The KeY Approach: Integrating Design and Formal Verification of Java Card Programs

Wolfgang Ahrendt[†] Thomas Baar[†] Bernhard Beckert[†]
Martin Giese[†] Elmar Habermalz[†] Reiner Hähnle[‡]
Wolfram Menzel[†] Peter H. Schmitt[†]

[†] Universität Karlsruhe
Inst. f. Logik, Komplexität
und Deduktionssyteme
D-76128 Karlsruhe, Germany

[‡] Chalmers University
of Technology
Dept. of Computing Science
S-41296 Gothenburg, Sweden

`i12www.ira.uka.de/~key`

Abstract

This paper reports on the ongoing KeY project aimed at bridging the gap between (a) object-oriented software engineering methods and tools and (b) deductive verification for the development of Java Card programs. In particular, we describe a Dynamic Logic for Java Card and outline a sequent calculus for this logic that axiomatises Java Card and is used in the verification component of the KeY system.

1 Introduction

The goal of the KeY project[1] (read "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 to be verified should be written in a "real" object-oriented (OO) programming language.
- The logical formalism should be as easy as possible to use for software developers (that do not have years of training in formal methods).

We decided to use Java Card as our target language. Since it is a "real" OO language, it has features that are difficult to handle in a software verification system, such as dynamic data structures, exceptions, and initialisation. But, 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: (a) Java Card applications are small; (b) at the same time, they are embedded into larger program systems or business processes which should be modeled (though not necessarily formally

[1]The KeY project was launched in November 1998. It is supported by the Deutsche Forschungsgemeinschaft (grant no. Ha 2617/2-1).

verified); (c) JAVA CARD applications are often security-critical, giving incentive to apply formal methods; (d) the high number of deployed smart cards constitutes a new motivation for formal verification, as arbitrary updates are not feasible.

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.

2 Analysis of the Current Situation

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 [11]. The real problem lies in the excessive demand imposed by current tools on the skills of prospective users: (1) Tools for formal software specification and verification are not integrated into industrial software engineering processes. (2) User interfaces of verification tools are not ergonomic: they are complex, idiosyncratic, and are often without graphical support. (3) Users of verification tools are expected to know syntax and semantics of one or more complex formal languages. Typically, at least a tactical programming language and a logical language are involved. And even worse, to make serious use of many tools, intimate knowledge of employed logic calculi and proof search strategies is necessary.

Successful specification and verification of larger projects, therefore, is done separately from software development by academic specialists with several years of training in formal methods, in many cases by the tool developers themselves. It is unlikely that formal software specification and verification will become a routine task in industry under these circumstances.

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 minimized. Moreover, techniques involving formal software specification and verification must be teachable in a structured manner. They should be integrated in courses on software engineering topics.

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. Even worse, in a linear model the extra effort needed for verification cannot be parallelized and thus compensated by greater work force.

3 The KeY Approach

The KeY project addresses the goals outlined in the previous section (here, we can only give a brief overview of the KeY project; a detailed description can be found in [1]).

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:

- 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 [16].
- 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 an axiomatic semantics of Java Card (see Section 6).
- 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 [3]).

4 The KeY System

A first KeY system prototype has been implemented, integrating the CASE tool TogetherJ and a deductive component (it has only limited capabilities and lacks the verification manager 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. The main components are described below.

4.1 The Modeling Component

This component is based on the CASE tool and is responsible for all user interactions (except interactive deduction). It is used to generate and refine models, and to store and process them. The extensions for precise modeling contains, e.g., editor and parser for the OCL. Additional functionality for the verification process is provided, e.g., for writing proof obligations.

4.2 The Verification Manager

This is the link between the modeling component and the deduction component. It generates proof obligations expressed in formal logic from the refinement relations in the model. It stores and processes partial and completed proofs; and it is responsible for correctness management (to make sure, e.g., that there are no cyclic dependencies in proofs).

4.3 The Deduction Component

The KeY system comprises a deductive component that can handle Dynamic Logic for JAVA CARD (see Section 6). It is used to actually construct proofs—or counter examples—for proof obligations generated by the verification manager. It is based on an interactive verification system combined with powerful automated deduction techniques that increase the degree of automation; it also contains a part for automatically generating counter examples from failed proof attempts. The interactive and automated techniques and those for finding counter examples are fully integrated and operate on the same data structures.

For interactive proof search, a technique of *schematic theory specific rules* is used, which combine purely logical knowledge, information on how this knowledge should be used, and information on when and where this knowledge should be presented for interactive use.[2].

Both automated and interactive deduction use the same data structures and proof rules [10].

5 The Modeling Process

Software development is generally divided into four activities: analysis, design, implementation, and test. The KeY approach embraces verification as a fifth category. The way in which the development activities are arranged in a sequential order over time is called software development *process*. It consists of different phases. The end of each phase is defined by certain criteria the actual model should meet (milestones).

In some older process models like the waterfall model or Boehm's spiral model no difference is made between the main activities—analysis, design, implementation, test—and the process phases. More recent process models distinguish between phases and activities very carefully; for example, the Rational Unified Process [14] uses the phases inception, elaboration, construction, and transition along with the above activities.

The KeY system does neither support nor require the usage of a *particular* process. However, it is taken into account that most modern processes have two principles in common. They are *iterative* and *incremental*. The design of an iteration is often regarded as the refinement of the design developed in the previous iteration. This has an influence on the way in which the KeY system treats UML models and additional verification tasks. The verification activities are spread across all phases in software development. They are often carried out after test activities.

The diagrams of the Unified Modeling Language provide, in principle, an easy and concise way to formulate various aspects of a specification, however [24, foreword]: "[...] there are many subtleties and nuances of meaning diagrams cannot convey by themselves." This was a main source of motivation for the development

[2]This technique has been implemented in the interactive proof system IBIJa (more information on IBIJa is available at i11www.ira.uka.de/~ibija).

of the Object Constraint Language (OCL), part of the UML since version 1.3. Constraints written in this language are understood in the context of a UML model, they never stand by themselves. The OCL allows to attach preconditions, postconditions, invariants, and guards to elements of a UML model.

When designing a system with KeY, one develops a UML model that is enriched by OCL constraints to make it more precise. This is done using the CASE tool integrated into the KeY system. To assist the user, the KeY system provides menu and dialog driven input possibility. Certain standard tasks, for example, generation of formal specifications of inductive data structures (including the common ones such as lists, stacks, trees) in the UML and the OCL can be done in a fully automated way, while the user simply supplies names of constructors and selectors. Even if formal specifications cannot fully be composed in such a schematic way, considerable parts usually can.

In addition, we have developed a method supporting the extension of a UML model by OCL constraints that is based on enriched design patterns. In the KeY system we will provide common patterns that come complete with predefined constraint schemata. These schemata are formulated in a language that is a slight extension of OCL. They are flexible and allow the user to easily generate well-adapted constraints for the different instances of a pattern [4]. Thus, the user needs not write formal specifications from scratch.

6 Verification of Java Card Programs

6.1 Dynamic Logic

We use Dynamic Logic (DL) [15]—an extension of Hoare logic [2]—as the logical basis of the KeY system's software verification component. We believe that this is a good choice as 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. In this section, we give a short account of our JAVA CARD DL; see [5] for a more detailed description.

DL is successfully used in the KIV software verification system [20] for a programming language that is not object-oriented; and Poetzsch-Heffter and Müller's definition of a Hoare logic for a JAVA subset [19] shows that there are no principal obstacles to adapting the DL/Hoare approach to OO languages.

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 legal JAVA CARD program); $\langle p \rangle$ refers to the successor worlds (called states in the DL framework) that are reachable by running the program p. In classical 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 $\phi \to \langle p \rangle \psi$ is valid if for every state s satisfying precondition ϕ a run of the program p starting in s terminates, and in the terminating state the postcondition 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, 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 (e.g., `instanceof`) are available in DL for the description of states. It is, therefore, not necessary to define an abstract data type *state* and to represent states as terms of that type (as has, for example, been done in [19]); 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 DL (that uses a simple "artificial" programming language), a DL for a "real" OO programming language like JAVA CARD has to cope with the following complications: (1) 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. (2) The evaluation of a JAVA expression may have side effects; thus, there is a difference between an expression and a logical term. (3) Language features such as built-in data types, exception handling, and object initialisation have to be handled.

6.2 Syntax of Java Card DL

We do not allow class definitions in the programs that are part of DL formulas, but define syntax and semantics of DL formulas w.r.t. a given JAVA CARD program (the context), i.e., a sequence of class definitions. The programs in DL formulas are executable code. The (basic) programs are all the legal JAVA CARD statements, including: (a) expression statements such as assignments, method calls, `new`-statements, etc.; (b) blocks and compound statements built with `if-else`, `switch`, `for`, `while`, and `do-while`; (c) statements with exception handling using `try-catch-finally`; and (d) statements that abruptly redirect the control flow (`continue`, `return`, `break`, `throw`).

We allow programs in DL formulas (not in the context) to contain logical terms. Wherever a JAVA CARD expression can be used, a term of the same type as the expression can be used as well. Accordingly, expressions can contain terms (but not vice versa).

Formulas are built as usual from the (logical) terms, the predicate symbols (including the equality predicate $\doteq$), 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 $\langle p \rangle$, i.e., if p is a program and ϕ is a formula, then $\langle p \rangle \phi$ is a formula as well.

6.3 Semantics of Java Card DL

The models of DL consist of program states. These states share the same universe containing a sufficient number of elements of each type. In each state a (possibly different) value (an element of the universe) of the appropriate type is assigned to: (a) the program variables, (b) the attributes (fields) of all objects, (c) the class attributes (static fields) of all classes in the context, and (d) the special object variable `this`. Variables and attributes of object types can be assigned the special value *null*. States do not contain any information on control flow such as a program counter or the fact that an exception has been thrown.

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

We consider programs that terminate abnormally to be non-terminating, such that nothing can be said about their final state. Examples are a program that throws an uncaught exception and a `return` statement that is not within the boundaries

$$\frac{\Gamma \vdash \mathit{cnd} \doteq \mathtt{true} \qquad \Gamma \vdash \langle \pi\ \mathit{prg}\ \mathtt{while}\,(\mathit{cnd})\,\mathit{prg}\ \ \omega\rangle\phi}{\Gamma \vdash \langle \pi\ \mathtt{while}\,(\mathit{cnd})\,\mathit{prg}\ \ \omega\rangle\phi} \tag{1}$$

$$\frac{\Gamma \vdash \mathit{cnd} \doteq \mathtt{false} \qquad \Gamma \vdash \langle \pi\omega\rangle\phi}{\Gamma \vdash \langle \pi\ \mathtt{while}\,(\mathit{cnd})\,\mathit{prg}\ \ \omega\rangle\phi} \tag{2}$$

$$\frac{\Gamma \vdash \mathit{instanceof}(\mathit{exc}, T) \qquad \Gamma \vdash \langle \pi\ \mathtt{try\{}\mathit{e}\mathtt{=}\mathit{exc}\mathtt{;}\ \mathit{q}\mathtt{\}finally\{}\mathit{r}\mathtt{\}}\ \ \omega\rangle\phi}{\Gamma \vdash \langle \pi\ \mathtt{try\{throw}\ \mathit{exc}\mathtt{;}\ \mathit{p}\mathtt{\}catch(}\mathit{T}\ \ \mathit{e}\mathtt{)\{}\mathit{q}\mathtt{\}finally\{}\mathit{r}\mathtt{\}}\ \ \omega\rangle\phi} \tag{3}$$

$$\frac{\Gamma \vdash \neg\mathit{instanceof}(\mathit{exc}, T) \qquad \Gamma \vdash \langle \pi\ \ \mathit{r}\mathtt{;}\ \mathtt{throw}\ \mathit{exc}\mathtt{;}\ \ \omega\rangle\phi}{\Gamma \vdash \langle \pi\ \mathtt{try\{throw}\ \mathit{exc}\mathtt{;}\ \mathit{p}\mathtt{\}catch(}\mathit{T}\ \ \mathit{e}\mathtt{)\{}\mathit{q}\mathtt{\}finally\{}\mathit{r}\mathtt{\}}\ \ \omega\rangle\phi} \tag{4}$$

$$\frac{\Gamma \vdash \langle \pi\ \ \mathit{r}\ \ \omega\rangle\phi}{\Gamma \vdash \langle \pi\ \mathtt{try\{\}catch(}\mathit{T}\ \ \mathit{e}\mathtt{)\{}\mathit{q}\mathtt{\}finally\{}\mathit{r}\mathtt{\}}\ \ \omega\rangle\phi} \tag{5}$$

Table 1: Some of the rules of our calculus for Java Card DL.

of a method invocation. Thus, for example, $\langle$`throw x;`$\rangle\phi$ is unsatisfiable for all ϕ.[3]

6.4 A Sequent Calculus for Java Card DL

Here, we outline the ideas behind our sequent calculus for JAVA CARD DL, and we give some of the basic rules.[4]

The DL 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` blocks, etc. 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 be handled appropriatly.[5]

As examples, we present the rules for `while` loops and for exception handling, respectively. Many more rules are needed to handle all language features of JAVA CARD.

These rules operate on sequents $\Gamma \vdash \phi$. The semantics of a sequent is that the conjunction of the DL formulas in Γ implies the DL formula ϕ. Sequents are used to represent proof obligations, proof (sub-)goals, and lemmata.

Rules (1) and (2) in Table 1 allow to "unwind" `while` loops. These are simplified versions that only work if (a) the condition cnd is a logical term (and, thus, its evaluation does not have side effects), and (b) the program prg does not contain a `continue` statement. These rules allow to handle loops if used in combination with induction schemata. Similar rules are defined for `do-while` and `for` loops.

Rules (3)–(5) in Table 1 allow to handle `try-catch-finally` blocks and the `throw` statement. Again, these are simplified versions of the actual rules; they are only applicable if (a) exc is a logical term (e.g., a program variable), and (b) the statements `break`, `continue`, `return` do not occur. Rule (3) applies if an

[3]Nevertheless, it is possible to express and (if true) prove the fact that a program p terminates abnormally using a DL formula. For example, $\langle$`try{`p`}catch{Exception e}`$\rangle(\neg\, \mathtt{e} \doteq \mathtt{null})$ expresses that p throws an exception.

[4]These are simplified versions of the actual rules. In particular, initialisation of objects and classes is not considered.

[5]In classical DL, 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 $\langle \pi p q\omega\rangle\phi$.

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 (4) applies. Rule (5) applies if the `try` block is empty and, thus, terminates normally.

7 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, 12, 17, 23]. There is also work on security of JAVA CARD and ACTIVEX applications as well as on secure smart card applications in general. We are, however, not aware of any project quite like ours. We mention some of the more closely related projects. The COGITO project [22] resulted in an integrated formal software development methodology and support system based on extended Z as specification language and Ada as target language. It is not integrated into a CASE tool, but stand-alone. The FUZE project [9] realized CASE tool support for integrating the FUSION OOAD process with the formal specification language Z (the aim was to formalize OOAD methods and notations such as the UML). The QUEST project's [21] goal is to enrich the CASE tool AUTOFOCUS for description of distributed systems with means for formal specification and support by model checking. Aim of the SYSLAB project [6] is the development of a scientifically founded approach for software and systems development. At the core is a precise and formal notion of hierarchical "documents" consisting of informal text, message sequence charts, state transition systems, object models, specifications, and programs. The goal of the PROSPER project[6] was to provide the means to deliver the benefits of mechanized formal specification and verification to system designers in industry. The difference to the KeY project is that the dominant goal is hardware verification.

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. Technical Report 2000/4, University of Karlsruhe, Department of Computer Science, Jan. 2000.

[2] K. R. Apt. Ten years of Hoare logic: A survey – part I. *ACM Transactions on Programming Languages and Systems*, 1981.

[3] T. Baar. Experiences with the UML/OCL-approach to precise software modeling: A report from practice. Available at i12www.ira.uka.de/~key, 2000.

[4] T. Baar, T. Sattler, R. Hähnle, and P. H. Schmitt. Entwurfsmustergesteuerte Erzeugung von OCL-Constraints. In *Proceedings, Softwaretechnik 2000, Berlin, Germany*, 2000. To appear. Available at i12www.ira.uka.de/~key.

[5] B. Beckert. A dynamic logic for Java Card. In *Proceedings, Formal Techniques for Java Programs, Workshop at ECOOP'00, Cannes, France*, 2000. Available at i12www.ira.uka.de/~key.

[6] R. Breu, R. Grosu, F. Huber, B. Rumpe, and W. Schwerin. Towards a precise semantics for object-oriented modeling techniques. In H. Kilov and B. Rumpe, editors, *Proceedings, Workshop on Precise Semantics for Object-Oriented Modeling Techniques at ECOOP'97*. Technical University of Munich, Technical Report TUM-I9725, 1997.

[7] E. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[6]See www.dcs.gla.ac.uk/prosper/index.html.

[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] R. B. France, J.-M. Bruel, M. M. Larrondo-Petrie, and E. Grant. Rigorous object-oriented modeling: Integrating formal and informal notations. In M. Johnson, editor, *Proceedings, Algebraic Methodology and Software Technology (AMAST), Berlin, Germany*, LNCS 1349. Springer, 1997.

[10] M. Giese. Integriertes automatisches und interaktives Beweisen: Die Kalkülebene. Diploma Thesis, Fakultät für Informatik, Universität Karlsruhe, June 1998. In German. Available at i11www.ira.uka.de/~giese/da.ps.gz.

[11] M. G. Hinchey and J. P. Bowen, editors. *Applications of Formal Methods.* Prentice Hall, 1995.

[12] 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.

[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] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process.* Object Technology Series. Addison-Wesley, 1999.

[15] 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.

[16] J. Martin and J. J. Odell. *Object-Oriented Methods: A Foundation, UML Edition.* Prentice-Hall, 1997.

[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] O. Slotosch. Overview over the project QUEST. In *Applied Formal Methods, Proceedings of FM-Trends 98, Boppard, Germany*, LNCS 1641, pages 346–350. Springer, 1999.

[22] O. Traynor, D. Hazel, P. Kearney, A. Martin, R. Nickson, and L. Wildman. The Cogito development system. In M. Johnson, editor, *Proceedings, Algebraic Methodology and Software Technology (AMAST), Berlin*, LNCS 1349, pages 586–591. Springer, 1997.

[23] 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.

[24] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modelling with UML.* Object Technology Series. Addison-Wesley, 1999.