In: *Java on Smart Cards: Programming and Security*, pages 6–24, LNCS 2041, Springer, 2001.

A Dynamic Logic for the Formal Verification of Java Card Programs

Bernhard Beckert

Universität Karlsruhe
Institut für Logik, Komplexität und Deduktionssysteme
D-76128 Karlsruhe, Germany
i12www.ira.uka.de/~beckert

Abstract. In this paper, we define a program logic (an instance of Dynamic Logic) for formalising properties of Java Card programs, and we give a sequent calculus for formally verifying such properties. The purpose of this work is to provide a framework for software verification that can be integrated into real-world software development processes.

1 Introduction

Motivation. The work that is reported in this paper 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 (that do not have years of training in formal methods).

The ultimate goal of the KeY project is to facilitate and promote the use of formal verification as an integral part of the development process of Java Card applications in an industrial context.

In this paper, after giving an overview of the KeY project in Section 2, we present a Dynamic Logic (a program logic that can be seen as an extension of Hoare logic) for Java Card. It allows to express properties of Java Card programs. The syntax of this logic is described in Section 3 and its semantics in Section 4. In Section 5, we present a calculus for this program logic that allows to reason about the properties of Java Card programs and verify them. The main ideas and principles of the calculus are described and its most important rules are presented (due to space restrictions, we cannot list all the rules in this paper). In Section 6, we give an example for the verification of a small Java Card program. As part of the KeY project we currently implement an interactive theorem prover for our calculus; this and other future work is described in Section 7, where we also compare our work with other approaches to the verification of Java Card programs.

Java Card. Since JAVA CARD is a “real” object-oriented language, it has features that are difficult to handle in a software verification system, such as dynamic data structures, exceptions, object initialisation, and dynamic binding. On the other hand, JAVA CARD lacks some crucial complications of the full JAVA language such as threads and dynamic loading of classes. JAVA smart cards are an extremely suitable application for software verification:

- JAVA CARD applications are small;
- at the same time, they are embedded into larger program systems or business processes which should be modeled (though not necessarily formally verified);
- JAVA CARD applications are often security-critical, giving incentive to apply formal methods;
- the high number of deployed smart cards constitutes a new motivation for formal verification, as arbitrary updates are not feasible.

Dynamic Logic. We use an instance of Dynamic Logic (DL) [14]—which can be seen as an extension of Hoare logic [3]—as the logical basis of the KeY system's software verification component. Deduction in DL is based on symbolic program execution and simple program transformations and is, thus, close to a programmer's understanding of JAVA CARD. DL is used in the software verification systems KIV [20] and VSE [12] (for a programming language that is not object-oriented). It has successfully been applied in practice to verify software systems of considerable size.

DL 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 such 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 $\phi \rightarrow \langle p \rangle \psi$ is valid if for every state s satisfying 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 $\phi \rightarrow \langle p \rangle \psi$ 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, whereas in DL they can contain programs. 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. Also, all JAVA constructs are available in DL for the description of states (including `while` loops and recursion). It is, therefore, 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.

In comparison to classical versions of DL that use a simple "artificial" programming languages, a DL for a "real" object-oriented programming language like Java Card has to cope with the following complications:

– A program state does not only depend on the value of (local) program variables but also on the values of the attributes of all existing objects.
– The evaluation of a Java expression may have side effects; thus, there is a difference between an expression and a logical term.
– Language features such as built-in data types, exceptions, object initialisation, and dynamic binding have to be handled.

2 The KeY Project[1]

While formal methods are by now well established in hardware and system design, usage of formal methods in software development is still (and in spite of exceptions [7, 8]) more or less confined to academic research. This is true though case studies clearly demonstrate that computer-aided specification and verification of realistic software is feasible [10].

The future challenge for formal methods is to make their considerable potential feasible to use in an industrial environment. This leads to the requirements:

1. Tools for formal software specification and verification must be integrated into industrial software engineering procedures.
2. User interfaces of these tools must comply with state-of-the-art software engineering tools.
3. The necessary amount of training in formal methods must be minimised.

To be sure, the thought that full formal software verification might be possible without any background in formal methods is utopian. An industrial verification tool should, however, allow for *gradual* verification so that software engineers at any (including low) experience level with formal methods may benefit. In addition, an integrated tool with well-defined interfaces facilitates "outsourcing" those parts of the modeling process that require special skills.

Another important motivation to integrate design, development, and verification of software is provided by modern software development methodologies which are *iterative* and *incremental*. *Post mortem* verification would enforce the antiquated waterfall model.

The KeY project [1]) addresses the goals outlined above. In the principal use case of the KeY system there are actors who want to implement a software system that complies with given requirements and formally verify its correctness (typically a smart card application). In this scenario, the KeY system is responsible for adding formal detail to the analysis model, for creating conditions that ensure the correctness of refinement steps (called proof obligations), for finding proofs showing that these conditions are satisfied by the model, and for generating counter examples if they are not. Special features of KeY are:

[1] More information on the KeY project can be found at `i12www.ira.uka.de/~key`.

– We concentrate on object-oriented analysis and design methods (OOAD)—because of their key role in today's software development practice—, and on JAVA CARD as the target language. In particular, we use the Unified Modeling Language (UML) [18] for visual modeling of designs and specifications and the Object Constraint Language (OCL) for adding further restrictions. This choice is supported by the fact, that the UML (which contains OCL) is not only an OMG standard, but has been adopted by all major OOAD software vendors and is featured in recent OOAD textbooks [15].
– We use a commercial CASE tool as starting point and enhance it by additional functionality for formal specification and verification. The tool of our choice is TogetherSoft LLC's TOGETHERJ.
– Formal verification is based on a Dynamic Logic for JAVA CARD.
– As a case study to evaluate the usability of our approach we develop a scenario using smart cards with JAVA CARD as programming language.
– Through direct contacts with software companies we check the soundness of our approach for real world applications (some of the experiences from these contacts are reported in [4]).

A first KeY system prototype has been implemented, integrating the CASE tool TOGETHERJ and a deductive component. Work on the full KeY system is under progress. Although consisting of different components, the KeY system is going to be fully integrated with a uniform user interface.

3 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. To define its syntax, we specify its types and the variable sets and signatures from which terms are built (which we often call "logical terms" in the following to emphasise that they are different from JAVA expressions). Then, we define which programs p are allowed in the operators $\langle p \rangle$, i.e., in the program parts of DL formulas. Finally, the syntax of DL formulas and sequents is defined.

Program Contexts. 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 is then defined with respect to a given context; and the programs in DL formulas are assumed to not contain class definitions.

A context must not contain any constructs that according to the JAVA language specification lead to a compile-time error or that are not available in JAVA CARD. 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).

Types. Given a program context, the set $\mathcal{T}$ of types contains:

- the primitive types of JAVA CARD (`boolean`, `byte`, `short`),
- the built-in classes `Object` and `String`,
- the classes defined in the program context,[2]
- an array type $T[\,]$ for each primitive type, each array type T, and each class,
- the type *Null*,
- abstract types.

Abstract types are not defined in the program context but are given separately. They can be declared to be generated by certain function symbols (called constructors), in which case they can be used for induction proofs (see Section 5). For example, a type *nat* may be declared to be generated by 0 and *succ*; and an abstract (data) type *list* may be declared to be generated by *cons* and *nil*. Axioms may be provided to specify the properties of abstract types. Since abstract types are not defined as JAVA classes, they can only be used in the non-program parts of a DL formula and not in programs (in particular not in the program context). Nevertheless, they can be used in DL formulas to describe the behaviour of programs (in particular they can be used as abstractions of object structures).

Note that there are three kinds of types in our DL: Built-in JAVA CARD types, types defined in the program context (classes), and abstract types defined separately from the program context. The classes, the array types, and *Null* are called *object types*.[3]

We assume that the methods and fields shown in Table 1 are implicitly defined for each class and each array type and can thus be used in DL formulas (but not in the program context). Note that they are not actually implemented, but only provide additional expressiveness for the logic. They allow to access information about the program state that is otherwise inaccessible in JAVA: a list of all existing objects of a class or array type and information on whether objects and classes are initialised (`classInitialised` is only available for classes and not for array types). The objects of a certain type are considered to be organised into an (infinite) ordered list; this list is used by `new` to "create" objects (intuitively, `new` changes the attributes `lastCreatedObj` of the class and sets the attribute `created` of the new object to `true`, see Section 5).

The sub-type relation $\preceq$ is transitive and reflexive. If C_1 is defined to be a sub-class of C_2 in the program context, then $C_1 \preceq C_2$ and $C_1[\,] \preceq C_2[\,]$. *Null* is a sub-type of all object types.

Variables. In classical versions of DL there is only one type of variables. Here however, to avoid confusion, we use two kinds of variables, namely *program variables* and *logical variables*.

Program variables are denoted with `x`, `y`, `z`, ... Their value can differ from state to state and can be changed by programs. They occur in programs as

[2] Interfaces defined in the context are not types of the logic.
[3] In JAVA, arrays are considered to be objects.

```
public static Cls firstObj;  // the first object in the list,
                             // whether already created or not
public static Cls lastCreatedObj;  // the last created object,
                                   // null if no object exists
public Cls prevObj;  // the previous object in the list,
                     // null for the first object
public Cls nextObj;  // the next object in the list
public boolean beforeObj(Cls obj);  // returns true if this
                                    // is before obj in the list
public boolean created;  // true if the object has already been
                         // created with new, and false otherwise
public static boolean classInitialised;  // true if the class resp.
public boolean objInitialised;           // the object is initialised
```

Table 1. Methods and fields that are implicitly defined for each class *Cls*.

local variables.[4] Program variables can also be used in the non-program parts of DL formulas (there they behave like modal constants, i.e., constants whose value can differ from state to state). They cannot be quantified. We assume the set of program variables to contain an infinite number of variables of each primitive type and each object type. In particular, it contains the special variable `this` of type `Object`.

Logical variables are denoted with x, y, z, ... They are assigned the same values in all states; a statement such as "x `= 1;`", which tries to change the value of the logical variable x, is illegal. Free occurrences of logical variables are implicitly universally quantified. The set of logical variables contains an infinite number of variables of each type.

Terms. Logical terms are constructed from program variables, logical variables, and the constant and function symbols of all types (observing the usual typing restrictions). The set of logical terms includes in particular all JAVA CARD literals for the primitive types, string literals, and the `null` object reference literal (which is of type *Null*).

In addition, (a) if o is a term of class type C (i.e., denotes an object) and `a` is a field (attribute) of class C, then o`.a` is a term. (b) If *`Class`* is a class name and *`a`* is a static field of *`Class`*, then *`Class.a`* is a term. (c) If a is an array type term and i is a term of type `byte`, then $a[i]$ is a term.

Example 1. Assume class `C` has an attribute `a` of type `C` and an attribute `i` of type `byte`, and `o` is a variable of type `C`. Then `o`, `o.a`, `o.a.a`, etc. are terms of type C; and `o.i`, `o.a.i` etc. are terms of type `byte`. Also, `1+2` and `o.i+1` are terms of type byte. The JAVA expression `o.i++`, however, is not a logical term

[4] In the JAVA language specification, certain more complex expressions such as `x.a` are called *variables* as well. According to our definitions, however, `x.a` is not a variable but a (complex) term.

because `++` is not a function symbol (it is an operator with side effects). The expression `o.i==1` is a logical term of type `boolean`.

Programs. Basically, the programs in DL formulas are executable JAVA CARD code; as said above, they must not contain class definitions but can only use classes that are defined in the program context. There are two additions that are not available in pure JAVA CARD: 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, i.e., we prove properties of pure JAVA CARD programs. Extended programs only occur within proofs; they result from rule applications.

The basic, non-extended programs either are a legal JAVA CARD statement or a (finite) sequence of such statements:

- expression statements such as "`x = 1;`" (assignments), "`m(1);`" (method calls), "`i++;`", "`new Cls;`", local variable declarations (which restrict the "visibility" of program variables)—expressions with inner classes are *not* allowed;
- blocks and compound statements built with `if-else`, `switch`, `for`, `while`, and `do-while`;
- statements with exception handling using `try-catch-finally`;
- statements that abruptly redirect the control flow (`throw`, `return`, `break`, `continue`);
- labelled statements;
- the empty statement.

A basic program must not contain anything that would lead to a compile-time error (according to the JAVA language specification) if it were used as, for example, a method's implementation. The only exception to that rule is that program variables may be used as local variables in a program without being declared.

Example 2. 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 of pure JAVA CARD 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 to record the program variable or object field that the result is to be assigned to and to mark the boundaries of the implementation when it is substituted for the method call. For that purpose, we allow statements of the form `call(`x`){`*prog*`}` resp. `call{`*prog*`}` to occur in DL programs, where *prog* is considered to be the implementation of a method and x is the variable or object field that the return value of *prog* is to be assigned to (if `(`x`)` is omitted, *prog* must not return a value).

The second extension is that we allow programs in DL formulas (not in the program context) to contain logical terms. A JAVA expression of type T can be replaced by a logical term of type T. 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.

Note that, according to our definitions, both program variables and logical variables can occur in the program parts as well as the non-program parts of a DL formula. Nevertheless, there is a difference between the two kinds of variables, as the following example demonstrates.

Example 3. If `x` is a program variable and y is a logical variable, then the formula $(\forall y)(\langle \mathtt{x}\texttt{=}y \rangle \mathtt{x} \doteq y)$ is syntactically correct. However, $(\forall y)(\langle y\texttt{=}\mathtt{x} \rangle \mathtt{x} \doteq y)$ is not a formula because logical variables must not be used as the left side of an assignment. And $(\forall \mathtt{x})(\langle \mathtt{x}\texttt{=}y \rangle \mathtt{x} \doteq y)$ is a not a formula because program variables cannot be quantified.

Formulas. Atomic formulas are built as usual from the (logical) terms and the predicate symbols of all the types, including the following special predicates:

- the equality predicate $\doteq$,
- the (unary) definedness predicate *isdef* (which, for example, is false for `x.a` if the value of `x` is *null*),
- the (binary) predicate *instanceof*.

Complex formulas are constructed from the atomic formulas using the logical connectives $\neg$, $\wedge$, $\vee$, $\rightarrow$, the quantifiers $\forall$ and $\exists$ (that can be applied to logical variables but not to program variables), and the modal operator $\langle \boldsymbol{p} \rangle$, i.e., if $\boldsymbol{p}$ is a program and ϕ is a formula, then $\langle \boldsymbol{p} \rangle \phi$ is a formula as well.

Updates. One of the main problems of designing a program logic for JAVA CARD (or any other object-oriented language) is *aliasing*. That is, different object type variables o_1 and o_2 can be aliases for the same object, such that changing an attribute of o_1 changes the same attribute of o_2 as well. A considerable amount of literature has been published on this problem (see e.g. [6] for an overview), which is comparable to the problem of array handling. In the same way, as o_1`.a` and o_2`.a` are the same if o_1 and o_2 have the same object as their value and `a` is an attribute, `a[`i_1`]` and `a[`i_2`]` are the same if the `byte` variables i_1 and i_2 have the same value and `a` is the name of an array.

To handle aliasing in our calculus, we need a way of syntactically denoting what the value of o_1`.a` (resp. `a[`i_1`]`) is in a state where the value o_2`.a` (resp. `a[`i_2`]`) has been changed; the representation should be independent of whether o_1 and o_2 (resp. i_1 and i_2) have the same value or not. For that purpose, we allow *updates* of the form $\boldsymbol{v} \leftarrow e$ to be attached as superscripts to terms, formulas, attributes, and array variables; $\boldsymbol{v}$ is either a local variable or of the form $o.\boldsymbol{a}$, and $\boldsymbol{e}$ is a logical term of compatible type. Thus, if U is an update and t and ϕ are a term resp. a formula, then t^U and ϕ^U are a term resp. a formula as well. Moreover, $o.\boldsymbol{a}^U$ is a term if $o.\boldsymbol{a}$ is a term, and $\boldsymbol{a}^U[i]$ is a term if $\boldsymbol{a}[i]$ is a term.

The intuitive meaning of an update is that the term or formula that it is attached to is to be evaluated after changing the state accordingly, i.e., $\phi^{\boldsymbol{x} \leftarrow e}$ has the same semantics as $\langle \boldsymbol{x} = e \rangle \phi$ but is easier to handle because the evaluation of e is known to have no side effects. Note, that the terms $o.\boldsymbol{a}^U$ and $(o.\boldsymbol{a})^U$ may have different values because in the former term the update does not apply to o (which is evaluated in the non-updated state) whereas in the latter term the update applies to o as well.

Rules for simplifying terms and formulas with attached updates are described in Section 5.

Example 4. The formula $(\langle \mathtt{i=j;} \rangle (\mathtt{i} \doteq \mathtt{j}))^{\mathtt{i} \leftarrow 1}$ is valid, i.e., true in all states. The formula $\langle \mathtt{i=j;} \rangle ((\mathtt{i} \doteq \mathtt{j})^{\mathtt{i} \leftarrow 1})$ is only valid in states where the value of `j` is 1.

Sequents. A sequent is of the form $\phi_1, \ldots, \phi_m \vdash \psi_1, \ldots, \psi_n$ $(m, n \geq 0)$, where the ϕ_i and ψ_j are DL formulas. The intuitive meaning of a sequent is that the conjunction of the ϕ_i's implies the disjunction of the ψ_j's.

4 Semantics of Java Card DL

In the definition of the semantics of JAVA CARD DL, we use the semantics of the JAVA CARD programming language. The language specification [9], though written in English and not in a formal language, is very precise. In case of doubt, we refer to the precise semantics of JAVA (and, thus, of the subset JAVA CARD) defined by Börger and Schulte [5] using Abstract State Machines.[5]

The models of DL are Kripke structures consisting of possible worlds that are called states. All states of a model share the same universe containing a sufficient number of elements of each type. In particular, they contain infinitely many objects of all classes and all array types and the special value *null*, which is the only element of type *Null*.

The function and predicate symbols that are not user-defined—such as the equality predicate and the function symbols of the primitive JAVA CARD types—have a fixed interpretation. In all models they are interpreted according to their intended semantics resp. their meaning in the JAVA CARD language.

Logical variables are interpreted using a (global) variable assignment; they have the same value in all states of a model.

[5] Following another approach, Nipkow and von Oheimb have obtained a precise semantics of a JAVA sublanguage by embedding it into Isabelle/HOL; they also use an axiomatic semantics [16].

States. In each state a (possibly different) value (an element of the universe) of the appropriate type is assigned to:

- the program variables (including `this`),
- the attributes (fields) of all objects (including arrays),
- the class attributes (static fields) of all types,

Variables and attributes of type T can be assigned a value of type T' if $T' \preceq T$. In particular, variables and attributes of any object type can be assigned the value *null*, because *Null* is a sub-type of all object types.

Note, that states do not contain any information on control flow such as a program counter or the fact that an exception has been thrown.

Programs and Formulas. The semantics of a program $\boldsymbol{p}$ is a state transition, i.e., it assigns to each state s the set of all states that can be reached by running $\boldsymbol{p}$ starting in s. Since JAVA CARD is deterministic, that set either contains exactly one state (in case $\boldsymbol{p}$ terminates) or is empty (in case $\boldsymbol{p}$ does not terminate). The set of states of a model must be closed under the reachability relation for all programs $\boldsymbol{p}$, i.e., all states that are reachable 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 $\langle \boldsymbol{p} \rangle\phi$ 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.[6] 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.

We consider programs that terminate abruptly to be non-terminating. Examples are a program that throws an uncaught exception and a `return` statement that is not within the boundaries of a method invocation. 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 $\boldsymbol{p}$ terminates abruptly. For example, the formula

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

is true in a state s if and only if the program $\boldsymbol{p}$, when started in s, terminates abruptly by throwing an exception.

[6] According to the JAVA language specification, a program either terminates normally or terminates abruptly (or does not terminate at all). It terminates abruptly if the reason for termination is an uncaught exception, or the execution of a **`break`**, **`continue`**, or **`return`** statement.

Sequents. The semantics of a sequent $\phi_1, \ldots, \psi_m \vdash \psi_1, \ldots, \psi_n$ is the same as that of the formula $(\forall x_1) \cdots (\forall x_k)((\phi_1 \wedge \ldots \wedge \psi_m) \to (\psi_1 \vee \ldots \vee \psi_n))$, where $x_1, \ldots, x_k$ are the free variables of the sequent.

5 A Sequent Calculus for Java Card DL

In this section, we outline the ideas behind our calculus for JAVA CARD DL, and we present some of the basic rules. As JAVA CARD has many features and programming constructs, many rules are required. Due to space restrictions, we only present one or two typical representatives from each class of rules. No rules are shown for method invocations,[7] local variable declarations, and type conversions; and the rules for the classical logical operators (including the cut rule) and for handling equality and the predicates *isdef* and *instanceof* are omitted as well. Moreover, we present simplified versions of our rules that do not consider initialisation of objects and classes.[8]

All the rules shown in this section, except the induction rules, handle certain constructs of the JAVA CARD language. It is easy to see, that these rules basically perform a symbolic program execution.

The semantics of sequent rules is that, if all sequences above the line (the premisses of the rule) are valid, then the sequence below the line (the conclusion) is valid as well. The rules are applied from bottom to top. That is, the proof search starts with the original proof obligation at the bottom.

Notation. In the definition of the calculus, we assume that the programs are parsed, i.e., they are not given as a string but their syntax tree is available. Thus, the calculus needs not to know about operator priorities etc., and we can use notions like "immediate sub-expression" in the definition of our rules.

Many formulas in the rules are of the form $(\langle \boldsymbol{p} \rangle \phi)^U$, where U is a sequence of state updates. Note, that the parentheses cannot be omitted, as the program $\boldsymbol{p}$ is to be executed in the updated state.

The rules of our calculus operate on the first *active* command $\boldsymbol{p}$ of a program $\pi \boldsymbol{p}\, \omega$. The non-active prefix π consists of an arbitrary sequence of opening braces "`{`", labels, beginnings "`try{`" of `try-catch` 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 commands `throw`, `return`, `break`, and `continue` that abruptly change the control flow can

[7] Method invocation is handled by syntactically replacing the method call by the implementation of the method. In case of dynamic binding, where the implementation that is to be used depends on the actual type that the value of an object variable has in the current state, method invocation leads to a case distinction in the proof, i.e., the proof tree branches.

[8] The complete rule set of our calculus for JAVA CARD DL can be found in a technical report that—at the date of submission of this paper—is in the process of being published. It will be publicly available before TACAS 2001; and I am happy to provide a draft of the report for the referees if they wish to have it.

be handled appropriatly.[9] The postfix ω denotes the "rest" of the program, i.e., everything except the non-active prefix and the part of the program that 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:

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

Rules for Assignment and Expression Evaluation. Since assignments are the basic state changing statements of JAVA, the rule for assignments is one of the basic and most important rules of the calculus:[10]

$$\frac{\Gamma \vdash \mathit{isdef}(o.\mathtt{a}^U) \qquad \Gamma \vdash \mathit{isdef}(\mathit{expr}^U) \qquad \Gamma \vdash (((\langle\pi\ \ \omega\rangle\phi)^{o.\mathtt{a}\leftarrow \mathit{expr}})^U}{\Gamma \vdash (\langle\pi\ \ o.\mathtt{a}\ \mathtt{=}\ \mathit{expr}\mathtt{;}\ \ \omega\rangle\phi)^U} \tag{R1}$$

Rule (R1) is not always applicable; it can only be used if the expression *expr* is a logical term. Otherwise, other rules have to be applied first to evaluate *expr* (as that evaluation may have side effects). An example is the following rule for evaluating expressions with the `++` prefix operator:

$$\frac{\Gamma \vdash \mathit{isdef}(v^U) \qquad \Gamma \vdash (\langle\pi\ \ e\mathtt{=}e\mathtt{+1;}\ \ v\mathtt{=}e\mathtt{;}\ \ \omega\rangle\phi)^U}{\Gamma \vdash (\langle\pi\ \ v\ \mathtt{=}\ \mathtt{++}e\mathtt{;}\ \ \omega\rangle\phi)^U} \tag{R2}$$

where v and e are logical terms.

There are also rules for decomposing complex expressions that are not a logical term and whose evaluation, thus, potentially has side effects. An example is the following rule:

$$\frac{\Gamma \vdash \mathit{isdef}(v^U) \qquad \Gamma \vdash (\langle\pi\ \ \mathtt{x}_1\mathtt{=e}_1\mathtt{;}\ \ \mathtt{x}_2\mathtt{=e}_2\mathtt{;}\ \ v\mathtt{=x}_1\mathtt{+x}_2\mathtt{;}\ \ \omega\rangle\phi)^U}{\Gamma \vdash (\langle\pi\ \ v\ \mathtt{=}\ \mathtt{e}_1\mathtt{+e}_2\mathtt{;}\ \ \omega\rangle\phi)^U} \tag{R3}$$

where v is a logical term, and $\mathtt{x}_1$ and $\mathtt{x}_2$ are new local variables. This rule has to be applied in case the expression $\mathtt{e}_1\mathtt{+e}_2$ is not a term; for example, the expression `(++i) + (++i)` has to be decomposed because the evaluation of its sub-expressions changes the state.

The premisses of the form $\Gamma \vdash \mathit{isdef}(v)$ in the above rules ensure that the expression v is defined in the state, i.e., its evaluation does not lead to a null pointer exception being thrown. That, for example, happens if $v = o.\mathtt{a}$ and the value of o is *null*. Other rules are available for handling this particular situation.

[9] 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 $\langle\pi p q\omega\rangle\phi$.

[10] A similar rule is defined for the case where the left side of the assignment is a local variable.

Rules for Update Simplification. In many cases, formulas and terms with an update can be simplified. For example, if $\mathtt{x}$ is a local variable, the term $\mathtt{x}^{v \leftarrow e}$ can be replaced by $\mathtt{x}$ in case $\mathtt{x} \neq v$ and by e in case $\mathtt{x} = v$. Another rule allows to replace a term of the form $(f(o))^{v \leftarrow e}$ by $f(o^{v \leftarrow e})$ if the function f does not depend on the state.

When no further simplification of a formula $\phi(o'.\mathtt{a}^{o.\mathtt{a} \leftarrow e})$ is possible, because the terms o and o' may be aliases for the same object, the following branching rule has to be applied:

$$\frac{\Gamma,\ o \doteq o' \ \vdash\ \phi(e) \qquad \Gamma,\ \neg(o \doteq o')\ \vdash\ \phi(o'.\mathtt{a})}{\Gamma\ \vdash\ \phi(o'.\mathtt{a}^{o.\mathtt{a} \leftarrow e})} \tag{R4}$$

where o and o' are terms of the same object type and $\mathtt{a}$ is an instance attribute, i.e., it is not declared `static`.

Rules for Creating Objects. The `new` statement is treated by the calculus as if it were a method implemented as follows (this implementation accesses the fields that are implicitly defined for all classes and array types, see the explanation in Section 3):

```
public static Cls new() {
  if (lastCreatedObj == null)
      lastCreatedObj = firstObj;
  else
      lastCreatedObj = lastCreatedObj.nextObj;
  lastCreatedObj.created = true;
  return lastCreatedObj;
}
```

Note, that this is a simplified version where object initialisation is not considered.

Rules for Loops. The following rule "unwinds" `while` loops. Its application is the prerequisite for symbolically executing the loop body. Similar rules are defined for `for` and `do-while` loops. These "unwind" rules allow to handle `while` loops if used together with induction schemata for the primitive and the user defined types (see below). Section 6 contains an example for the verification of a `while` loop.

$$\frac{\Gamma\ \vdash\ (\langle \pi\ \ l'\mathtt{:\{if(}c\mathtt{)}\,l''\mathtt{:\{}p'\mathtt{\}}\ \ l\,\mathtt{:while(}c\mathtt{)\{}p\mathtt{\}\}}\ \ \omega\rangle\phi)^{U}}{\Gamma\ \vdash\ (\langle \pi\ \ l\,\mathtt{:while(}c\mathtt{)\{}p\mathtt{\}}\ \ \omega\rangle\phi)^{U}} \tag{R5}$$

where l' and l'' are new labels, and p' is the result of (simultaneously) replacing in p (a) every `break` (with no label) that has the `while` loop as its target by `break` l', and (b) every `continue` (with no label) that has the `while` loop as its target by `break` l''.[11]

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

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

Rule R5 only applies to unlabelled `while` loops, i.e., in case π is not of the form $\pi'\, l$:; another rule is defined for labelled `while` loops.

From the general `while` rule (R5), the following simpler rules can be derived. The two rules are applicable if (a) the loop condition is a logical term c (and, thus, its evaluation does not have side effects), and (b) the loop body p does not contain any `break` or `continue` statements.

$$\frac{\Gamma \vdash \mathit{isdef}(c^U) \qquad \Gamma \vdash c^U \doteq \mathtt{true} \qquad \Gamma \vdash (\langle \pi \;\, p \;\; \mathtt{while}\,(c)\, p \;\; \omega\rangle\phi)^U}{\Gamma \vdash (\langle \pi \;\, \mathtt{while}\,(c)\, p \;\; \omega\rangle\phi)^U} \tag{R6}$$

$$\frac{\Gamma \vdash \mathit{isdef}(c^U) \qquad \Gamma \vdash c^U \doteq \mathtt{false} \qquad \Gamma \vdash (\langle \pi \;\; \omega\rangle\phi)^U}{\Gamma \vdash (\langle \pi \;\, \mathtt{while}\,(c)\, p \;\; \omega\rangle\phi)^U} \tag{R7}$$

Induction Rules. Induction schemata are available for the primitive type `byte` and all abstract types that are declared to be generated by constructors. The following rules are the induction schemata for `byte` and for an abstract type *list* generated by *cons* and *nil*:

$$\frac{\Gamma \vdash \psi(0) \qquad \Gamma \vdash (\forall x : \mathtt{byte})(\psi(x) \to \psi(x+1))}{\Gamma \vdash (\forall x : \mathtt{byte})\psi(x)} \tag{R8}$$

$$\frac{\Gamma \vdash \psi(\mathit{nil}) \qquad \Gamma \vdash (\forall l : \mathit{list})(\forall o : \mathtt{Object})(\psi(l) \to \psi(\mathit{cons}(o,l)))}{\Gamma \vdash (\forall l : \mathit{list})\psi(l)} \tag{R9}$$

Rules for Conditionals. Two rules are available for handling `if-then-else` statements: One rule for the case where the condition evaluates to true and one for the case where the condition evaluates to false:

$$\frac{\Gamma \vdash \mathit{isdef}(c^U) \qquad \Gamma \vdash c^U \doteq \mathtt{true} \qquad \Gamma \vdash (\langle \pi \;\, p \;\; \omega\rangle\phi)^U}{\Gamma \vdash (\langle \pi \;\, \mathtt{if}(c) \;\, p \;\, \mathtt{else} \;\, q \;\; \omega\rangle\phi)^U} \tag{R10}$$

$$\frac{\Gamma \vdash \mathit{isdef}(c^U) \qquad \Gamma \vdash c^U \doteq \mathtt{false} \qquad \Gamma \vdash (\langle \pi \;\, q \;\; \omega\rangle\phi)^U}{\Gamma \vdash (\langle \pi \;\, \mathtt{if}(c) \;\, p \;\, \mathtt{else} \;\, q \;\; \omega\rangle\phi)^U} \tag{R11}$$

These rules are only applicable if the condition `c` is a logical term. Otherwise, rules for the decomposition and evaluation of `c` have to be applied first.

Similar rules are defined for `if-then` without `else` and for the `switch` statement.

Rules for Handling Exceptions. The following rules allow to handle `try`-`catch`-`finally` blocks and the `throw` statement. These are restricted versions of the actual rules, they apply to the case where there is exactly one `catch` clause and one `finally` clause. And again, these rules are only applicable if both the exception *exc* that is thrown and the variable e that it is bound by the `catch` clause are logical terms. If they are more complex expressions, they first have to be decomposed and evaluated by applying other rules.

$$\frac{\Gamma \vdash \mathit{isdef}(\mathit{exc}^U) \qquad \Gamma \vdash \mathit{instanceof}(\mathit{exc}^U, T) \qquad \Gamma \vdash \mathit{isdef}(e^U) \qquad \Gamma \vdash (\langle \pi \ \texttt{try\{}e\texttt{=}\mathit{exc}\texttt{;}\ q\texttt{\}finally\{}r\texttt{\}}\ \omega\rangle\phi)^U}{\Gamma \vdash (\langle \pi \ \texttt{try\{throw}\ \mathit{exc}\texttt{;}\ p\texttt{\}catch(}T\ e\texttt{)\{}q\texttt{\}finally\{}r\texttt{\}}\ \omega\rangle\phi)^U} \tag{R12}$$

$$\frac{\Gamma \vdash \mathit{isdef}(\mathit{exc}^U) \qquad \Gamma \vdash \neg\mathit{instanceof}(\mathit{exc}^U, T) \qquad \Gamma \vdash (\langle \pi \ r\texttt{;}\ \texttt{throw}\ \mathit{exc}\texttt{;}\ \omega\rangle\phi)^U}{\Gamma \vdash (\langle \pi \ \texttt{try\{throw}\ \mathit{exc}\texttt{;}\ p\texttt{\}catch(}T\ e\texttt{)\{}q\texttt{\}finally\{}r\texttt{\}}\ \omega\rangle\phi)^U} \tag{R13}$$

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

Rule (R12) 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 (R13) applies. Rule (R14) applies if the `try` block is empty and, thus, terminates normally.

Rules for the `break` Statement. The following rule handles `break` statements:

$$\frac{\Gamma \vdash (\langle \pi \ \omega\rangle\phi)^U}{\Gamma \vdash (\langle \pi\, l\,\texttt{:\{}\pi'\ \texttt{break}\ l\,\texttt{;}\ \omega'\texttt{\}}\omega\rangle\phi)^U} \tag{R15}$$

where $\pi\, l\,\texttt{:\{}\pi'$ is a non-active prefix and $\texttt{\{}\pi'\ \texttt{break}\ l\,\texttt{;}\ \omega'\texttt{\}}$ is a block, i.e., the two braces in the conclusion of the rule are the opening and the closing brace of the same block.

Note, that according to the JAVA language specification, a label l is not allowed to occur within a block that is itself labelled with l. This ensures that the label l occurs only once in the prefix $\pi\, l\,\texttt{:\{}\pi'$.

Similar rules are defined for `break` statements without label and for the `continue` statement.

6 Example

As an example, we use the calculus presented in the previous section to prove that, if the while loop

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

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

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

is valid, where p_{while} is an abbreviation for the above while loop. Instead of proving (1) directly, we first use the induction rule (R8) to derive the sequence

$$\vdash \ (\forall n)((0 \leq n \wedge n \leq 10) \rightarrow (\langle p_{\text{while}} \rangle \mathtt{i} \doteq 10)^{\mathtt{i} \leftarrow 10-n}) \tag{2}$$

as a lemma (the logical variable n is of type `byte`). It basically expresses the same as (1), the difference is that its form allows it to be proved 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 tools would require user interaction.

Due to space restrictions, we only show the proof for the induction base $n = 0$; the proof for the induction step is omitted. The proof obligation for the induction base is

$$\vdash \ (0 \leq 0 \wedge 0 \leq 10) \rightarrow (\langle p_{\text{while}} \rangle \mathtt{i} \doteq 10)^{\mathtt{i} \leftarrow 10-0}) \tag{3}$$

which simplifies to

$$\vdash \ (\langle p_{\text{while}} \rangle \mathtt{i} \doteq 10)^{\mathtt{i} \leftarrow 10}$$

An application of the rule for `while` loops (R5) results in the new proof obligation

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

Now, the rule for conditionals with a condition that evaluates to true (R10) can be applied. This results in three new proof obligations:

$$\vdash \ \mathit{isdef}(\mathtt{true}^{\mathtt{i} \leftarrow 10}) \tag{4}$$

$$\vdash \ \mathtt{true}^{\mathtt{i} \leftarrow 10} \doteq \mathtt{true} \tag{5}$$

$$\vdash \ (\langle \mathtt{l1:\{l2:\{if\ (i==10)\ break\ l1;\ else\ i++;\}}\ p_{\text{while}}\mathtt{\}} \rangle \mathtt{i} \doteq 10)^{\mathtt{i} \leftarrow 10} \tag{6}$$

Sequences (4) and (5) can easily be shown to be valid. To prove sequence (6), we apply rule (R10) again and derive the proof obligations

$$\vdash \ \mathit{isdef}((\mathtt{i==10})^{\mathtt{i} \leftarrow 10}) \tag{7}$$

$$\vdash \ (\mathtt{i==10})^{\mathtt{i} \leftarrow 10} \doteq \mathtt{true} \tag{8}$$

$$\vdash \ (\langle \mathtt{l1:\{l2:\{break\ l1;\ else\ i++;\}}\ p_{\text{while}}\mathtt{\}} \rangle \mathtt{i} \doteq 10)^{\mathtt{i} \leftarrow 10} \tag{9}$$

Sequence (7) can easily be shown to be valid, as well as sequence (8), which can be simplified to $\vdash \ (\mathtt{10==10}) \doteq \mathtt{true}$.

To prove (9) to be valid, the rule for `break` statements (R15) has to be applied. The result is $\vdash (\mathtt{i} \doteq 10)^{\mathtt{i} \leftarrow 10}$. This simplifies to $\vdash 10 \doteq 10$ and can thus be shown to be valid.

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

$$\vdash (0 \leq 10 - \mathtt{i} \wedge 10 - \mathtt{i} \leq 10) \to (\langle p_{\text{while}} \rangle \mathtt{i} \doteq 10)^{\mathtt{i} \leftarrow 10-(10-\mathtt{i})}$$

which can be simplified to

$$\vdash (0 \leq \mathtt{i} \wedge \mathtt{i} \leq 10) \to (\langle p_{\text{while}} \rangle \mathtt{i} \doteq 10)^{\mathtt{i} \leftarrow \mathtt{i}} \tag{10}$$

And, since (10) is derivable, the original proof obligation (1) is derivable as well, because the trivial update $\mathtt{i} \leftarrow \mathtt{i}$ can be omitted.

7 Conclusion

Extensions and Future Work. We are currently implementing an interactive prover for our calculus as part of the KeY project. Such an implementation is a prerequisite for applying the calculus to more complex examples.

Further work is to prove soundness and relative completeness of the calculus w.r.t. a *formal* semantics. And we plan to extend the calculus with the concept of parameters (or meta-variables) that can be instantiated with logical terms "on demand" during the proof using unification. Meta-variables are the most important technique for automated deduction in classical logic, and this promises to make the automated proof search in Java Card DL much more efficient as well.

Related Work. There are many projects dealing with formal methods in software engineering including several ones aimed at Java as a target language. Work on the verification of Java programs includes [19, 13, 11, 17, 21]. The main difference of all these approaches to our work is that they use a Hoare logic instead of full DL, i.e., formulas and programs remain separated.

In [19], states are represented as terms of an abstract data type, whereas in our approach the states correspond to "worlds" in the models. They are not represented as terms but described with formulas. This allows to use the full expressiveness of DL to formalise the properties of a state.

Another important difference to other approaches is that abrupt termination, in particular exception handling, is either not treated at all or is treated in a completely different way (e.g. [11] where the reason for abrupt termination is made a part of the states, which leads to a more complex notion of states and of method return values).

Acknowledgements

I thank B. Sasse for working out the details of some of the rules of the calculus, and W. Ahrendt, T. Baar, M. Giese, E. Habermalz, R. Hähnle, W. Menzel, and P. H. Schmitt for many fruitful discussions and comments on earlier versions of this paper.

References

1. W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hähnle, W. Menzel, and P. 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. J. Alves-Foss, editor. *Formal Syntax and Semantics of Java.* LNCS 1523. Springer, 1999.
3. K. R. Apt. Ten years of Hoare logic: A survey – part I. *ACM Transactions on Programming Languages and Systems*, 1981.
4. T. Baar. Experiences with the UML/OCL-approach to precise software modeling: A report from practice. Available at i12www.ira.uka.de/~key, 2000.
5. E. Börger and W. Schulte. A programmer friendly modular definition of the semantics of Java. In Alves-Foss [2], pages 353–404.
6. C. Calcagno, S. Ishtiaq, and P. W. O'Hearn. Semantic analysis of pointer aliasing, allocation and disposal in Hoare logic. In *Proceedings, International Conference on Principles and Practice of Declarative Programming, Montreal, Canada*. ACM, 2000.
7. E. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.
8. D. L. Dill and J. Rushby. Acceptance of formal methods: Lessons from hardware design. *IEEE Computer*, 29(4):23–24, 1996. Part of: Hossein Saiedian (ed.). *An Invitation to Formal Methods.* Pages 16–30.
9. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification.* Addison Wesley, second edition, 2000.
10. M. G. Hinchey and J. P. Bowen, editors. *Applications of Formal Methods.* Prentice Hall, 1995.
11. M. Huisman and B. 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.
12. D. Hutter, B. Langenstein, C. Sengler, J. H. Siekmann, and W. Stephan. Deduction in the Verification Support Environment (VSE). In M.-C. Gaudel and J. Woodcock, editors, *Proceedings, International Sympoium of Formal Methods Europe (FME), Oxford, UK*, LNCS 1051. Springer, 1996.
13. B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about Java classes (preliminary report). In *Proceedings, Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 329–340. ACM Press, 1998.
14. D. Kozen and J. 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.

15. J. Martin and J. J. Odell. *Object-Oriented Methods: A Foundation, UML Edition.* Prentice-Hall, 1997.
16. T. Nipkow and D. von Oheimb. Machine-checking the Java specification: Proving type safety. In Alves-Foss [2], pages 119–156.
17. T. Nipkow, D. von Oheimb, and C. Pusch. μJava: Embedding a programming language in a theorem prover. In F. L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation.* IOS Press, 2000. To appear.
18. Object Management Group, Inc., Framingham/MA, USA, www.omg.org. *OMG Unified Modeling Language Specification, Version 1.3*, June 1999.
19. A. Poetzsch-Heffter and P. 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.
20. W. 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.
21. D. von Oheimb. Axiomatic semantics for Java^{light}. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Proceedings, Formal Techniques for Java Programs, Workshop at ECOOP'00, Cannes, France*, 2000.