as terms of that type; instead DL formulas can be used to give a (partial) description of states, which is a more flexible technique and allows to concentrate on the relevant properties of a state.

3.2 Syntax of Java Card DL

As said above, a dynamic logic is constructed by extending some non-dynamic logic with modal operators of the form $\langle p \rangle$. The non-dynamic base logic of our DL is a typed first-order predicate logic. We do not describe in detail what the types of our logic are (basically they are identical with the JAVA types) nor how exactly terms and formulas are built, as this is not relevant for the handling of abrupt termination. The definitions can be found in [2]. Note, that terms (which we often call "logical terms" in the following) are different from JAVA expressions; they never have side effects.

In order to reduce the complexity of the programs occurring in DL formulas, we introduce the notion of a *program context*. The context can consist of any legal Java Card program, i.e., it is a sequence of class and interface definitions. Syntax and semantics of DL formulas are then defined with respect to a given context; and the programs in DL formulas are assumed not to contain class definitions.

A context must not contain any constructs that lead to a compile-time error or that are not available in Java Card.[2]

The programs in DL formulas are basically executable Java Card code; as said above, they must not contain class definitions but can only use classes defined in the program context. We introduced two additional constructs that are not available in plain Java Card but are necessary for certain rule applications: Programs can contain a special construct for method invocation (see below), and they can contain logical terms. These extensions are not used in the input formulas, they occur only within proofs, i.e., we prove properties of pure Java Card programs.

Example 1. The statement `i=0;` may be used as a program in a DL formula although `i` is not declared as a local variable.

The statement `break l;` is `not` a legal program because such a statement is only allowed to occur inside a block labelled with `l`. Accordingly, `l:{break l;}` is a legal program and can be used in a DL formula.

The purpose of our first extension is the handling of method calls. Methods are invoked by syntactically replacing the call by the method's implementation. To handle the `return` statement in the right way, it is necessary (a) to record the object field or variable x that the result is to be assigned to, (b) to record the old value *old* of `this`, and (c) to mark the boundaries of the implementation $prog$ when it is substituted for the method call. For that purpose, we allow statements of the form `call(`*old*`,`x`){`$prog$`}` to occur in DL programs.

The second extension is to integrate logical terms in programs contained in DL formulas (not in the program context). This is necessary to be able to replace Java expressions with possible side effects by a logical term of the same type. However, since the value of logical terms cannot and must not be changed by a program, a logical term can only be used in positions where a `final` local variable could be used according to the Java language specification (the value of local variables that are declared `final` cannot be changed either). In particular, logical terms cannot be used as the left hand side of an assignment.

3.3 Semantics of Java Card DL

The semantics of a program p is a state transition, i.e., it assigns to each state s the set of all states that can be reached by running p starting in s. Since Java

[2] An additional restriction is that a program context must not contain *inner classes* (this restriction is "harmless" because inner classes can be removed with a structure-preserving program transformation and are rarely used in Java Card anyway).

CARD is deterministic, that set either contains exactly one state (if p terminates normally) or is empty (if p does not terminate or terminates abruptly). The set of states of a model must be closed under the reachability relation for all programs p, i.e., all reachable states must exist in a model (other models are not considered).

The semantics of a logical term t occurring in a program is the same as that of a JAVA expression whose evaluation is free of side-effects and gives the same value as t.

For formulas ϕ that do not contain programs, the notion of ϕ being satisfied by a state is defined as usual in first-order logic. A formula $\langle p \rangle\phi$ is satisfied by a state s if the program p, when started in s, terminates normally in a state s' in which ϕ is satisfied. A formula is satisfied by a model M, if it is satisfied by one of the states of M. A formula is valid in a model M if it is satisfied by all states of M; and a formula is valid if it is valid in all models.

As mentioned above, we consider programs that terminate abruptly to be non-terminating. Thus, for example, $\langle \texttt{throw x;} \rangle\phi$ is unsatisfiable for all ϕ. Nevertheless, it is possible to express and (if true) prove the fact that a program p terminates abruptly. For example, the formula

$$\texttt{e} \doteq \texttt{null} \ \rightarrow \ \langle \texttt{try\{}p\texttt{\}catch(Exception e)\{\}} \rangle (\neg\, (\texttt{e} \doteq \texttt{null}))$$

is true in a state s if and only if the program p, when started in s, terminates abruptly by throwing an exception (as otherwise no object is bound to `e`).

Sequents are notated following the scheme

$$\phi_1, \ldots, \psi_m \ \vdash \ \psi_1, \ldots, \psi_n \ ,$$

which has the same semantics as the formula

$$(\forall x_1) \cdots (\forall x_k)((\phi_1 \wedge \ldots \wedge \psi_m) \rightarrow (\psi_1 \vee \ldots \vee \psi_n)) \ ,$$

where $x_1, \ldots, x_k$ are the free variables of the sequent.

4 Sequent Calculus Rules for Handling Abrupt Termination

4.1 Notation

The rules of our calculus operate on the first *active* command p of a program $\pi p \omega$. The non-active prefix π consists of an arbitrary sequence of opening braces "`{`", labels, beginnings "`try{`" of `try-catch-finally` blocks, and beginnings "`call(...){`" of method invocation blocks. The prefix is needed to keep track of the blocks that the (first) active command is part of, such that the abruptly terminating statements `throw`, `return`, `break`, and `continue` can be handled appropriately.[3] The postfix ω denotes the "rest" of the program, i.e., everything except the non-active prefix and the part of the program the rule operates on. For example, if a rule is applied to the following JAVA block operating on its first active command `i=0;`, then the non-active prefix π and the "rest" ω are the marked parts of the block:

$$\underbrace{\texttt{l:\{try\{}}_{\pi}\ \texttt{i=0;}\ \underbrace{\texttt{j=0; \}finally\{ k=0; \}\}}}_{\omega}$$

4.2 Loop Rules

Due to space restrictions, we present only one specific rule for `while` loops to demonstrate the properties of loop rules. `for` and `do-while` loops are handled analogously.

The following rule "unwinds" `while` loops. Its application is the prerequisite for symbolically executing the loop body. These "unwind" rules allow to handle `while` loops if used together with induction schemata for the primitive and the user defined types (see the example in Section 5).

$$\frac{\Gamma \vdash (\langle \pi\ \texttt{if(}c\texttt{)}\,l'\texttt{:\{}l''\texttt{:\{}p'\texttt{\}}\ l_1\texttt{:}\cdots l_n\texttt{:while(}c\texttt{)\{}p\texttt{\}\}}\ \omega\rangle\phi)}{\Gamma \vdash (\langle \pi\ \ l_1\texttt{:}\cdots l_n\texttt{:while(}c\texttt{)\{}p\texttt{\}}\ \omega\rangle\phi)} \tag{R1}$$

where

- l' and l'' are new labels,
- p' is the result of (simultaneously) replacing in p
  (a) every `break` l_i (for $1 \leq i \leq n$) and every `break` (with no label) that has the `while` loop as its target by `break` l', and
  (b) every `continue` l_i (for $1 \leq i \leq n$) and every `continue` (with no label) that has the `while` loop as its target by `break` l''.[4]

The list l_1:, ... ,l_n: usually has only one element or is empty, but in general a loop can have more than one label.

In the "unwound" instance p' of the loop body p, the label l' is the new target for `break` statements and l'' is the new target for `continue` statements, which both had the `while` loop as target before. This results in the desired behaviour: `break` abruptly terminates the whole loop, while `continue` abruptly terminates the current instance of the loop body.

A `continue` with or without label is never handled by a rule directly, because it can only occur in loops, where it is always transformed into a break by the loop rules.

[3] In DL versions for simple artificial programming languages, where no prefixes are needed, any formula of the form $\langle p\ q\rangle\phi$ can be replaced by $\langle p\rangle\langle q\rangle\phi$. In our calculus, splitting of $\langle \pi p q \omega\rangle\phi$ into $\langle \pi p\rangle\langle q \omega\rangle\phi$ is not possible (unless the prefix π is empty) because πp is not a valid program; and the formula $\langle \pi p \omega\rangle\langle \pi q \omega\rangle\phi$ cannot be used either because its semantics is in general different from that of $\langle \pi p q \omega\rangle\phi$.

[4] The target of a `break` or `continue` statement with no label is the loop that immediately encloses it.

4.3 Rules for the Abruptly Terminating Statements

Possible Combinations of Prefix and Abruptly Terminating Statement. In the following, we present rules for combinations of prefix type (beginning of a block, method invocation or `try`) and abruptly terminating statement (`break`, `return` or `throw`). Due to restrictions of the language specification, the combination method invocation/`break` does not occur. Also, `switch` statements, which may contain a `break`, are not considered here; they are transformed into a sequence of `if` statements.

Evaluation of Arguments. The arguments exc and val of statements `throw` exc resp. `return` val must already be evaluated (they must be logical terms) before the appropriate rule for redirecting the control flow can be applied to the abruptly terminating statement. Otherwise, a rule such as the following (rule (R2)) has to be used first, which then allows the application of other rules that evaluate the expression exc.

$$\frac{\Gamma \vdash \langle \pi \ \texttt{\{}x\texttt{=}exc\texttt{;}\ \texttt{throw}\ x\texttt{;\}}\ \omega\rangle\phi}{\Gamma \vdash \langle \pi \ \texttt{throw}\ exc\texttt{;}\ \omega\rangle\phi} \tag{R2}$$

where x is a new variable of the same type as the expression exc. Since, in this paper we focus on the handling of abrupt termination here and not on the evaluation of expressions, we assume in the following that this has already been done.

We also do not consider the problem of undefined expressions in this paper, whose evaluation results in an exception being thrown (e.g., the expression `o.a` if the value of `o` is `null`). If an expression e occurs that may be undefined, the rules have a further premiss $\Gamma \vdash isdef(e)$ in the full version of the calculus.

Rule for Method Call/`return`. The rule for this combination symbolically executes every step the virtual machine does when a method invocation is terminated: The return value is assigned to the location recorded in the method call prefix and `this` is restored to the value it had before method invocation.

$$\frac{\Gamma \vdash \langle \pi \ x\texttt{=}y\texttt{;}\ \texttt{this=}old\texttt{;}\ \omega\rangle\phi}{\Gamma \vdash \langle \pi \ \texttt{call(}old\texttt{,}\ x\texttt{):\{return}\ y\texttt{;}\ pgm\texttt{\}}\omega\rangle\phi} \tag{R3}$$

In pure JAVA it is not possible to explicitly assign a value to `this`. Our assignment rule, however, can handle such a statement and produces the desired effect. The "rest" program pgm of the method body, which is not executed, may be empty.

Rule for Method Call/`throw`. In this case, the method is terminated and `this` is restored to its old value, but no return value is assigned. The `throw` statement remains unchanged (i.e., the exception is handed up to the invoking program).

$$\frac{\Gamma \vdash \langle \pi \ \texttt{this=}\mathit{old}\texttt{;}\ \texttt{throw}\ \mathit{exc}\ \texttt{;}\ \omega\rangle\phi}{\Gamma \vdash \langle \pi \ \texttt{method call(}\mathit{old},\ x\texttt{):\{throw}\ \mathit{exc}\ \texttt{;}\ \mathit{pgm}\texttt{\}}\omega\rangle\phi} \quad \text{(R4)}$$

Again, the "rest" *pgm* of the method body, which is not executed, may be empty.

Rules for `try/throw`. The following rules allow to handle `try-catch-finally` blocks and the `throw` statement. These are simplified versions of the actual rules that apply to the case where there is exactly one `catch` clause and one `finally` clause.

$$\frac{\Gamma \vdash \mathit{instanceof}(\mathit{exc}, T) \qquad \Gamma \vdash (\langle \pi \ \texttt{try\{}e\texttt{=}\mathit{exc}\texttt{;}q\texttt{\}finally\{}r\texttt{\}}\ \omega\rangle\phi}{\Gamma \vdash (\langle \pi \ \texttt{try\{throw}\ \mathit{exc}\texttt{;}\ p\texttt{\}catch(}T\ e\texttt{)\{}q\texttt{\}finally\{}r\texttt{\}}\ \omega\rangle\phi)} \quad \text{(R5)}$$

$$\frac{\Gamma \vdash \neg\mathit{instanceof}(\mathit{exc}, T) \qquad \Gamma \vdash (\langle \pi \ r\texttt{;}\ \texttt{throw}\ \mathit{exc}\ \texttt{;}\ \omega\rangle\phi)}{\Gamma \vdash (\langle \pi \ \texttt{try\{throw}\ \mathit{exc}\ \texttt{;}\ p\texttt{\}catch(}T\ e\texttt{)\{}q\texttt{\}finally\{}r\texttt{\}}\ \omega\rangle\phi)} \quad \text{(R6)}$$

Rule (R5) applies if an exception *exc* is thrown that is an instance of exception class T, i.e., the exception is caught; otherwise, if the exception is not caught, rule (R6) applies.

Rules for `try/break` and `try/return`. A `return` or a `break` statement within a `try-catch-finally` statement causes the immediate execution of the `finally` block. Afterwards the `try` statement terminates abnormally with the `break` resp. the `return` statement (a different abruptly terminating statement in the `finally` block takes precedence). This behaviour is simulated by the following two rules:

$$\frac{\Gamma \vdash \langle \pi \ r\ \texttt{break}\ l\ \texttt{;}\ \omega\rangle\phi}{\Gamma \vdash \langle \pi \ \texttt{try\{break}\ l\ \texttt{;}\ p\texttt{\}catch(}T\ \mathit{exc}\texttt{)\{}q\texttt{\}finally\{}r\texttt{\}}\ \omega\rangle\phi} \quad \text{(R7)}$$

$$\frac{\Gamma \vdash \langle \pi \ r\ \texttt{return}\ v\ \texttt{;}\ \omega\rangle\phi}{\Gamma \vdash \langle \pi \ \texttt{try\{return}\ v\ \texttt{;}\ p\texttt{\}catch(}T\ \mathit{exc}\texttt{)\{}q\texttt{\}finally\{}r\texttt{\}}\ \omega\rangle\phi} \quad \text{(R8)}$$

Rules for block/`break`, block/`return`, and block/`throw`. Rules (R9), (R10) apply to blocks being terminated by a `break` statement without label resp. with a label l matching one of the labels $l_1, \ldots, l_k$ of the block ($k \geq 0$).

$$\frac{\Gamma \vdash \langle \pi \ \ \omega\rangle\phi}{\Gamma \vdash \langle \pi \ \ l_1\texttt{:}\cdots l_k\texttt{:\{break;}\ pgm\texttt{\}}\ \omega\rangle\phi} \tag{R9}$$

$$\frac{\Gamma \vdash \langle \pi \ \ \omega\rangle\phi}{\Gamma \vdash \langle \pi \ \ l_1\texttt{:}\cdots l_k\texttt{:\{break}\ l\texttt{;}\ pgm\texttt{\}}\ \omega\rangle\phi} \quad \text{where } l \in \{l_1,\ldots,l_k\} \tag{R10}$$

The following rules handle labelled and unlabelled blocks that are abruptly terminated by a `break` statement with a label l not matching any of the labels of the block (Rule (R11)), or by a `return` or `throw` statement (Rules (R12) resp. (R13)).

$$\frac{\Gamma \vdash \langle \pi \ \texttt{break}\ l\texttt{;}\ \omega\rangle\phi}{\Gamma \vdash \langle \pi \ \ l_1\texttt{:}\cdots l_k\texttt{:\{break}\ l\texttt{;}\ pgm\texttt{\}}\ \omega\rangle\phi} \quad \text{where } l \notin \{l_1,\ldots,l_k\} \tag{R11}$$

$$\frac{\Gamma \vdash \langle \pi \ \texttt{return}\ v\texttt{;}\ \omega\rangle\phi}{\Gamma \vdash \langle \pi \ \ l_1\texttt{:}\cdots l_k\texttt{:\{return}\ v\texttt{;}\ pgm\texttt{\}}\ \omega\rangle\phi} \tag{R12}$$

$$\frac{\Gamma \vdash \langle \pi \ \texttt{throw}\ e\texttt{;}\ \omega\rangle\phi}{\Gamma \vdash \langle \pi \ \ l_1\texttt{:}\cdots l_k\texttt{:\{throw}\ e\texttt{;}\ pgm\texttt{\}}\ \omega\rangle\phi} \tag{R13}$$

In all the rules above, the program pgm (that is not executed) may be empty.

Rules for Empty Blocks. Rule (R14) applies to empty `try` blocks, which terminate normally. There are similar rules for empty blocks and empty method invocations.

$$\frac{\Gamma \vdash (\langle \pi \ \ r \ \ \omega\rangle\phi)}{\Gamma \vdash (\langle \pi \ \texttt{try\{\}catch(}T\ e\texttt{)\{}q\texttt{\}finally\{}r\texttt{\}}\ \omega\rangle\phi)} \tag{R14}$$

5 Example

As an example, we use the calculus presented in the previous section to verify that, if the program

```
while (true) {
  if (i==10) break;
  i++;
}
```

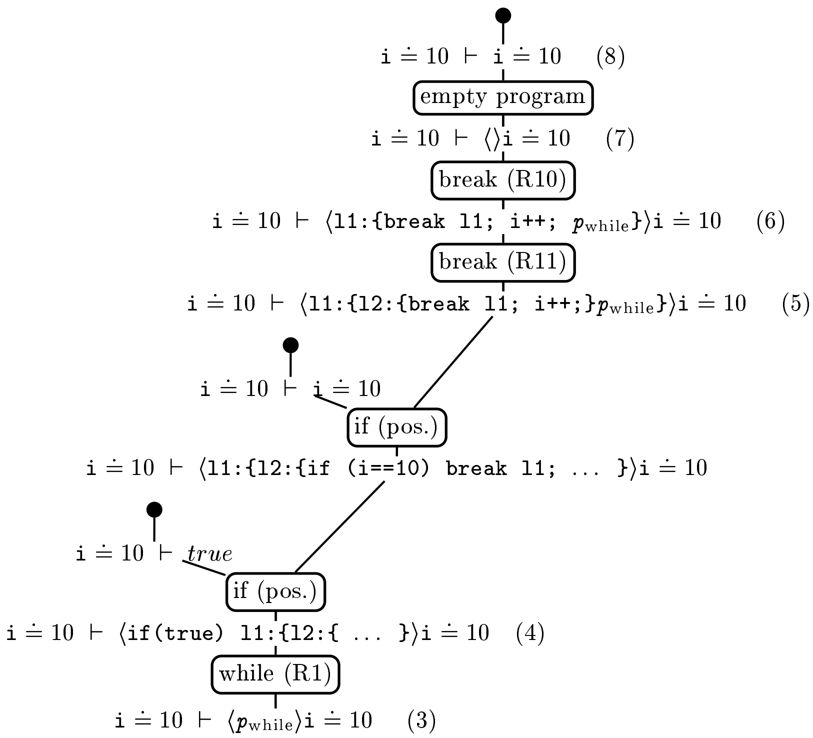

Fig. 1. Base case ($n \doteq 0$) of the inductive proof for sequent (2).

is started in a state in which the value of the variable `i` is between 0 and 10, then it terminates normally in a state in which the value of `i` is 10.[5] That is, we prove that the sequence

$$0 \leq \mathtt{i} \land \mathtt{i} \leq 10 \ \vdash \ \langle p_{\text{while}} \rangle \mathtt{i} \doteq 10 \tag{1}$$

is valid, where p_{while} is an abbreviation for the above while loop. Instead of proving (1) directly, we first use induction to derive the sequence

$$\vdash \ (\forall n)((n \leq 10 \land \mathtt{i} \doteq 10 - n) \to \langle p_{\text{while}} \rangle \mathtt{i} \doteq 10) \tag{2}$$

as a lemma. It basically expresses the same as (1), the difference is that its form allows a proof by induction on n. The introduction of this lemma is the only step in the proof where an intuition for what the JAVA CARD program p_{while} actually does is needed and where a verification tool may require user interaction.

The derivation of (2) is shown schematically in Figures 1 and 2 (arithmetic simplifications are omitted). The first sequent which appears in the base case (Fig. 1) after applying the induction rule and some simplifications is

$$\mathtt{i} \doteq 10 \ \vdash \ \langle \mathtt{while\ (true)\ \{if\ (i==10)\ break;\ i++;\}} \rangle \mathtt{i} \doteq 10 \tag{3}$$

[5] This example program was first presented in [3].

An application of the rule for `while` loops (R1) results in the new proof obligation

$$\begin{aligned}&\mathtt{i} \doteq 10 \ \vdash \\ &\quad \langle\texttt{if (true) l1:\{l2:\{if (i==10) break l1; i++;\}}\ p_{\text{while}}\texttt{\}}\rangle\mathtt{i} \doteq 10\end{aligned} \tag{4}$$

Here, two new labels are introduced: `l1` is the target for `break` statements in the loop body and `l2` is the target for `continue` statements (the latter does not occur in this example).

The next step is to use twice the rule for `if` statements with a condition that evaluates to true (the proof obligations stemming from the conditions are axiomatic and easily discharged). After the second application, we get the sequent

$$\mathtt{i} \doteq 10 \ \vdash \ \langle\texttt{l1:\{l2:\{break l1; i++;\}}\ p_{\text{while}}\texttt{\}}\rangle\mathtt{i} \doteq 10 \tag{5}$$

in which the next executable statement is `break l1`. Now, the rule for labelled `break` statements in a block with a non-matching label (R11) has to be applied, which eliminates the block labelled with `l2`:

$$\mathtt{i} \doteq 10 \ \vdash \ \langle\texttt{l1:\{break l1;}\ p_{\text{while}}\texttt{\}}\rangle\mathtt{i} \doteq 10 \tag{6}$$

Then, the rule for labelled `break` statements in a block with a *matching* label (R10) is used. The result is

$$\mathtt{i} \doteq 10 \ \vdash \ \langle\rangle(\mathtt{i} \doteq 10) \tag{7}$$

This simplifies with the rule for the empty program to

$$\mathtt{i} \doteq 10 \ \vdash \ \mathtt{i} \doteq 10 \tag{8}$$

and can thus be shown to be valid.

The step case (Fig. 2) proceeds in a similar way (the main difference is that it closes with an application of the induction hypothesis while the base case closes with an axiomatic sequent). The first sequent in the step case after applying some simplifications is

$$n \leq 9, \mathtt{i} \doteq 9 - n \ \vdash \ \langle\texttt{while (true) \{if (i==10) break; i++;\}}\rangle\mathtt{i} \doteq 10 \tag{9}$$

Again, we first apply the rule for `while` loops (R1) and then the rule for `if` statements with a condition that evaluates to true. Then, there is a difference to the base case: In the step case `i` is not equal to 10, and we now have to apply the rule for `if` statements with a condition that evaluates to false. The result is the sequent

$$n \leq 9, \mathtt{i} \doteq 9 - n \ \vdash \ \langle\texttt{l1:\{l2:\{i++;\}}\ p_{\text{while}}\texttt{\}}\rangle\mathtt{i} \doteq 10 \tag{10}$$

The `break` statement has been skipped, and the next executable statement is `i++`. After applying the corresponding rule, we get

$$n \leq 9, \mathtt{i} \doteq 10 - n \ \vdash \ \langle\texttt{l1:\{l2:\{\}}\ p_{\text{while}}\texttt{\}}\rangle\mathtt{i} \doteq 10 \tag{11}$$

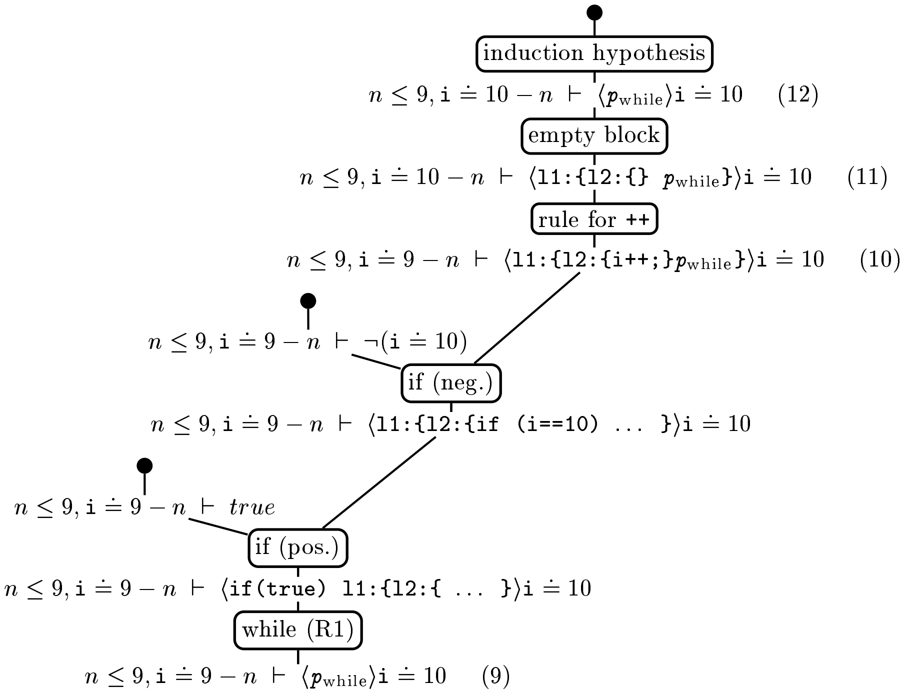

Fig. 2. Step case ($n \to n+1$) of the inductive proof for sequent (2).

Then, the rule for empty blocks is applied, and a rule that allows to replace l :{p} by p provided that the label `l` does not occur in p. The result is

$$n \le 9, \mathtt{i} \doteq 10 - n \ \vdash \ \langle p_{\text{while}} \rangle \mathtt{i} \doteq 10 \tag{12}$$

Now, since $n \le 9$ implies $n \le 10$, the inductions hypothesis can be applied to close the step case.

After the lemma (2) has been proved by induction, it can be used to prove the original proof obligation (1). First, we use a quantifier rule to instantiate n with $10 - \mathtt{i}$. The result is

$$0 \le \mathtt{i} \land \mathtt{i} \le 10 \ \vdash \ (10 - \mathtt{i} \le 10 \land \mathtt{i} \doteq 10 - (\mathtt{i} - 10)) \to (\langle p_{\text{while}} \rangle \mathtt{i} \doteq 10)$$

which can be simplified to

$$0 \le \mathtt{i} \land \mathtt{i} \le 10 \land \mathtt{i} \doteq \mathtt{i} \ \vdash \ (\langle p_{\text{while}} \rangle \mathtt{i} \doteq 10) \tag{13}$$

And, since (13) is derivable, the original proof obligation (1) is derivable as well, because the trivial equality $\mathtt{i} \doteq \mathtt{i}$ can be omitted.

References

1. Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Wolfram Menzel, and Peter H. Schmitt. The KeY approach:

Integrating object oriented design and formal verification. In M. Ojeda-Aciego, I. P. de Guzman, G. Brewka, and L. M. Pereira, editors, *Proceedings, Logics in Artificial Intelligence (JELIA), Malaga, Spain*, LNCS 1919. Springer, 2000.
2. Bernhard Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, LNCS 2041, pages 6–24. Springer, 2001.
3. Marieke Huisman and Bart Jacobs. Java program verification via a Hoare logic with abrupt termination. In *Proceedings, Fundamental Approaches to Software Engineering (FASE), Berlin, Germany*, LNCS 1783. Springer, 2000.
4. Dieter Hutter, Bruno Langenstein, Claus Sengler, Jörg H. Siekmann, and Werner Stephan. Deduction in the Verification Support Environment (VSE). In M.-C. Gaudel and J. Woodcock, editors, *Proceedings, International Symposium of Formal Methods Europe (FME), Oxford, UK*, LNCS 1051. Springer, 1996.
5. Dexter Kozen and Jerzy Tiuryn. Logic of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 14, pages 789–840. Elsevier, Amsterdam, 1990.
6. Arnd Poetzsch-Heffter and Peter Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *Proceedings, Programming Languages and Systems (ESOP), Amsterdam, The Netherlands*, LNCS 1576, pages 162–176. Springer, 1999.
7. Wolfgang Reif. The KIV-approach to software verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report*, LNCS 1009. Springer, 1995.
8. Kurt Stenzel. Verification of Java Card programs. Technical Report 2001-5, Institut für Informatik, Universität Augsburg, 2001.