W04. Formal Techniques for Java Programs

Gary T. Leavens, Sophia Drossopoulou, Susan Eisenbach,
Arnd Poetzsch-Heffter, and Erik Poll

leavens@cs.iastate.edu, scd@doc.ic.ac.uk, se@doc.ic.ac.uk,
Arnd.Poetzsch-Heffter@Fernuni-Hagen.de, erikpoll@cs.kun.nl

Abstract. This report gives an overview of the third ECOOP Workshop on Formal Techniques for Java Programs. It explains the motivation for such a workshop and summarizes the presentations and discussions.

1 Introduction

The ECOOP 2001 workshop on Formal Techniques for Java Programs was held in Budapest, Hungary. It was a follow-up for last year's ECOOP workshop on the same topic [DEJ$^+$00b] [DEJ$^+$00a], the first ECOOP workshop on this topic [JLMPH99], and the Formal Underpinnings of the Java Paradigm workshop held at OOPSLA '98 [Eis98]. The workshop was organized by Susan Eisenbach (Imperial College, Great Britain), Gary T. Leavens (Iowa State University, USA), Peter Müller (FernUniversität Hagen, Germany), Arnd Poetzsch-Heffter (FernUniversität Hagen, Germany), and Erik Poll (University of Nijmegen, The Netherlands). Besides the organizers the program committee of the workshop included Gilad Bracha (Sun Microsystems, USA), Sophia Drossopoulou (Imperial College, Great Britain), Doug Lea (State University of New York at Oswego, USA), and Rustan Leino (Compaq Computer Corporation, USA). The program committee was chaired by Sophia Drossopoulou.

There was lively interest in the workshop. Out of many submissions, the organizers selected 13 papers for longer presentations, and three for short presentations. There was one invited talk, given by Gilad Bracha of Sun Microsystems. Overall, 36 people from 30 universities, research labs, and industries attended the workshop.

Motivation. Formal techniques can help to analyze programs, to precisely describe program behavior, and to verify program properties. Applying such techniques to object-oriented technology is especially interesting because:

- the OO-paradigm forms the basis for the software component industry with their need for certification techniques.
- it is widely used for distributed and network programming.
- the potential for reuse in OO-programming carries over to reusing specifications and proofs.

Such formal techniques are sound, only if based on a formalization of the language itself.

Java is a good platform to bridge the gap between formal techniques and practical program development. It plays an important role in these areas and is becoming a de facto standard because of its reasonably clear semantics and its standardized library.

However, Java contains novel language features, which are not fully understood yet. More importantly, Java supports a novel paradigm for program deployment, and improves interactivity, portability and manageability. This paradigm opens new possibilities for abuse and causes concern about security.

Thus, work on formal techniques and tools for Java programming and formal underpinnings of Java complement each other. This workshop aims to bring together people working in these areas, in particular on the following topics:

– specification techniques and interface specification languages
– specification of software components and library packages
– automated checking and verification of program properties
– verification technology and logics
– Java language semantics
– dynamic linking and loading, security

Structure of Workshop and Report. The one-day workshop consisted of a technical part during the day and a workshop dinner in the evening. While the food in Budapest was delightful, this report deals only with the technical aspects of the workshop.

The presentations at the workshop were structured as follows.

– 9:00 10:00 Opening Session, and Invited Talk by Gilad Bracha: "Adventures in Computational Theology: Selected Experiences with the Java(tm) Programming Language "
– 10:15 11:15 Language Semantics I
  • Alessandro Coglio: "Improving the Official Specification of Java Bytecode Verification"
  • Kees Huizing and Ruurd Kuiper: "Reinforcing Fragile Base Classes"
– 11:25 12:25 Language Semantics II
  • Davide Ancona, Giovanni Lagorio, and Elena Zucca: "Java Separate Type Checking is not Safe"
  • Mirko Viroli : "From FGJ to Java according to LM translator"
  • Mats Skoglund and Tobias Wrigstad : "A mode system for read-only references in Java"
– 13:45 15:45 Specification and Verification I (Java Card)
  • Pierre Boury and Nabil Elkhadi : "Static Analysis of Java Cryptographic Applets"
  • Peter Müller and Arnd Poetzsch-Heffter : "A Type System for Checking Applet Isolation in Java Card"

  - Gilles Barthe, Dilian Gurov, and Marieke Huisman "Compositional specification and verification of control flow based security properties of multi-application programs"
- 16:00 17:30 Specification and Verification II
  - Peter Müller, Arnd Poetzsch-Heffter, Gary T. Leavens : "Modular Specification of Frame Properties in JML"
  - John Boyland : "The Interdependence of Effects and Uniqueness"
  - Ana Cavalcanti and David Naumann : "Class Refinement for Sequential Java"
  - Joachim van den Berg, Cees-Bart Breunesse, Bart Jacobs, and Erik Poll : "On the Role of Invariants in Reasoning about Object-Oriented Languages"
- 17:45 18:30 Short presentations and closing session
  - Claus Pahl: "Formalising Dynamic Composition and Evolution in Java Systems"
  - M. Carbone, M. Coccia, G. Ferrari and S. Maffeis: "Process Algebra-Guided Design of Java Mobile Network Applications"
  - Peep Küngas, Vahur Kotkas, and Enn Tyugu : "Introducing Meta-Interfaces into Java"

The rest of this report is structured as follows: Sections 2 to 6 summarize the presentations and discussions of the technical sessions of the workshop. The conclusions are contained in Section 7. A list of participants with email addresses can be found in the Appendix. The workshop proceedings are contained in [DEL$^+$01].

2 Language Semantics I

Because Java security is based on type safety, correct implementation of bytecode verification is of paramount importance to the security of an implementation of the Java Virtual Machine (JVM). In his contribution [Cog01a], Alessandro Coglio provided a comprehensive analysis of the official specification of bytecode verification and explained techniques to overcome the shortcomings. The insight underlying the analysis was gained during a complete formalization of Java bytecode verification [Cog01b]. The critique on the official specification lists places of redundancy, unclear terminology (e.g., static and structural constraints are both static in nature which might lead to confusion), lack of explanation, contradictions (e.g., the specification says that uninitialized objects cannot be accessed as well as that a constructor can store values into some fields of the object that is being initialized), and errors that were reported elsewhere.

Based on this critique, Coglio suggested improvements to the JVM specification. He discussed the merging of reference types and suggested that sets of type names should be used to express the merged types. This is an improvement over solutions that are based on existing common supertypes, because it avoids premature loading. For checking the subtype relation, he also proposed a solution that does not need premature loading. The basic idea is to generate subtype constraints at verification time. These constraints are checked when additional classes are loaded.

Furthermore, Coglio suggested that Java should make a clear distinction between *acceptable* code, i.e., code that does not lead to runtime type errors, and *accepted* code, i.e., code that satisfies declarative and decidable conditions guaranteeing type safety. Since it is, in general, undecidable whether a bytecode sequence is acceptable, the bytecode specification should define what accepted code is and compilers should be constraint to generate only accepted code.

The second presentation of this session given by K. Huizing is discussed in section "Specification and Verification II", because its topic is closer related to the other papers of that session.

3 Language Semantics II

Davide Ancona, Giovanni Lagorio, and Elena Zucca [ALZ01] looked at the problem of ensuring type safety for separately-compiled language fragments. They showed that with current Java development systems it is possible to compile fragments, alter and recompile some of them and in the process introduce type errors that are only caught at link time. They believe that these errors should have been caught earlier, in the compilation process.

Ancona, Lagorio, and Zucca went on to propose a formal framework for expressing separate compilation. They defined a small subset of Java and conjecture that within their framework, type errors would be caught at compilation rather than linking time. Their scheme would mean that a programmer could always have a single set of source fragments, that could be recompiled together to produce the current executable, a property not held by actual Java programs.

Mirko Virolli [Vir01] described work he is doing on adding parametric polymorphism to Java starting with Featherweight Generic Java [IPW99]. Parametric polymorphism is added via a translator called LM and the translation process is complex. To understand the complexity, a formalization of LM has been done so reasoning about its properties is possible. The LM translator is modeled as a compilation of FGJ into full Java. The model should help with the implementation of a correct translator.

Skoglund and Tobias Wrigstad [SW01] proposed an extension of Java syntax (with type rules) to include modes that will restrict some mutations of shared objects. There are four proposed modes, which include **read**, **write**, **any** and **context**. Objects in **read** mode cannot be altered, although objects in **write** mode can be. Objects in **any** and **context** mode are readable or writable depending on the context. The system is statically checkable. In addition to the modes, there is a dynamic construct called **caseModeOf** which enables the programmer to write code depending on the mode of an object. Java's lack of a construct (unlike C++) to restrict the altering of objects has been tackled by others, but the authors believe earlier attempts to solve this problem are too restrictive.

4 Specification and Verification I (Java Card)

Three of the papers presented at the workshop were about Java Card, a simplified version of Java designed for programming smart cards, which has been attracting a lot of attention in the formal methods community. The small size of Java Card programs, which have to run on the extremely limited hardware available on a smart card, and the vital importance of correctness of typical smart card applications, notably for bank cards and mobile phone SIMs, make Java Card an ideal application area for formal techniques. Indeed, both the smart card industry and their customers have recognized the important role that formal methods can play in ensuring that the (very high) security requirements for smart card applications are satisfied.

The paper by Pierre Boury and Nabil Elkhadi [BE01] was about a technique for static analysis of confidentiality properties of Java Card programs (also called "applets").

Java Card programs typically rely on cryptography to ensure security properties, such as authentication (e.g., of a mobile phone SIM to the mobile phone network) and confidentiality (by the creation of secure channels). Even though standard cryptographic algorithms are used for this, ensuring that these are used correctly in a program, so that no confidential data is ever disclosed (leaked) to unauthorized parties, is notoriously difficult. The approach taken by the authors has been to adapt formal models that have been proposed for the analysis of cryptographic protocols to the verification of confidentiality properties of Java Card applets, using the technique of abstract interpretation. The outcome of this work is an tool, called "StuPa", which can automatically verify confidentiality properties of Java Card class files.

The paper by Gilles Barthe, Dilian Gurov, and Marieke Huisman [BGH01] considered another class of security properties for Java Card programs, namely properties about control flow.

Global properties about a collection of interacting Java Card applets on a single smart card are expressed in temporal logic. The idea is then that "local" properties about the individual applets are found, and it is then formally proved that these local properties together imply the required global properties. The proof system used to prove that the local properties imply the required global ones is based on the modal μ-calculus. To verify the local properties an existing technique, which uses an essentially language independent program model, is instantiated to Java Card. The local properties of the individual applets are verified automatically, using model checking.

The paper by Peter Müller and Arnd Poetzsch-Heffter [MPH01] is an application of type system ideas to the Java Card language. In Java Card, applets are isolated in the sense that they are "not allowed to access fields and methods of other applets on the same smart-card." (If this happens anyway, a `SecurityException` is supposed to be thrown by the smart-card's virtual machine.) The problem addressed is how to statically check applet isolation in Java

Card programs, which would avoid the problem of producing lots of Java smart-cards that have code that could cause `SecurityExceptions`. The type system in the paper prevents such errors, using mostly static checks.

The approach taken to this problem is to adopt the author's previous work on type systems for alias control [MPH00]. The type system tracks what context owns each reference in a conservative manner, and thus can prohibit accesses to other contexts that would otherwise cause a `SecurityException`. The checking is not completely static, because in some cases one must use Java-style downcasts to recover exact information about the context to which a reference belongs. Besides easing debugging, this type system could also lead to improvements in the runtime overhead experienced by Java Card programs, provided the execution engine takes the previous work of the type checker into account.

5 Specification and Verification II

The paper by Kees Huizing and Ruurd Kuiper [HK01] contained a new analysis of the fragile base class problem. The fragile base class problem arises when one has a correct base class (i.e., a superclass), a correct derived class (i.e., a subclass), and then changes the base class in such a way that it is still correct, but the derived class is no longer correct [MS98]. One difficulty is that the programmers working on the base class may have no idea of what derived classes exist, so they have no way of ensuring the correctness of the derived classes. This points to a weakness of the specification of the base class.

The analysis by Huizing and Kuiper focuses in particular on the specification of class invariants. The problem, they say, is that the derived class may have a stronger invariant than the base class (as permitted by standard definitions of behavioral subtyping), but that this stronger invariant is not preserved by methods of the base class. They describe a specification formalism, called "cooperative contracts" to avoid this problem, as well as a stronger notion of behavioral subtyping. This stronger notion of behavioral subtyping (called "reinforced behavioral subtyping"), requires that every non-private method inherited from the base class (i.e., which is not not overridden in the derived class), does, in fact, preserve the derived class's possibly stronger invariant. The programmer of the derived class can use the source code of the base class in this proof, and can override base class methods to meet this proof obligation as well. However, in some cases, to allow reasonable derived classes and the use of method inheritance, the authors further propose stronger forms of method specification. In particular, a "cooperative contract" allows the postcondition of a method to refer to the postconditions of other methods it calls. The developer of the base class can plan ahead for future derived classes by referring to the postconditions of other methods in its method specifications. These parameterized postconditions change their meaning in the context of a derived class, since the specifications of the methods of a derived class will refer to the overriding method.

The extended abstract by Cavalcanti and Naumann [CN01], describes "ongoing" work "on refinement calculus for sequential Java." The refinement calculus is a formalism for systematic development of correct programs [Mor90]. Refinements can involve both algorithmic and data refinements. A data refinement replaces a data structures and the methods that manipulate them with different data structures and methods. The paper extends these ideas to classes, by allowing one to replace the implementation of a class (including its fields and methods) with another implementation. In the refinement calculus, such a refinement step must be proved correct by exhibiting a forward simulation relation, which relates the states of the original (abstract) class to those of the replacement (concrete) class. The paper states a soundness theorem for this form of data refinement.

The work contains two "surprises." The first is that the forward simulation must be surjective in the sense that there cannot be any concrete values that do not represent some abstract values. This requirement is needed in a language that can have uninitialized variables, or call-by value-result or result. The second surprise was that the simulation relation has to be total if the language has angelic variables (or an equally powerful construct, such as specification statements). Another interesting aspect discussed is that, apparently, the results show that an equality test that tests the identity of objects does not preserve data refinement.

The paper by Peter Müller, Arnd Poetzsch-Heffter, and Gary Leavens [MPHL01] tackles the notorious problem of modular verification of frame properties. Frame properties specify which locations (i.e., fields) may be modified by a method. For an object oriented language it should be possible to specify such properties in an abstract way, so that subclasses have the freedom to introduce additional fields that can be modified. Clearly one wants to reason about these properties in a modular way, so that frame properties can be specified and verified of an individual class irrespective of the context in which it is used, and irrespective of additional (sub)classes that may exists in this context. Finding a proof system that allows modular verification of frame properties has been recognized as an important open problem, and has attracted a lot of attention in the past few years.

This paper presents a concrete proposal to extend the behavioral interface specification language JML (Java Modeling Language [LBR01]) for Java with constructs to specify frame properties in a modular way, and a sound proof system allowing modular verification of these frame properties. The technique is based on a more general framework for the modular verification of Java programs introduced in the recent PhD thesis of Peter Müller [Mül01], which in turn builds on work by Rustan Leino [Lei95]. Key to the whole approach is an ownership model, in which the object store (i.e., the heap) is partitioned into a hierarchy of so-called "universes" and which imposes restrictions on references between different universes. This alias control type system allows frame properties of a method to only refer to "relevant" locations, so that the frame properties are effectively underspecified. A client calling a method can use the methods specification to reason about the called method's relevant locations, and can reason directly about abstract locations that may depend on them.

The paper by John Boyland [Boy01] described interdependence between effects and uniqueness of references. The *effects* of a method are the locations that the method reads and writes. As in the work described above and Leino's work, effects should be described abstractly, without mention of implementation details such as protected or private fields of objects. To enforce this abstraction, for example, to prevent representation exposure, one must prevent representation exposure, by confining subobjects used in the implementation of an abstract object. One way to accomplish this without copying is by having unique references. A *unique* reference is the only reference to an object in a program. The enforcement of uniqueness and optimization techniques such as alias burying, depend on knowing the effects of methods. For example, alias burying requires that when a unique variable is read, all aliases to it must be dead; hence this technique requires knowledge of what locations a method will read.

Since checking both effects and uniqueness annotations requires the semantics of the other, one can check them modularly by using specifications for each method. That is, when checking a method's annotations for effects and uniqueness, one assumes the annotations for all called methods.

The paper by Joachim van den Berg, Cees-Bart Breunesse, Bart Jacobs, and Erik Poll [vdBBJP01] discussed problems relating to the semantics and modular verification of class invariants. This presentation used JML to illustrate the issues. Invariants can cause significant problems for modular verification, because the class invariants of all objects are supposed to hold at all calls and returns from all public methods. Aliasing, if not controlled, can cause methods to break invariants of objects other than the method's receiver. Call-backs also cause difficulties, because a method that might call back may find the receiver object in an inconsistent state. This suggests that the official semantics in JML [LBR01] is unworkable for verification, because it requires that before a method is called on an object o, one must establish not only the calling class's invariant, but also the invariant for o's class. This touched off a lively discussion among the participants, with Boyland suggesting that read clauses could help [Boy01], and Poetzsch-Heffter suggesting that alias control could allow the invariant to be checked modularly [Mül01, MPHL01].

6 Coordination, Scripting and Specification

The last session consisted of three short presentations.

Claus Pahl [Pah01] developed a process calculus to capture the establishment and release of contracts. This calculus, which is a variant of the π-calculus, formalizes dynamic composition and evolution in Java systems.

Marco Carbone, Matteo Coccia , Gianluigi Ferrari and Sergio Maffeis [CCFM01] proposed ED_{-}, a coordination and scripting language. This language is based on Hennessy and Riely's Distributed π-calculus [HR99], which they implemented in Java, using the class loader, reflection, and sockets. They are now working on a type system for security.

Peep Küngas, Vahur Kotkas and Enn Tyugu in [KKT01] suggested "meta-interfaces" as a means to specify classes, and defined their composition. Meta-interfaces define which interface variables are computable from others, and under what conditions. The techniques used are similar to those used in constraint logic programming.

7 Conclusions

The interest in the workshop, and the lively discussions at the workshop itself show that many researchers are applying their techniques to either the Java language or to programs written in Java.

Java provides interesting language features, which need to be explored further. Their precise description is both scientifically interesting, and practically relevant.

Although Java is a rather complex language, having a common programming language makes it significantly easier to compare and discuss different approaches and techniques, and stimulates cooperations. This synergy was evident at the workshop, which helped make it a highly successful event that was appreciated by the participants.

The interests of the participants were very wide: source language semantics, source language extensions, bytecode verification, specification languages, reasoning about program correctness, security, applets, Java card, effects, refinement, coordination. Nevertheless, the common language provided enough common ground to allow for many interactions. In future workshops, it has been suggested that papers could be distributed in advance, and the program committee could assign to smaller working groups the task of reading these papers and addressing specific questions.

Several of the participants have come to that workshop for the third time, and expressed the intention of coming again.

List of Participants

Last Name	**First Name**	**Email**
Ancona	Davide	davide@disi.unige.it
Boury	Pierre	Pierre.Boury@dyade.fr
Boyland	John	boyland@cs.uwm.edu
Bracha	Gilad	gilad.bracha@sun.com
Breunesse	Cees-Bart	ceesb@cs.kun.nl
Clarke	David	dave@cs.uu.nl
Coglio	Alessandro	coglio@kestrel.edu
David	Alexandre	adavid@docs.uu.se
Drossopoulou	Sophia	sd@doc.ic.ac.uk
Eisenbach	Susan	sue@doc.ic.ac.uk
El Kadhi	Nabil	nelkadhi@club-internet.fr
Hamie	Ali	a.a.hamie@brighton.ac.uk
Huisman	Marieke	Marieke.Huisman@sophia.inria.fr
Huizing	Kees	keesh@win.tue.nl
Josko	Bernhard	josko@offis.de
Kotkas	Vahur	vahur@cs.ioc.ee
Kuiper	Ruurd	wsinruur@win.tue.nl
Kungas	Peep	peep@cs.ioc.ee
Lea	Doug	dl@cs.oswego.edu
Leavens	Gary	leavens@cs.iastate.edu
Maffeis	Sergio	maffeis@di.unipi.it
Markova	Gergana	gvm@cs.purdue.edu
Mughal	Khalid	khalid@ii.uib.no
Naumann	David	naumann@cs.stevens-tech.edu
Pahl	Claus	cpahl@compapp.dcu.ie
Poetzsch-Heffter	Arnd	poetzsch@fernuni-hagen.de
Poll	Erik	erikpoll@cs.kun.nl
Pollet	Isabelle	ipo@info.fundp.ac.be
Rensink	Arend	rensink@cs.utwente.nl
Retert	William	williamr@pabst.cs.uwm.edu
Skoglund	Mats	matte@dsv.su.se
Teschke	Thorsten	thorsten.teschke@offis.de
Viroli	Mirko	mviroli@deis.unibo.it
Vu Le	Hanh	Hanh.Vu_Le@irisa.fr
Wrigstad	Tobias	tobias@dsv.su.se
Zucca	Elena	zucca@disi.unige.it

References

[ALZ01] D. Ancona, G. Lagorio, and E. Zucca. Java separate type checking is not safe. Available in [DEL$^+$01], 2001.

[BE01] P. Boury and N. Elkhadi. Static analysis of Java cryptographic applets. Available in [DEL$^+$01], 2001.

[BGH01] G. Barthe, D. Gurov, and M. Huisman. Compositional specification and verification of control flow based security properties of multi-application programs. Available in [DEL$^+$01], 2001.

[Boy01] J. Boyland. The interdependence of effects and uniqueness. Available in [DEL$^+$01], 2001.

[CCFM01] M. Carbone, M. Coccia, G. Ferrari, and S. Maffeis. Process algebra-guided design of Java mobile network applications. Available in [DEL$^+$01], 2001.

[CN01] A. Cavalcanti and D. Naumann. Class refinement for sequential Java. Available in [DEL$^+$01], 2001.

[Cog01a] A. Coglio. Improving the official specification of Java bytecode verification. Available in [DEL$^+$01], 2001.

[Cog01b] A. Coglio. Java bytecode verification: A complete formalization. Technical report, Kestrel Institute, Palo Alto, 2001. Forthcoming at `www.kestrel.edu/java`

[DEJ$^+$00a] S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors. *Formal Techniques for Java Programs.* Technical Report 269, Fernuniversität Hagen, 2000. Available from `www.informatik.fernuni-hagen.de/pi5/publications.html`.

[DEJ$^+$00b] Sophia Drossopoulou, Susan Eisenbach, Bart Jacobs, Gary T. Leavens, Peter Müller, and Arnd Poetzsch-Heffter. Formal techniques for Java programs. In Jacques Malenfant, Sabine Moisan, and Ana Moreira, editors, *Object-Oriented Technology. ECOOP 2000 Workshop Reader*, volume 1964 of *Lecture Notes in Computer Science*, pages 41–54. Springer-Verlag, 2000.

[DEL$^+$01] S. Drossopoulou, S. Eisenbach, G. T. Leavens, P. Müller, A. Poetzsch-Heffter, and E. Poll. Formal techniques for Java programs. Available from `http://www.informatik.fernuni-hagen.de/import/pi5/workshops/ecoop2001_papers.html`, 2001.

[Eis98] S. Eisenbach. Formal underpinnings of Java. Workshop report, 1998. Available from `www-dse.doc.ic.ac.uk/~sue/oopsla/cfp.html`.

[HK01] K. Huizing and R. Kuiper. Reinforcing fragile base classes. Available in [DEL$^+$01], 2001.

[HR99] M. Hennessy and J. Riely. Type-safe execution of mobile agents in anonymous networks. In Jan Vitek and Thomas Jensen, editors, *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, volume 1603 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[IPW99] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java, a minimal core calculus for Java and GJ. *OOPSLA '99 Conference Proceedings*, pages 132–146, October 1999.

[JLMPH99] B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter. Formal techniques for Java programs. In A. Moreira and D. Demeyer, editors, *Object-Oriented Technology. ECOOP'99 Workshop Reader*, volume 1743 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[KKT01] P. Küngas, V. Kotkas, and Enn Tyugu. Introducing meta-interfaces into Java. Available in [DEL$^+$01], 2001.

[LBR01] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06p, Iowa State University, Department of Computer Science, August 2001. See `www.cs.iastate.edu/~leavens/JML.html`.

[Lei95] K. Rustan M. Leino. *Toward Reliable Modular Programs.* PhD thesis, California Institute of Technology, 1995. Available as Technical Report Caltech-CS-TR-95-03.

[Mor90] Carroll Morgan. *Programming from Specifications.* Prentice Hall International, Hempstead, UK, 1990.

[MPH00] P. Müller and A. Poetzsch-Heffter. A type system for controlling representation exposure in Java. Published in [DEJ$^+$00a], 2000.

[MPH01] Peter Müller and Arnd Poetzsch-Heffter. A type system for checking applet isolation in Java Card. Available in [DEL$^+$01], 2001.

[MPHL01] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular specification of frame properties in jml. Available in [DEL$^+$01], 2001.

[MS98] Leonid Mikhajlov and Emil Sekerinski. A study of the fragile base class problem. In Eric Jul, editor, *ECOOP '98 — Object-Oriented Programming, 12th European Conference , Brussels, Proceedings*, volume 1445 of *Lecture Notes in Computer Science*, pages 355–382. Springer-Verlag, July 1998.

[Mül01] P. Müller. *Modular Specification and Verification of Object-Oriented Programs.* PhD thesis, FernUniversität Hagen, 2001.

[Pah01] C. Pahl. Formalising dynamic composition and evolution in Java systems. Available in [DEL$^+$01], 2001.

[SW01] M. Skoglund and T. Wrigstad. A mode system for read-only references in Java. Available in [DEL$^+$01], 2001.

[vdBBJP01] J. van den Berg, C.-B. Breunesse, B. Jacobs, and E. Poll. On the role of invariants in reasoning about object-oriented languages. Available in [DEL$^+$01], 2001.

[Vir01] M. Viroli. From FGJ to Java according to LM translator. Available in [DEL$^+$01], 2001.