Formalising the Safety of Java, the Java Virtual Machine and Java Card

Pieter H. Hartel
Dept. of Computer Science, Univ. of Twente,
Email: pieter@cs.utwente.nl
and
Luc Moreau
Dept. of Electronics and Computer Science, Univ. of Southampton,
Email: L.Moreau@ecs.soton.ac.uk

We review the existing literature on Java safety, emphasizing formal approaches, and the impact of Java safety on small footprint devices such as smart cards. The conclusion is that while a lot of good work has been done, a more concerted effort is needed to build a coherent set of machine readable formal models of the whole of Java and its implementation. This is a formidable task but we believe it is essential to building trust in Java safety, and thence to achieve ITSEC level 6 or Common Criteria level 7 certification for Java programs.

Categories and Subject Descriptors: C.3 [**Computer Systems Organisation**]: Special-purpose and application-based systems—*Smart cards*; D.2.4 [**Software**]: Software/Program Verification; D.3.1 [**Software**]: Formal Definitions and Theory—*Semantics*

General Terms: Programming,Verification

Additional Key Words and Phrases: Common Criteria

1. INTRODUCTION

To illustrate what we mean by safety and security consider gaining access to a building. The first concern would be our safety. For example, one of the hinges of the door might be broken, so that a person entering the building could get hurt opening the door. By *safety*, therefore we mean that nothing bad will happen. Our second concern would be the security of the building. This means that even if the hinges are in good condition, to make the building secure, a lock on the door is required to control access to the building. By *security* therefore, we mean that there is access control to resources. Supposing that access to the building is safe

This work was funded in part by Technology Group 10 of the MOD Corporate Research Programme.

and secure, the third concern would be that we can enter the building whenever we wish to. By *liveness* we mean that eventually something good will happen. Finally how can we trust that those entering the building are indeed entitled to do so? This can be achieved for example by putting a surveillance camera near the door. Therefore, to trust our other arrangements we require the ability to *audit* safety and security. One might ask at this stage: What does all this have to do with Java?

Java is designed to support, distributed and in particular Internet applications. Java programs may travel the Internet, and may be executed on unsuspecting user's computers. Users would like to be sure that valuable resources are adequately managed and protected. Users may expect better safety from Java programs than from any other kind of mobile programs, because Java is a type-safe and memory-safe programming language.

Memory safety means that Java programs cannot forge pointers, or overrun arrays. Java offers references to objects which cannot be manufactured by the user but only by the system. A reference is either nil or it points at a valid object. Dangling pointers as in C cannot occur in Java. Users are not responsible for de-allocating objects, because Java supports automatic garbage collection.

Type safety means that the Java compiler is able to ensure that methods and operators will only be applied to operands of the correct type. Java is a strongly typed language, like Pascal and Ada, and unlike C and C++.

Java is unique in that both the compiler and the runtime environment enforce the same safety conditions. This means that tampering with compiled code will be detected if endangering type and memory safety. This property makes Java a good language for writing distributed code, for Web browsers but also for smart cards. The question then remains: to what extent can we trust the safety of Java? The answer is in auditing the safety arrangements, particularly by studying the formal semantics of Java.

While type safety and memory safety are necessary, they are not sufficient to make Java programs secure [Fritzinger and Mueller 1996; Cohen 1998]. For a Java program to be secure we make two main requirements [NCSC 1985]. The first is that Java programs can be restricted to accessing only certain resources, as defined by an appropriate security policy [Gong 1998]. While some resources are easy to control, like use of the runtime stack, or access to certain GUI events, other resources are harder to control, like execution time [Crary and Weirich 2000]. The second requirement is that secure systems can be audited. No system can be trusted to be secure unless it is possible to audit the behaviour of the system. While current Java systems have made good progress towards satisfying our first requirement, we do not know of any work done on auditing security. A more detailed introduction to the relevant terminology and concepts of safety and security in programming languages may be found in Volpano and Smith [1998]. We should also like to point out that the boundary between safety and security is sometimes a little vague [Thorn 1997]. In this paper we only mention security briefly, the main focus is on Java safety.

By itself, Java offers limited safety. For example, programming a credit card application in Java would not guarantee that the credit limit could never be exceeded. Program verification techniques are needed to reason about such high-level safety properties. Therefore we survey the application of such techniques to Java.

In the remaining sections of the paper we survey all aspects of formalising the safety of Java, its implementation based on the Java Virtual Machine (JVM), and its use in the verification of programs.

Throughout the paper we have tried to focus is on the novel aspects components of Java and its implementation. We have made an effort not to lose sight of the fact that those novel components must interact harmoniously with the other, tried and tested components.

In Section 2 we present an overview of the architecture of Java. Section 3 reviews a number of methods commonly followed to reason formally about the safety of programs and programming languages. The formalisation efforts of Java and its implementation are surveyed in Sections 4 (on Java), 5 (on the JVM) and 6 (on the compiler). Section 7 investigates the application of formal methods tools and techniques to the verification of Java programs. Section 8 presents a detailed case study of the application of Java to smart cards. The last section concludes the paper.

The paper contains three tables which provide an index into the relevant literature. These are Table 1 (on Java), Table 2 (on the JVM), and Table 3 (on the compiler).

2. THE JAVA ARCHITECTURE

According to Sun's Reference implementation, the Java programming language is compiled by the Java compiler into byte codes that can be interpreted by the Java Virtual Machine (JVM). Byte codes are stored in class files. The JVM is able to load class files and to execute byte codes by a process of interpretation. Byte codes are architecture neutral and can be run by a JVM on any architecture to which a JVM has been ported. Recent JVM implementations support Just-In-Time (JIT) compilers able to compile loaded byte codes into native assembly code, directly executable by processors.

In the rest of this section, we review the various language features found in the core of Java, and then we summarise some novel aspects of the Java architecture such as byte code verification, dynamic class loading and stack inspection.

2.1 Language Features

The Java and JVM languages have a number of interesting features. Some apply only to Java, some to the JVM and some to both. The most important aspects are the following. An imperative core consists of basic data, expressions and statements. The object-oriented nature of Java is based on objects, classes, interfaces and arrays. Java is a strongly typed language and uses a type system both at the Java language and JVM levels. Additionally, Java supports exceptions and multi-threading. Finally, Java is a garbage collected language.

Most researchers in the field model parts of Java's imperative core, and many also deal with object orientation. We will say little more about this core, as it is rather well understood. Instead, we will concentrate on the novel aspects of Java in what follows.

We have not been able to find any work on modelling garbage collection in the context of studying either Java or the JVM. This is a problem because garbage collection is not transparent since deallocating an object triggers its finalizer method.

2.2 Dynamic Class Loader

As opposed to other programming languages such as C or C++, byte code files are not linked into an executable. Instead, the JVM interpreter relies on a loader able to load classes dynamically. *Loading refers to the process of finding the binary form of a class or interface type with a particular name, perhaps by computing it on the fly, but more typically by retrieving a binary representation previously computed from source code by a compiler, and constructing from that binary form a* `Class` *object to represent the class or interface* [Gosling et al. 2000, Ch 12.2].

Once some byte code has been loaded, the class loader makes sure that code is properly linked with the rest of the runtime. *Linking is the process of taking a binary form of a class or interface type and combining it into the runtime state of the JVM, so that it can be executed* [Gosling et al. 2000, Ch 12.3].

Java is not the only language to support dynamic loading: Common Lisp also has such a facility, and most modern operating systems also support dynamic loading of libraries. In the presence of dynamically loaded code, a number of checks must be performed to ensure that the right code with the right signatures has been loaded. Such checks are performed by the byte code verifier, which we describe in Section 2.3. Java differs from other programming languages because it performs such checks at link-time as opposed to execution time; such an optimisation guarantees that checks are performed only once. Additionally, the JVM loads code in a lazy manner, when code is required in memory for execution to be able proceed. Consequently, at runtime, we can observe the interleaving of different activities such as code execution, code loading and linking; code linking is itself composed of three distinct activities: byte code verification, preparation and resolution of symbolic references [Gosling et al. 2000, Ch 12.3].

The role of the class loader in Java is complex: besides loading new classes, it also controls name spaces so that newly loaded applets [Fritzinger and Mueller 1996] or applications [Liang and Bracha 1998] may only see the resources they have the right to see.

Finally, dynamic loaders can be programmed and customised according to the applications' needs. For instance, byte code may be loaded from various locations or following different strategies, explicitly programmable by the user.

2.3 Byte Code Verification

The task of the byte code verifier is to check JVM code for type consistency and some other properties. The main checks performed by the JVM byte code verifier are that:

—Stack frames do not under- or overflow. (Stacks may still overflow because of lack of space for the next frame).

—Every byte code is valid.

—Jumps lead to legal instructions (and not into the middle of instructions).

—Method signatures (i.e. name and type of a method and its arguments) contain valid information.

—Operands to all instructions are of the correct type.

—Access control is obeyed (e.g., a private method is called only from within the class where the method is defined).

—Objects are initialised before use.

—Subroutines used to implement exceptions and `synchronized` statements are used in FIFO order.

2.4 Security Manager

The security manager is a runtime component that helps to implement Java security policies; for instance, access to the file system may be prohibited, connections can be made only to certain network addresses, or an applet or application can process only certain windowing events.

By default, Java relies on an *access controller* to determine whether a request for an operation should be granted or not. We explain here the notion of access controller, permissions and stack inspection used in this process. For the sake of illustration, let us consider operations on files; a permission for such an operation consists of a file name and the operation description (read, write or execute). Before reading a file, Java will request its security manager to determine whether the permission of reading the file is granted in the current context; by default, the access controller performs a *stack inspection* that determines the answer to the question.

Any stack frame results from the activation of a method, itself defined in a class, identified by its codebase (the urls that a class loader used to load the class) and its certificate. The codebase and certificate associated with a class are called its *code source*. Java associates each code source with a set of permissions. Starting from the most recent stack frame, the stack inspection algorithm determines if the code source of each stack frame has the permission to perform the operation. If the search encounters a frame whose code source does not have the permission to carry out the operation, then the search terminates: the access is forbidden and a security exception is thrown. Intuitively, the permissible operations of an execution context are defined by the *intersection* of all permission sets granted to the code sources it refers to. In practice, this picture is slightly more complex because Java has the equivalent of the "super user" mode, which in effect consists of interrupting the stack inspection algorithm to a frame that is marked as privileged; obviously defining privilege operations is itself controlled by the same security mechanism.

The role of the *policy manager* is to determine which permission is granted to a code source. Java provides a customisable policy manager, able to process security files, loadable from different locations. Each security file grants permissions to codebases and signers. The actual permissions that are granted to a given code source are defined by the *union* of all permissions given to its codebase and signers. Most recent implementations of the JVM offer stack inspection to allow different 'principals' access to different resources.

2.5 Discussion

Link-time byte code verification, lazy loading, name spaces, and user-programmability make dynamic class loading a novel element of the Java architecture. Consequently, the class loader and the byte code verifier are critical elements as far as safety is concerned. Additionally, the complementary role of the security manager at runtime constitutes the third pillar of what is usually called the *Java sandbox* [McGraw and Felten 1999].

Dynamic class loading *is not* a form of optimisation offered by Java, but it is an essential element of its philosophy. It allows the *distributed* development of programs, where programmers reuse libraries offered by others, without having to recompile them. It also allows new paradigms such as mobile agents, according to which the decision of where to execute code is made dynamically by programs.

However, we would argue that performing all checks at link-time is an optimisation, which requires the code to be checked only once. Optimisation are a common cause of errors [Dean et al. 1996], in class loaders, byte code verifiers, and in particular in the complex interplay between the class loader, the byte code verifier and the runtime system. Each single error is a safety loophole. Worse yet, each error may give rise to a bug fix, and system administrators will soon grow tired of installing yet another bug fix [Fong and Cameron 1998].

Unfortunately, we have not found any statistics for the performance benefits offered by link-time byte code verification, though it is hard to imagine that such statistics would not exist. Compile time techniques that remove unnecessary dynamic type checks were shown to improve the overall execution performance of Scheme programs by up to 10-20% [Serrano 1995; Wright 1994]. This figure is an approximate indication of what may be gained by link-time byte code verification.

To cultivate trust in the safety of Java it is necessary to study the three novel aspects of Java in detail. However, studying each in isolation is not sufficient as there are interactions with te rest of the system, including the imperative core of Java, its object system etc. Since the latter are in some sense well understood, it should be possible to adapt existing theories in the study of the interactions of all Java components.

3. METHODOLOGY

The ITSEC [1993] standard and the more recent Common Criteria [NIST 1999] for evaluating the security of IT products stipulate that for the highest levels of evaluation formal specifications must be provided of the system to be evaluated. The process of developing appropriate specifications can be broken down into the following steps:

—Construct clear and concise formal specifications of the relevant components.

—Validate the specifications by executing them, and by stating and proving relevant properties of the components. Examples include type soundness (i.e. a program that is well typed will not go wrong with a typing error at runtime), and compiler correctness (i.e. compiling a Java program to a JVM program should preserve the meaning of the program).

—Refine the specifications into implementations, or alternatively implement the specification by ad-hoc methods with an a-posteriori correctness proof.

—Create all specifications in machine-readable form, so that they can be used as input to theorem provers, model checkers, and other tools [Syme 1999].

Not all practitioners in the field work with machine readable specifications, and even those that do work with a plethora of different and incompatible formats, tools and notations.

3.1 Formalising Java

To establish the meaning of programs it is neccessary to establish the meaning of the programming language. Therefore we advocate the formal specification of Java consisting of:

—The semantics of Java.

—The semantics of the JVM language.

—The Java to JVM compiler.

—The runtime support. A specification of the runtime suppport is needed because for example starting and stopping threads is effectuated via the Java API and not via JVM instructions.

Regardless of Java's claims of being a small and simple language, which by comparison to C++ it is, Java is too complex and too large to make it easy for a complete formal specification to be built. It also contains some novel combinations of language features that have not been studied before. The principal difficulties are:

—Many different features need to be modelled, such as multi-threading, dynamic class loading, and garbage collection.

—Careful consideration has to be given to the interaction of these features, in particular when relatively novel features are concerned, such as byte code verification.

—The official Sun references [Gosling et al. 1996; Lindholm and Yellin 1996] are sometimes ambiguous, inconsistent and incomplete. See for example Bertelsen [2000], who provides a long list of ambiguities in the JVM specification.

—The reference implementation is complex (the Sun JDK), and not always consistent with the documentation.

Attracted by the potential benefits, and challenged by the difficulties, many authors have formalised aspects of Java, and/or its implementation. At the time of writing we counted more than 50 teams of researchers from all over the world. Many of those have specified the semantics of subsets of Java. Others have worked on the semantics of subsets of the JVM language. Some authors have worked on both, often in an attempt to relate the two, with the ultimate goal of proving the specification of a Java compiler correct. To our knowledge, no single attempt has been made at specifying full Java, the full JVM, or the full compiler. As far as we have been able to establish, only two groups have modelled small parts of the Java API: Coscia and Reggio [1998a] and the LOOP team [Poll et al. 2000]. (See Section 7.3).

The vast majority of the studies that we have found make various assumptions to abstract away detail, thus making the specifications more manageable. Popular assumptions include:

—There is unlimited memory.

—Individual storage locations can hold all primitive data types (i.e. `byte` as well as `double`).

—Individual JVM program locations can hold all byte code instructions.

While such abstractions help to reduce clutter in the specifications, they also make it impossible to model certain safety problems, such as jumping in the middle of an instruction. It is an art to model systems precisely at the right level of abstraction, with just enough detail to be able to discuss the features of interest.

3.2 Styles of Semantics

The styles of semantics used are Abstract State Machine Semantics (ASM), Axiomatic Semantics (AS), Context Rewriting semantics (CR), Continuation or monad Semantics (CS), Denotational Semantics (DS), Natural Semantics (NS), Operational Semantics (OS), Structural Operational Semantics (SOS), or a semantic embedding in a higher odrder logic (HOL).

We refer the reader to Nielson and Nielson [1991] for an introduction into programming language semantics, and to Nipkow *et al.* [2000] for a brief introduction into the notion of embedding the semantics of a programming language into the logic of a theorem prover.

4. THE JAVA LANGUAGE

This section reviews the literature on formal aspects of the Java language. We begin discussing the semantics of the Java core, showing that the majority of work in this area is concerned with proving that the type system is sound. The following subsections discuss the novel features of Java in detail: dynamic class loading, the memory model with multi-threading and stack inspection. We conclude the section with a general discussion.

We identify the methodological approaches and the Java subsets being studied. The reason is that some specification methods, and in particular the accompanying support tools, are perhaps more appropriate for the task in hand than others. We are also keen to identify methods and tools that are able to cope with the largest amount of complexity in the Java language, with the most features taken into account.

Table 1 summarises our findings on Java, showing the reference to the work, whether the work is particularly relevant to small footprint devices, the purpose of the activity, a reference to work on which the current work is based, the tools used, a characterisation of the subset studied, an indication of the style of semantics used, and whether any proofs have been reported.

4.1 Core Semantics

We begin our survey with papers that discuss a dynamic semantics of Java. Alves-Foss and Lam [1999] present a denotational semantics of a Java subset (excluding multi-threading and garbage collection, but including class loading). The specification gives considerable detail on the various basic data types in Java. The semantics is not used for any particular purpose. An alternative denotational semantics is offered by Cenciarelli [1999] using a monadic style to handle multi-threading and exceptions; in Section 5, we shall see that such a monadic style has also been adopted by Yelland [1999] and Jones [1998] to model the semantics of byte codes.

Attali *et al.* [1998] discuss a reasonably complete executable, operational semantics of Java built using the Centaur system. The specification includes concurrency, but it omits exceptions, arrays, and packages. The aim of this work is to use the

Table 1. Java Language – For legend see Table 3

Reference	Small	Purpose	Base	Tool	CL	EH	MT	Semantics	Proof
[Ábrahám-Mumm and de Boer 2000]		prove Hoare logic sound			no	no	yes	OS	yes
[Alves-Foss and Lam 1999]		study semantics			yes	yes	no	DS	no
[Attali et al. 1998]		executable specification		Centaur	no	no	yes	NS+SOS	no
[Bertelsen and Anderson 1996]		study semantics			yes	no	no	DS	no
[Börger and Schulte 1999b]		study semantics			no	yes	yes	ASM	no
[Cenciarelli et al. 1999]		study semantics			no	yes	yes	SOS	yes
[Cenciarelli 1999]		study semantics			no	no	yes	DS	no
[Coscia and Reggio 1998b] & [Coscia and Reggio 1998a]		verification	[Drossopoulou and Eisenbach 1997]		no	yes	no	NS+SOS	yes
[Demartini et al. 1998]		verification		SPIN	yes	yes	yes	embed	yes
[Drossopoulou et al. 1999] & [Drossopoulou and Eisenbach 1997] & [Drossopoulou and Eisenbach 1999] & [Drossopoulou and Valkevych 2000]		prove type soundness			no	yes	no	CR	no
[Drossopoulou et al. 1998] & [Drossopoulou 2000]		study semantics	[Drossopoulou et al. 1999]		no	no	no	CR	yes
[Drossopoulou et al. 1999]		study semantics			yes	no	no	AS	yes
[Glesner and Zimmermann 1998]		study semantics			no	no	no	NS	no
[Gontmakher and Schuster 2000]		memory model	[Ahamad et al. 1993]		no	no	yes	AS	yes
[Havelund and Pressburger 2000]		verification		SPIN	yes	no	yes	embed	yes
[Igarashi and Pierce 2000]		prove type soundness			no	no	no	SOS	yes
[Igarishi et al. 1999]		calculus			no	no	no	SOS	yes
[Jacobs et al. 1998]		verification		PVS	yes	yes	no	embed	yes
[Jensen et al. 1999]		verification			no	no	no	SOS	no
[Kassab and Greenwald 1998]		study security			no	no	yes	HOL	no
[Leino et al. 1999]		verification	[Detlefs et al. 1998]	ESC/Java	yes	yes	yes	embed	yes
[Maessen et al. 2000]		memory model			no	no	yes	algebraic	no
[Manson and Pugh 2001]		memory model			no	no	yes	OS	no
[Nipkow and von Oheimb 1998]		prove type soundness		Isabelle/HOL	no	no	no	NS	yes
[Poetzsch-Heffter and Muller 1999]		prove Hoare logic sound		Isabelle/HOL	no	no	no	NS	yes
[Skalka and Smith 2000]		study stack inspection			no	no	no	OS	no
[Stärk et al. 2001]		study of semantics, verification	[Börger and Schulte 1999b]	AsmGofer	yes	yes	yes	ASM	yes
[Syme 1999]		prove type soundness	[Drossopoulou and Eisenbach 1999]	DECLARE	no	yes	no	SOS	yes
[van den Berg et al. 2000]		verification	[Jacobs et al. 1998]	PVS, Isabelle/HOL	yes	yes	no	embed	yes
[von Oheimb and Nipkow 1999]		prove type soundness	[Nipkow and von Oheimb 1998]	Isabelle/HOL	no	yes	no	NS	yes
[von Oheimb 2000]		soundness, completeness	[von Oheimb and Nipkow 1999]	Isabelle/HOL	no	yes	no	AS	yes
[Wallace 1997]		study semantics			no	yes	yes	ASM	no
[Wallach and Felten 1998]		study stack inspection	[Abadi et al. 1993]		no	no	no	modal logic	yes
[Wallach 1999]		study stack inspection			no	no	no	translation	yes

Centaur system to generate an appropriate programming environment from syntactic and semantic specifications of Java.

All remaining papers discussed in this section present a static as well as a dynamic semantics of Java, mostly with the general aim of proving type soundness.

4.1.1 *Type Soundness.* The Java language provides not only single inheritance but also interfaces, which allow objects to support common behaviour without sharing any implementation. Drossopoulou *et al.* [1999] modelled these features, by giving a static semantics and a small-step dynamic semantics of a fairly large subset of sequential Java. Drossopoulou *et al.* then state the soundness of a type system based on simple subtyping using subject reduction. In a separate paper, Drossopoulou and Eisenbach [1999] extend their subset to include exception handling. Results were proved by hand but neither paper gives proofs. Instead Syme [1999] encodes some of the models of Drossopoulou *et al.* in his DECLARE system, and gives proofs. The mere activity of encoding hand built specifications in a mechanised system is reported to have uncovered 40 errors made during the translation. More importantly, Syme has also found two non-trivial errors in the hand written proofs of Drossopoulou and Eisenbach.

Drossopoulou and Valkevych [2000] propose a semantics of a Java subset with both checked and unchecked exceptions. The semantics relies on Felleisen's [1992] context rewriting technique to model the dynamic behaviour of exceptions. As far as typing is concerned, exceptions are regarded as "effects" [Talpin and Jouvelot 1992]; two types are associated with each expression: a normal type describes the type of the program in the absence of exceptions (or more precisely, when all exceptions are caught), and an abnormal type describes the type of uncaught exceptions. Type soundness is established by subject reduction up to type widening.

Glesner and Zimmermann [1998] specify the type system for a small fragment of Java as an example of their work on many sorted logic. Their method allows a generic static semantics to be instantiated to a type system, or a static analysis.

Igarashi and Pierce [2000] give a static semantics of a small subset of Java with just inner classes and inheritance. They show that this semantics coincides with a semantics given by translation of Java with innerclass to Java with just top level classes, as stipulated by the official Sun documentation.

Nipkow and von Oheimb [1998] prove type soundness of their Java$_{\ell ight}$ subset, which is similar to the subset used by Drossopoulou *et al.* [1999]. However, the former use Isabelle/HOL to machine-check the proofs from the outset, giving a higher degree of confidence in the correctness of the specifications and the proofs. While the semantics are verified using a proof checker, von Oheimb and Nipkow [1999] were not able to validate the specifications due to the lack of support for generating executable semantics. One conclusion of their work is that theorem provers are too sensitive to the precise formulation of a specification, and that more support in the provers is needed to support the development of programming language semantics [von Oheimb and Nipkow 1999, Page 151]. To complement the operational semantics of Java$_{\ell ight}$, von Oheimb [2000] presents an axiomatic semantics, and proves the soundness and completeness of the latter with respect to the operational semantics.

4.1.2 *Calculi for Java.* Igarashi, Pierce and Wadler [1999] present Featherweight Java (FJ) as a calculus for Java, like the lambda-calculus is a calculus for languages such as ML and Haskell. Minimality of the calculus is a key feature of their design; only the following constructs are supported: recursive class definition, object creation, field access method invocation, method override, method recursion through this, subtyping and casting. As assignment is not part of the calculus, FJ looks like a "functional" version of Java. Their calculus is smaller than ClassicJava proposed by Flatt *et al.* [1999] as a formal foundation for mixins, a kind of functor over classes.

4.1.3 *Polymorphism.* While the previous papers are concerned with the core semantics of Java, some authors note that the lack of polymorphism in Java is not conducive to safety: without proper polymorphism, Java programmers must make explicit use of type casts, creating scope for errors. Consider for example the collection classes in Java 2. The class of the items being stored and manipulated by the collection classes is `Object`. So when storing an object of some meaningful type, say `MyObject`, one must remember to cast the raw object back into the user class `MyObject` when retrieving the information. Erroneous type casts will eventually cause unexpected runtime exceptions. Java extensions such as Pizza [Odersky and Wadler 1997] and Generic Java [Bracha et al. 1998] address these problems by providing *generic types* for Java, a mechanism by which classes and methods may be abstracted with respect to type; for instance, programmers may defined vectors of `MyObject`. Generic types are compiled into Java using a method called *type erasure* removing type parameters and automatically inserting the required type casts; Generic Java then guarantees that no cast inserted by the compiler will fail. Generic Java programs inter-work perfectly with legacy code, the compiler is even able to make legacy Java code available for use with genericity without the need to even recompile the legacy code. As an application of their FJ calculus, Igarashi *et al.* [1999] show how to compile Featherweight Generic Java (FGJ) in to FJ, where FGJ is defined as the calculus version of Generic Java. While a number of approaches with similar goals have been proposed, such as type parameterisation [Agesen et al. 1997], NexGen [Cartwright and Jr. 1998], PolyJ [Myers et al. 1997], it remains a challenge to design extensions and associated compilers, which do not require changes to the JVM and preserve backward and forward compatibility with existing libraries. Only the type erasure technique is currently able to do so, but it has an important limitation due to its lack of support for parameterisation over primitive types.

4.2 Dynamic Class Loader

This section covers dynamic class loading from the Java point of view. Section 5.3 revisits class loading from the JVM point of view.

According to the Java language specification, a change to a type is *binary compatible* with preexisting binaries if preexisting binaries that previously linked without error will continue to link without error (cf. [Gosling et al. 1996; Gosling et al. 2000, Ch. 13]). More specifically, a list of important binary compatible changes supported by Java is enumerated in the language specification. Drossopoulou, Eisenbach and Wragg [1998, 1999] observe that not only do binary compatible changes not require re-compilation of other units, but such re-compilations *may not be possible*. For the purpose of illustration, they provide the source code of a class and some change, which is binary compatible according to the Java specification; they show that the changed class still allows linking and execution of the code, but the source code of all the classes taken together is no longer type correct. Consequently, separate compilation as supported by Java is *not* equivalent to compilation of all units together, and therefore requires a specific study. Binary compatibility is a powerful concept because it is needed for distributed development of programs, but it is an immature language feature, which deserves a formal understanding.

Drossopoulou *et al.* [1998] study the problem of separate compilation and binary compatibility on the significant subset of Java described in [Drossopoulou et al. 1999], and by using the well-typedness judgment there defined. Notions of fragment concatenation and compilation are introduced, and the concept of link compatible change is defined so as to capture the guarantees given by binary compatibility. A set of properties of link compatibility are proved; in particular, link compatible changes are shown not to satisfy a diamond property, which explains why programmers cannot apply independent link compatible changes to the same fragment and expect the linking capabilities to be preserved [Drossopoulou et al. 1998]. Then, a notion of type preserving change is defined and proved to imply link compatibility. They show that all but one of the changes enumerated in the language specification are type preserving; interestingly, the change that is not type preserving was used in the example given by Drossopoulou *et al.* [1998, 1999].

In contrast to their first formalisation of binary compatibility, Drossopoulou *et al.* [1999] investigate an alternative approach based on axiomatic definitions of fragments, compilation and linking, which allow them to reflect on these issues at a more abstract level. The authors discuss a number of interpretations of the language specification [Gosling et al. 1996; Gosling et al. 2000]; in particular, they analyse the notion of locality, according to which properties established in an environment also hold in a larger one. Overall properties of link compatibility similar to [Drossopoulou et al. 1998] are established using the new axiomatic definitions.

In a subsequent paper, Drossopoulou [2000] considers the complete sequence of Java components in a single framework, namely evaluation, loading, verification, preparation and resolution. Her work differs from work on the JVM discussed in Section 5.3 because she considers a source language close to Java [Drossopoulou et al. 1999] and not the byte code language. Additionally, she does not consider each component in isolation, but instead proposes a single framework to understand their interplay. An informal description of her investigation may be summarised as follows. Verification does not ensure the presence of fields or methods, it only ensures that all methods in a verified class respect their signatures. Resolution checks for the presence of fields and methods of given signatures. The resolver and the verifier are mutually dependent: the verifier relies on resolution to pick some of the possible errors, and resolution is safe on code previously checked by the verifier. Verification alone does not guard against link-time errors but guarantees the integrity of the system. The integrity of the system is demonstrated by a subject reduction lemma, and it relies on the well-typedness of the expression and prepared code. This work does not consider multiple class loaders as opposed to [Qian et al. 2000], and does not integrate the notion of link compatibility [Drossopoulou et al.

1998].

4.3 Memory Model and Multi-threading

Java is the only widespread programming language that provides a memory model for parallel execution. The Java language specification allocates a whole chapter to the complex Java memory model. *In Java, each thread of control has its own private working memory, in which it keeps its own working copy of variables that it must use or assign. As the thread executes a program, it operates on these working copies. The main memory contains the master copy of every variable. There are rules about when a thread is permitted or required to transfer the contents of its working copy of a variable into the master copy or vice-versa [Gosling et al. 2000, Ch. 17.1].*

These rules are not straightforward to understand because they rely on a double indirection, for the transfer of values between a thread engine and its working memory, and for the transfer of values between its working memory and the main memory. Cenciarelli *et al.* [1999] propose a structural operational semantics (SOS) that is parametric in a notion of event space, used to formalise the constraints between memory actions. As an illustration of the parametric nature of their semantics, the authors express the set of constraints characterising prescient store actions. Such store actions, defined by the Java language specification [Gosling et al. 2000, Ch 17.8], use a set of relaxed constraints, allowing optimising Java compilers to perform some code rearrangement that preserve the semantics of properly synchronised programs. Cenciarelli *et al.* then establish that any properly synchronised program without prescient store actions is equivalent to itself in the presence of prescient store actions. (The equivalence used in the proof is a bisimulation.)

Gontmakher and Schuster [2000] investigate the Java memory model in isolation (independently of the language itself). They provide a non-operational, trace based characterisation of the memory model, which they compare with existing memory models [Ahamad et al. 1993]. Gontmakher and Schuster's investigation considers the programmer's and the implementor's points of view. For the programmer, understanding memory models may assist in the selection of algorithms and their porting to Java. The programmer can rely on Cache Consistency (Coherence) [Ahamad et al. 1993] and some weak variation of Causality for regular variables, on Sequential Consistency for volatile variables, and on Release Consistency when using the `synchronize` construct. On the other hand, the implementor has to provide an implementation of the JVM that supports all byte codes that comply with the Java specification; the memory behaviour is shown to be a combination variant of Cache Consistency and Causality.

Java designers intended that a relaxation of the Java memory model would be exploited by implementors of parallel or distributed Java systems. The Hyperion system [Antoniu et al. 2000] is a Java implementation on top of a specifically designed Distributed Shared Memory (DSM), which gives the programmer the illusion that a cluster of processors operates as a single JVM. An informal discussion on how Java Consistency is achieved can be found in the paper: a thread's object cache is flushed upon entry to a monitor and local modifications to cached objects are transmitted to the main memory when a thread exits a monitor. Additionally, Hyperion shared memory is able to bring copies of objects to the node using them, hereby offering better locality to the threads accessing these objects. Chen and Allan [1998] describe MultiJav an alternative cluster implementation of the JVM; they discuss their implementation of Sequential Consistency for synchronisation-protected shared objects and Release Consistency for volatile variables. cJVM is another implementation providing a single system image of a JVM while executing in a distributed fashion on the nodes of a cluster [Aridor et al. 1999; Aridor et al. 2000]. In cJVM, methods calls are performed on the node that holds the master copy; additionally, several caching techniques have been proposed to improve the performance. The papers above on relaxation of the Java memory model tend only to discuss implementation issues and do not present appropriate formalisms and theories, which would be required for a complete proof of correctness of these implementations.

Pugh [2000] wrote a critique of the Java memory model and he argues that it requires Coherence (Cache Consistency) [Ahamad et al. 1993], which informally means that the memory should be sequentially consistent on a per-location basis. He then shows that fairly standard compiler optimisations are not possible with the Java memory model, and moreover, many JVM implementations are not compliant with the Java specification (cf. Javasoft bug 4242244). Additionally, he shows that some assumptions made by Java programmers may no longer hold: for instance, `String`s may not be thread safe, and it may be difficult to implement efficiently their thread safety on shared memory multiprocessors with weak memory models. Alternative memory models, proposed by Manson and Pugh [2001] and Maessen *et al.* [2000], answer some of the problems raised by the Java memory model. Both proposals bear some similarities as they arise independently from discussions on the Java Memory Model mailing list. In both cases formal semantics are proposed; [Manson and Pugh 2001] contains an operational description of memory related operations, whereas [Maessen et al. 2000] enumerates algebraic rules specifying the order of these operations. The authors have not re-established that the behaviour of properly synchronised programs is preserved over their proposed memory model (as Cenciarelli *et al.* [1999] did for the Java memory model). Finally, they also discuss the validity of lock elimination transformations, which have been shown to remove approximately 50% of the useless locking [Aldrich et al. 1999; Bogda and Hölzle 1999; Choi et al. 1999; Whaley and Rinard 1999], but may break the semantics of programs over the new proposed memory models.

Coscia and Reggio [1998b, 1998a] present a structural operational semantics (SOS) of Java's multi-threading. Their aim is the development of a foundation for program verification. Java's rather loose coupling of threads, via the separated working and main memory model is identified as the source of difficulty in writing correct, concurrent Java programs. Coscia and Reggio characterise a subset of Java, which avoids some of the concurrency problems. Unfortunately, the subset cannot be statically characterised. This makes it less than straightforward for programmers to take heed of the advice offered. Coscia and Reggio include small parts of the Java API in their formalisation, namely the methods necessary to control threads, such as `start`, `stop` etc.

Ábrahám-Mumm and de Boer [2000] present an operational and an axiomatic semantics of a subset of Java. The purpose of their work is to provide a framewrok for proving properties of multi-threaded control flow. Their subset is an abstract version of the Java core with multi-threading. Soundness and completeness are proved in a separate technical report.

Kassab and Greenwald [1998] create a state based abstraction of Java threads and security policies to study the enhanced Java 2 security model. The main thrust of the paper is the complexity analysis of the thread abstraction, which is shown capable of coping with generalisations of the Java 2 security model. The authors are concerned however, that adding further flexibility to the Java security model will make it too difficult to implement correctly.

4.4 Stack Inspection

The stack inspection algorithm summarised in Section 2.4 allows implementors to control permissions in a fine grained manner. Stack inspection assumes an operational model with a stack that is not visible at the Java level. Even informal reasoning about stack inspection requires making the stack explicit at a level where the stack is not visible. Stack inspection is a low level concept that it is at odds with the abstractions provided in a high level language.

Wallach and Felten [1998] develop an abstract model of stack inspection in terms of a belief logic, known as ABPL (Abadi, Burrows, Lampson and Plotkin [1993]). Java's access control decisions are shown to correspond to proving statements in ABPL.

The stack inspection approach suffers from a number of problems. First, the stack inspection algorithm expects a specific stack layout, which, though standardised by the JVM, prevents optimisations that just-in-time compilers are expected to perform. Second, the result of this algorithm is potentially sensitive to optimisations such as tail recursion optimisations (in particular in the case of mutually tail-recursive methods belonging to different classes). To solve these problems, Wallach [1999] proposes a program transformation, *security-passing* style, which like continuation-passing style, adds an extra argument to each method and each method invocation, which is the current security context of the program. This has two advantages. First, it does not restrict compiler optimisations. Second, security-passing style allows the access controller to be written as regular Java code, so that it can be ported to JVMs that do not implement the stack inspection algorithm.

Skalka and Smith [2000] propose a lambda calculus formalising Java stack inspection. Lambda expressions are annotated by the name of the principal they "belong to". The application of a lambda expression extends a stack with an association between principals and a set of privileges. A `letpriv` expression allows adding privileges to the current principal. A `checkpriv` expression performs stack inspection in a similar way as the JDK does. Then the authors define *security stack safety* as the property according to which no well-typed program will ever have any stack inspection failure during runtime execution. The essence of their type system is the annotation of the arrow type with the set of privileges necessary to execute that function. They establish the safety and subject reduction of the typing system, with respect to their operational semantics. As expected with a type system, there are programs that are operationally safe but not typeable; a significant reason given by the authors is that the type system is monomorphic. This is a promising approach, which may improve efficiency, but it needs to be extended to the full language. Additionally, Java programs are able to test their privileges and act accordingly; the type system does not currently support type that depends on the dynamic security level of the execution context.

4.5 Discussion

Java is type safe and supports dynamic loading, binary compatibility, a memory model for parallel execution and security management. All these topics have been investigated by the research community, and research has helped to understand them. In some cases, research has helped to identify flaws in the specification of the language, and resulted in new solutions.

Many investigations typically focus on a subset of the language or on some specific aspect in isolation of other components. There is a need for unifying frameworks that help understand interactions between components. This is an ambitious task and its size requires mechanical tools able to handle formalisations.

Garbage collection is an integral part of Java and its API since programmers have the right to program finalizers [Hayes 1992] and have access to different kinds of weak references through the package `java.lang.ref`. Conferences such as ISMM and OOPSLA regularly include papers on garbage collection for Java, but these publications tend to focus on implementation specific issues. The notion of object reachability has been studied in a language-independent manner [Morrisett et al. 1995], but we have been unable to trace any work integrating such a type of formalisation with the Java semantics.

Other APIs can be regarded as language extensions: for instance, remote method invocation (RMI) adds distribution to Java, serialisation is a process that has to preserve some safety invariants across JVMs, and Java's spaces introduce the idea of coordination to Java. These will also have to be investigated by the research community.

Once Java safety has been understood, writing safe and secure applications will be another major challenge. Tools are needed to understand the behaviour of programs: for instance, understanding the visibility of data and the permissions required to access them, in the presence of packages, dynamic loading, and explicit security is a non-trivial task: automatic tools advising programmers would be useful. While calculi such as FJ are emerging as theoretical foundations of the Java language, we have not yet observed the type of equational reasoning that exists for functional languages: notions of observational equivalence or bisimulation are rarely mentioned (except by Abadi [1999] and Cenciarelli *et al.* [1999], respectively).

We have not found many papers focusing on Java for small footprint devices, such as PDAs, mobile phones and smart cards. Yet embedded systems are a particularly interesting application area for Java, as witnessed for example by the successful introduction of Java for smart cards (See section 8).

5. THE JVM LANGUAGE

The JVM language was specifically designed to compile Java programs and bears numerous similarities with Java: it is also object-oriented and supports packages, threads and dynamic loading. The JVM language differs from Java. For example the JVM has a notion of subroutine and it does not have built-in inner classes. Due to their differences, the Java language and the JVM language have different observable properties; such a phenomenon was described by Abadi [1999], who

Table 2. JVM Language– For Legend see Table 3

Reference	Small	Purpose	Base	Tool	CL	EH	MT	Semantics	Proof
[Bertelsen 1997]		study semantics			yes	yes	no	OS	no
[Bertelsen 2000]		study semantics	[Bertelsen 1997]		no	no	no	OS	no
[Bigliardi and Laneve 2000]		study semantics	[Stata and Abadi 1998]		no	no	yes	OS	yes
[Börger and Schulte 2000a]		study semantics			yes	no	no	ASM	no
[Coglio et al. 1998]		prove type soundness	[Goldberg 1998]	SpecWare	yes	no	no	embed	no
[Cohen 1997]		study semantics		ACL2	yes	no	no	embed	no
[Dean 1997]		study semantics		PVS	yes	no	no	embed	yes
[Denney and Jensen 2000]	yes	prove converter correct	[Bertelsen 1997]	Coq	no	no	no	OS	yes
[Fong and Cameron 1998]		study semantics			yes	no	no	HOL	no
[Freund 1998]		study semantics	[Freund and Mitchell 1998]		no	yes	no	OS	no
[Freund and Mitchell 1999a] & [Freund and Mitchell 1999b]		study semantics	[Stata and Abadi 1998]		no	yes	no	OS	yes
[Goldberg 1998]		prove type soundness		SpecWare	yes	no	no	embed	no
[Hagiya and Tozawa 1998]		study semantics	[Stata and Abadi 1998]		no	yes	no	OS	yes
[Tozawa and Hagiya 1999a] & [Tozawa and Hagiya 1999b]		study semantics	[Stata and Abadi 1998]		yes	no	no	OS	yes
[Hartel et al. 1999]	yes	study semantics		LETOS	yes	no	no	OS	no
[Jensen et al. 1998]		study semantics			yes	no	no	OS	no
[Jones 1998]		prove type soundness		Haskell	no	yes	no	CS	no
[Klein and Nipkow 2000]	yes	prove type soundness	[Rose and Rose 1998]	Isabelle	no	no	no	NS	yes
[Lanet and Requet 1998]	yes	prove converter correct	[Qian 1999a]	Atelier B	no	no	no	embed	yes
[Moore 1999]		verification	[Cohen 1997]	ACL2	no	no	no	embed	yes
[O'Callahan 1999]		study semantics	[Stata and Abadi 1998]		no	yes	no	CS	no
[Basin et al. 1999]	yes	prove type soundness	[Börger and Schulte 1998]	SMV	no	yes	no	tool	yes
[Pusch 1998]		prove type soundness	[Qian 1999a]	Isabelle/HOL	yes	no	no	OS	yes
[Pusch 1999]		prove type soundness	[Pusch 1998]	Isabelle/HOL	yes	no	no	OS	yes
[Qian 1999a]		prove type soundness			no	yes	no	OS	no
[Qian et al. 2000]		prove type soundness			yes	no	no	OS	yes
[Requet 2000]	yes	prove type soundness	[Stata and Abadi 1998]	Atelier B	no	yes	no	OS	yes
[Rose and Rose 1998]	yes	prove type soundness	[Rose 1998]		no	no	no	NS	yes
[Stata and Abadi 1998] & [Stata and Abadi 1999]		study semantics			no	yes	no	OS	yes
[Stephenson 1998]		study semantics			yes	yes	no	DS	no
[Wallach and Felten 1998]		study security			no	no	no	HOL	yes
[Yelland 1999]		prove type soundness		Haskell	yes	yes	no	CS	yes

Fig. 1. a Java statement and the equivalent JVM code showing how information is scattered by compilation into byte codes (compiled by Sun JDK 1.1.2 on SunOS 5.6).
Java statement:

```
Point p = new Point (1,0);
```

Compiled JVM code:

```
   0 new #1 <Class Point>
   3 dup
   4 iconst_1
   5 iconst_0
   6 invokespecial #5 <Method Point(int,int)>
```

observes that the compilation process of the Java language is not fully abstract. (See Section 6.1.)

A crucial role of the JVM is to ensure that byte codes do not corrupt its internal state and lead to undesired behaviour. Typically byte code programs are generated by compilers, but they may also be written by hand, or corrupted during network transmission. Therefore, the JVM uses a byte code verifier that performs a number of consistency checks before executing the code.

As in the previous section on Java, the current section on the JVM is structured to highlight the novel aspects of the JVM language: byte code verification and class loading. We begin with a brief mention of papers that merely present a semantics of a JVM subset. We conclude with a discussion of alternative representations of byte codes as a potential improvement to the JVM and a conclusion. Table 3 summarises the literature on the JVM.

5.1 Core Semantics

Bertelsen [1997, 2000], and Stephenson [1998] give an operational semantics of a subset of the JVM. A detailed specification of a subset of the JVM (excluding multi-threading and garbage collection, but including class loading) is given by Cohen [1997]. His specification is large but executable (using ACL2), which makes it relatively easy to validate. Cohen's semantics comprises explicit runtime checks to assure type-safe execution of byte codes instead of requiring a link-time byte code verification. The purpose of each of these works is purely to study semantics.

5.2 Byte Code Verification

Byte code verification is difficult for two reasons. First, byte codes offer scope for optimisation, which has the tendency to destroy information. Second, the information necessary to check certain properties is often spread across a section of byte code instructions, whereas the same information would be more readily available in the original Java sources. Consider for example the initialisation of Figure 1; we see that one Java expression corresponds to 5 separate JVM instructions, which could even be mixed with other instructions.

The byte code verifier has to recreate the information that was once readily available in the Java source. This re-discovery of information complicates the byte code verifier, and makes it difficult to specify in a clear and concise fashion what byte code verification is.

Stata and Abadi [1998, 1999] were the first to propose the use of typing rules to describe the byte code verification process. Such rules are more precise than a natural language description of the verifier [Lindholm and Yellin 1996] and easier to understand and reason about than Sun's reference implementation. Stata and Abadi's initial publication [1998] considered a small but representative sets of byte codes (9 instructions out of over 200) and addressed the problem of subroutines, which will be explained in Section 5.2.1. Their proposal was followed and extended by several authors. Freund and Mitchell [1998] used the same typing framework to investigate the problem of object initialisation. Then, Freund and Mitchel [1999b] integrated in a single framework the handling of subroutines and object initialisation. The same authors again [1999a] added objects, classes, interfaces, arrays, exceptions and double types. Using the same approach, Hagiya and Tozawa [1999a, 1999b] formalised the instruction `invokevirtual` and its relationship with dynamic loading, which we study in detail in Section 5.3. The approach was then extended by Bigliardi and Laneve [2000] to check that monitors entering and leaving instructions were properly balanced.

In its simplest version, Stata and Abadi's framework is based on a form of dataflow analysis and uses type judgements to express the existence of valid instructions at the location pointed by the program counter, the type of registers and stack content and the type compatibility of the current instruction with its arguments. Successive papers extend these judgement with, e.g., information related to subroutine calls or the initialisation of objects. Soundness is established by showing that well-typed programs only get stuck when a `halt` instruction is reached, which means that programs do not attempt to perform any illegal operation.

Alternative approaches to formalise the byte code verification process have been proposed. Goldberg [1998], and Qian [1999a] aim to prove type soundness of the JVM (i.e. the correctness of the byte code verifier) for a relatively small subset of the JVM, excluding multi-threading, garbage collection, class loading etc. The main tool is a data flow analysis, which establishes constraints at all program points. Solving the generated constraints then establishes the desired properties of the program, such as the byte code verification conditions listed in Section 2.3. In a joint paper Coglio, Goldberg and Qian [1998] extend their previous work towards a provably correct implementation of their specifications, using SpecWare [Srinivas and Jullig 1995]. This tool supports refinement of specifications towards Lisp and C++. As far as we know, this is the only proposal to derive a provably correct implementation of (part of) a JVM implementation. Qian [1999b] revisits byte code verification, using a chaotic fixed point iteration technique to compute a least type for a byte code program. If such a type exists, then the byte code is well typed.

Pusch [1998, 1999] offers an alternative to Qian's work [1999a], by stating and proving type soundness at the JVM level using Isabelle/HOL to mechanise the proofs.

Two independent approaches regard byte code verification as equivalent to typing a Haskell program. Jones [1998] models individual byte code instructions, and their compositions, as appropriately typed functions in the functional language Haskell. This allows him to describe the process of byte code verification as a type inference that guarantees that the execution of verified programs will not "go wrong". He presents his type system as a general framework for data-flow analysis. In a first instance, byte codes are defined as functions mapping a frame and a stack onto a frame and a stack. He then extends the model to support multi-word values such as long that are stored in two ints. The limitations of the initial framework become clear when I/O, exceptions and mutable state should be formalised: Jones redefines his byte codes so that they have a reference to a monad. Finally, Jones sketches how general effects on objects in the heap may be formalised. A number of questions need to be addressed. In the framework, fields of frames are named and codes refer to names. Supporting instructions that refer to field indexes as in the real JVM is not obvious because all fields would be denoted by "ints". Finally, Jones has only investigated a few instructions. It is not straightforward that this framework will support the whole JVM instruction set.

Yelland [1999] presents a continuation semantics of a small subset of the JVM (the μVM) using the functional programming language Haskell. Using Wadler's monads [1990] and Rémy's encoding of sub typing in Haskell's type system [1989] (Haskell does not support sub typing), Yelland is able to use the Haskell type checker as a 'byte code verifier'. The encoding is rather inefficient, as it requires an n-tuple as a representation of a class, when there are in total n-classes defined. However using a pure functional language as the notational vehicle does give a compositional framework, allowing specifications of further byte codes to be added as a conservative extension.

5.2.1 *Subroutines.* At the JVM level, exception handlers are implemented as subroutines. This technique saves code space, as exceptions raised from different places may transfer control to the same exception handler code. However, this optimisation is problematic both for the byte code verifier and for the garbage collector.

The first problem with subroutines used for exception handling is that a local variable that is not actually used in a subroutine may be polymorphic. To illustrate this, consider the example of Figure 2 originating from Stata and Abadi's paper [1998]; their byte code is slightly different, probably because of our using a more recent Java compiler. The method `bar` uses five locations to store local variables:

location	type	purpose
0	ref	to refer to the current object `this`
1	`int`	to represent the `int` argument of the method `bar`
2	`int`	to save the `int` type result of `this.foo()`
3	ref	to save the exception reference
4	address	to save the address to which the subroutine at address 27 should return.

Each location is consistently used with only one type. However, it is possible to optimise the code to use fewer local variables. For example we could reuse location 2 to store the return address at statement 21, and to reload the return address again from location 2 at statement 25, as indicated in comments in the JVM code

Fig. 2. A Java method and the equivalent JVM code showing how subroutines are used to space code space for exceptions (compiled by Sun JDK 1.1.2 on SunOS 5.6).
Java Method:

```
int bar(int i) {
  try {
    if(i==3) return this.foo();
  } finally {
    this.ladida();
  }
  return i;
}
```

Compiled JVM code:

```
Method int bar(int)
   0 iload_1
   1 iconst_3
   2 if_icmpne 15
   5 aload_0
   6 invokevirtual #4 <Method int foo()>
   9 istore_2
  10 jsr 27
  13 iload_2
  14 ireturn
  15 jsr 27
  18 goto 35
```

Exception handler

```
  21 astore_3
  22 jsr 27
  25 aload_3
  26 athrow
```

The `finally` block

```
  27 astore 4
  29 aload_0
  30 invokevirtual #5 <Method void ladida()>
  33 ret 4
```

Return statement at the end of the method `bar`.

```
  35 iload_1
  36 ireturn
```

of Figure 2. This would remove the need for location 3. While verifying the byte code of the subroutine at address 27, the verifier must now take into account that location 2 may either have type `int` (when the call originates from 10), or that location 2 may be a reference (when the call originates from 22). This optimisation is safe because location 2 is not actually used in the subroutine. We say that location 2 is a polymorphic variable of the subroutine.

The second potential problem with JVM subroutines is that the `jsr` (Jump subroutine) and `ret` (return from subroutine) instruction pair does not use a stack to store return addresses but instead stores a return address in a local variable. Stata and Abadi take a special care to ensure that the `ret` instruction returns control to the instruction following the last executed `jsr` instruction. The JVM however supports returns of multiple subroutines with a single `ret` instruction, and this facility is formalised by Freund and Mitchel [1999b].

Concerned about the byte code verification complexity, Freund [1998] shows that expanding out subroutines increases code size by about 0.02% for the whole JDK 1.1.5. This suggests that the benefits of having polymorphic subroutines may not compensate for the increased complexity.

Hagiya and Tozawa [1998] follow a different approach to combat the complexity in the proofs by proposing a new type system for byte codes. They show that the complexity of the proof of type soundness is significantly reduced.

O'Callahan [1999] provides yet another solution to the problem by using a polymorphic type system for subroutines. This gives a strictly more powerful type system than that of the other proposals referred to above. Unfortunately, using the polymorphism further increases the complexity of the byte code verifier.

Subroutine polymorphism is also a problem for type-precise garbage collection [Agesen et al. 1998], because the type of local variables that are use in a polymorphic fashion depends on the call site of the subroutine. Therefore, instead of the types of all local variables being dependent only on the current state of the execution, now the type depends on the history of the execution.

The Java language provides two means to express synchronizations, through synchronized method and synchronized statements. By their syntactic nature these constructs are balanced, with clearly specified beginning and end. The JVM language implements synchronized blocks with the instructions `monitorenter` and `monitorexit` and is required to exit a monitor at the end of the method invocation it was entered, whether completion is normal or abrupt [Lindholm and Yellin 1996]. Bigliardi and Laneve [2000] extend the Stata and Abadi [1998] and Freund and Mitchell [1998] typing methods to those byte codes, and prove that program typing implies the proper release of monitors.

5.3 Dynamic Class Loader

This section covers dynamic class loading from the JVM point of view. Dean [1997] offers a simple model of an early Java class loader using HOL. He proves, using PVS, that newly loaded classes form a conservative extension to previously loaded classes. This means that any valid property remains valid, no matter how many further classes are loaded. Examples of interesting properties are the well-formedness and well-typing of byte codes.

In 1997, Saraswat [1997] published a bug related to type spoofing by use of dynamic class loaders. Saraswat's solution was based on checks for type-safety at runtime: byte code instructions such as `invokevirtual` need to check that the actual class of the object being operated upon is the same as the one obtained by the name resolution process. His informal discussion of the interaction of class loaders and type safety was the beginning of a formal underpinning of Java dynamic class loaders.

Sun's answer to the bug was a major change of the class loading mechanism of JDK 1.2, as described by Liang and Bracha [1998]. Java differs from other languages that also support dynamic loading (such as Common Lisp) because it performs checks at link-time as opposed to runtime; such an optimisation guarantees that checks need to be performed only once. Liang and Bracha's solution relies on three ideas:

(1) a distinction is introduced between the initiating and defining loaders of a class;

(2) a loaded class cache per loader maintains temporal namespace consistency;

(3) and namespace consistency between delegating loaders is preserved by a set of constraints.

Loading constraints are triples $\langle L_1, L_2, N\rangle$, consisting of two class loaders L_1 and L_2 and a class name N; their meaning is that using L_1 and L_2 as the initialising loaders for name N yields the same (loaded) class if they both succeed. The paper does not introduce any formalisation of this solution, but explains what constraints should be generated and how they should be verified. While it is not precisely specified when constraints should be verified, the paper explains that it should take place at link-time, which makes this solution different from Saraswat's.

Independently, Jensen *et al.* [1998] offer a formalisation of name space control and its interaction with Java visibility modifiers. They provide an abstraction of the JVM state describing the hierarchy of classes and members and their visibility rule. The main result is that the model provides an answer to the problem uncovered by Saraswat [1997]. The formalisation of Jensen *et al.* is criticised for containing some inaccuracies, both inadvertent and deliberate ones [Bracha 1999]. This shows that while formalisations often uncover problems, it is also necessary for system designers to scrutinise the formalisations of their systems to ensure that the formalisation agrees with their intention. Clearly, this kind of comments cannot be given if formalisations are not sufficiently accessible to system designers.

The difficulty of formalising the JDK 1.2 dynamic loading is due to the fact that, since the JVM loads classes lazily, the JVM cannot determine what class a name N denotes until the name N is actually resolved by the class it is declared in, using the defining class loader of that class. Tozawa and Hagiya [1999a, 1999b] present a formalisation of Liang and Bracha's loading constraint system, which highlight the subtle relationship between the constraint scheme and byte code verification. Their approach is to extend Stata and Abadi's byte code verification technique by typing rules to cope with class loaders. Their study only focuses on the `invokevirtual` and `areturn` instructions. The soundness is established by showing that the JVM preserves the well-typedness invariant.

Their formalisation allowed them to exhibit further flaws in the implementation of the JDK [Tozawa and Hagiya 1999a]. The first flaw was as a bug in the JDK implementation, and is another good illustration of the need for a complete formalisation of a system, down to the implementation. The second flaw is explained by the fact that the JDK does not verify its system classes, which indicates a major discrepancy between the formalisation and the implementation that needs to be addressed. Further work is required to integrate ignored Java features such as primitive types, field members, array types, member modifiers and threads.

Qian, Goldberg and Coglio [2000] offer another formalisation of dynamic class loading, expressed as a state transition system. The result is close to Sun's solution to the problem raised by Saraswat. Qian, Goldberg and Coglio present a type safety proof that relates the dynamic semantics with some static type information, currently loaded classes and currently posted constraint (which express requirements on not-yet-loaded classes). In Sun's implementation, the byte code verifier may have to resolve class names to check subclass relationships. The authors observe that this strategy is not the laziest possible. Therefore, they introduce a notion of subtype constraint, which is an important difference with respect to Tozawa and Hagiya's [1999a, 1999b] formalisation. Subtype constraints have the form $\langle L, N_1, N_2 \rangle$ and mean that if a loader L yields classes C_1 and C_2 for names N_1 and N_2, then C_2 must be a subclass of C_1.

Qian *et al.* formalised the JVM by its global state consisting of a loaded class cache, a set of loading and subtype constraints and a heap for storing objects. A valid state is a state defined as satisfying the type constraints. Loading constraints, subtype constraints, and the loaded class cache are constantly maintained in a mutually consistent state. The primary advantage is lazier loading, because no class needs to be loaded for verification purposes. Additionally, Qian *et al.*'s view of the verifier is simple: the verifier is just a component that takes a class as argument and returns a yes/no answer plus a set of subtype constraints as a result. The paper does not contain the full semantics, but handles only a few byte codes; the authors also make some simplifications to the JVM, and for instance, ignore multi-threading and exceptions. Type safety is expressed by showing that validity, i.e. satisfiability of all constraints, is preserved by state transitions; from a valid state, they show that it is always possible to move to another valid state, unless either there is no more code to execute or some of the listed conditions that correspond to runtime checks in the actual JVM fail. Coglio and Goldberg [2000], describe examples of type safety violation that the proposed formalisation avoids.

Fong and Cameron [1998] are concerned that byte code verification, class loading and linking are too tightly coupled, to allow for the three components to be verified effectively. Their proposal on a proof linking architecture uncouples the three components by making their interactions explicit in queries and updates on an underlying data base of facts. Before a class is loaded, the modular verifier generates facts about the class, and checks that these facts are consistent with those already present in the data base. The proposal does not compare the efforts involved in verifying a standard architecture versus to the newly proposed proof linking architecture, and as such is inconclusive.

5.4 Alternative Representations

Several authors propose alternative representations for an intermediate code than byte codes, with as main aim to avoid the problems and expense of byte code verification. Knoblock and Rehof [2000] propose Java Intermediate Representation (JIR), which sits between Java and the Java byte codes. In JIR local variables that are used with different types at different point in the execution of a program are notionally duplicated. Sub-type completion, a technique that basically creates a new type where one would otherwise have to work with a set of types, is used to typecheck programs. This yields a typing for JIR code that is closer to the typing found in Java programs, making type checking inherently more efficient than byte code verification. Gagnon and Hendren [1999] propose three address intermediate code, and represent type checking as a constraint satisfaction problem.

Some authors change the JVM to avoid the problems with modelling subroutine polymorphism. Others reject the JVM language, and instead suggest the use of tree-based code. Indeed, to provide aggressive optimisations, compilers rely on the control- and data-flow information that is readily available in an abstract syntax tree. Kistler and Franz [1996] Slim binaries avoid the problem of code scattering as in Figure 1 by using a tree structured intermediate code or ‘slim binaries’. These allow the allocation and initialisation to remain coupled, just as in the Java program. Unfortunately, slim binaries require rather more memory and processing power than is readily available on small systems.

5.5 Discussion

The JVM language bears some similarity with typed assembly languages, such as TAL [Morrisett et al. 1999]. The JVM offers less polymorphism than TAL but supports subroutines absent from TAL. Stata and Abadi have proposed a formal framework that can express these features and that has been extended by several authors to cover byte code verification for most of the JVM language.

From a flourishing activity on the dynamic class loader, the use of constraints has emerged as a good mechanism to express properties to be verified by classes not loaded yet. The interleaving between byte code verification and class loading is definitely subtle. A complete integration of these ideas in Stata and Abadi’s framework is an important target, but its size probably requires automatic tools to handle the formalisation.

Deriving a correct and efficient implementation of these formalisations is also a challenge. We refer to Section 7 for a survey of methods that can be applied in that context.

In Section 4.5, we indicated the need for tools able to expose security properties of programs. As the JVM language has different observable properties than the Java language, tools working on the JVM byte code would also be useful (furthermore, because the source code may not be available for analysis).

6. THE COMPILER

In Sun’s reference implementation, the compiler `javac` takes as input Java classes and generates their byte code representation, organised as class files, and ready to be loaded by the JVM. In this section, we survey the formalisations of this compilation

Table 3. Combined Java and JVM languages

Reference	Small	Purpose	Base	Tool	CL	EH	MT	Semantics	Proof
[Börger and Schulte 1998]		prove compiler correct	[Börger and Schulte 2000a]		yes	yes	yes	ASM	no
[Börger and Schulte 2000b]		prove compiler correct	[Börger and Schulte 2000a]	AsmGofer	no	yes	no	ASM	yes
[Choi et al. 1999]		static analysis			yes	yes	yes	non standard	no
[Diehl 1998]		study semantics			no	no	no	ASM	no
[Flanagan and Freund 2000]		static analysis			yes	yes	yes	CR	no
[Nipkow et al. 2000]		prove type soundness	[Pusch 1999]	Isabelle	no	no	no	embed	yes
[Rose 1998]	yes	prove compiler correct			yes	no	no	NS	no
[Stärk 1998]		prove compiler correct	[Börger and Schulte 2000a]		no	yes	yes	ASM	yes
[Stärk et al. 2001]		study of semantics, verification	[Börger and Schulte 1999b]	AsmGofer	yes	yes	yes	ASM	yes
[Whaley and Rinard 1999]		static analysis			yes	no	yes	non standard	no

Legend:

Small	Whether the work is targeted at smart cards or small footprint devices
Purpose	Main purpose of the author for writing the paper
Base	A reference to work on which the current work is based
Tools	The formal methods or semantics tools used
CL	Class loading
EH	Subroutines/exception handling
MT	Memory model & multi-threading
Semantics	Style of semantics used
Proof	Whether the paper presents proofs or proof sketches

process, and examine some of the proposed static analysis methods to improve the performance of the generated code. However, the open Java architecture offers new opportunities to translate to and from different representations: we also survey the compilation of Java source code to alternative representations, and byte-code to byte-code transformations, which may be used at load time.

The section begins with a presentation of the work that discusses the combination of Java and JVM semantics. The only work we have been able to find that complements Java and JVM semantics with a specification of the compiler is discussed in a separate section on the Abstract State Machine approach. Table 3 includes summaries of the efforts described below.

6.1 Relating Java and JVM Semantics

Abadi [1999] observes that the compilation process of the Java language, i.e. translating Java source code into JVM byte code, is not fully abstract. He shows that some compiler generated byte code, put in the context of some JVM-valid byte code (though not necessarily generated by a compiler), exhibits different properties than its original source program. In technical terms, the JVM is said to have a finer notion of observational equivalence, i.e. the JVM is able to distinguish more Java programs than the Java language itself.

Diehl [1998] gives the compilation schemes for a subset of the Java that excludes exception handling, multi-threading and garbage collection to the corresponding subset of the JVM. He also offers an operational semantics of this JVM subset. No specification of the Java subset is given, thus missing the opportunity to prove the compilation schemes correct.

Rose [1998] gives a natural semantics of a subset of Java, the corresponding subset of the JVM, static type systems for both and a specification of the compiler for the subsets. No proofs are given either of the soundness of the type systems, or of the correctness of the compiler.

With μJava, Nipkow *et al.* [2000] offer a HOL/Isabelle embedding of Java's imperative core with classes. They present a static and a dynamic semantics of the language at the Java and JVM levels. The objective of the paper is to show that the soundness property of the type system can be parameterised over the operational semantics (i.e. Java or JVM). This achieves a significant level of reuse in the soundness proofs of the type system. The authors do not discuss the scalability of the approach to full Java.

6.2 Abstract State Machine Approach

For a number of years, Börger *et al.* have been working on formal specifications of Java, the JVM and the compiler. All their work is based on the Abstract State Machine formalism, a full semantic account of which may be found in Gurevich [1995]. Two earlier papers specify a modular semantics of a subset of the JVM [Börger and Schulte 2000a], and a subset of Java [Börger and Schulte 1999b]. Both specifications follow a modular approach, where each new feature is added to the specification as a conservative extension. The two subsets do not entirely coincide; for example, the Java specification includes multi-threading but the JVM specification does not. This makes the two subsets somewhat less ideal as a basis for further work to specify the compiler and to prove the compiler correct with respect to the semantics of Java and the JVM. Yet in a third paper [Börger and Schulte 1998] this is exactly what is done, by further reducing the subsets of Java and the JVM to omit Multi-threading, class loading and arrays. The main result is an informal theorem stating the correctness of the compiler. Two further papers by the same authors revisit exception handling and object initialisation, again based on the two initial papers. The first of these further papers [Börger and Schulte 1999a] reports on problems with the initialisation of objects, for which the official Sun documentation provides conflicting information. The problems were identified thanks to the building of the specification. The second paper [Börger and Schulte 2000b] revisits the exception handling mechanism of Java, the JVM, and the compiler. The main result is a formulation of the correctness of compiling exception handling, with a full proof. Stärk [1998] revisits the specification of Java and the JVM from Börger and Schulte [2000a, 1999b]. Stärk also presents a compiler from the imperative core of Java enriched with method calls and gives a correctness proof of the compiler with respect to the semantics of Java and the JVM for the same fragments.

The forthcoming 'Jbook' by Stärk, Schmid and Börger [2001] provides a complete revision and extension of this earlier work described above. The main contributions of the Jbook are:

—An ASM specification of Java as a series of five layers for the 1) Imperative core, 2) Static Classes, 3) Object Orientation, 4) Exceptions and 5) Threads. Each layer (except the first) is a *conservative extension* of the previous.

—A specification of the JVM with the same layering as for Java.

—A novel bytecode verifier computing principal types, and a loading component.

—A specification of the compiler for the first 4 layers, i.e. excluding threads.

—All specifications are machine readable; they are executable using the AsmGofer toolkit. This includes an algorithm AsmGofer uses to compute the types for a program if it is typable at all.

—Formal proofs of the soundness of the Java thread synchronisation model, the type soundness of Java and the type soundness for the combination of the JVM with the byte code verifier.

—A formal proof of the correctness of the compiler with respect to the Java and JVM specifications, and a formal proof that the compiler is complete with respect to the bytecode verifier.

The Jbook does not consider Java packages, compilation units, garbage collection, and class loading at the Java level – class loading at the JVM level is taken into account. The memory model is an abstraction of Java's mechanism to keep local working memories consistent with the main memory. However, the Jbook represents a significant amount of work and gives the most comprehensive and consistent formal account of the combination of Java and the JVM, to date.

Wallace [1997] gives a reasonably complete specification of Java, also based on the ASM framework, but not closely related to the work of Börger and Schulte. Wallace's work includes multi-threading, and exception handling, but excludes class loading and garbage collection. The work is purely a study of semantics.

6.3 Static Analysis

A quick look at the proceedings of conferences and workshops on programming languages indicates that various static analyses have been adapted to or specifically devised for Java. Static analysis and associated optimisations are expected to be correct, i.e. they are supposed to preserve the meaning of programs. Even though this is an important aspect of security, a complete survey of this domain is beyond the scope of this paper. However, we just draw our attention to two types of analysis that study some Java specific aspects: escape analysis for object-oriented languages and analysis related to multi-threading. (Another frequent analysis concerns array bound checks.)

Whaley and Rinard [1999] present a combined pointer and escape analysis algorithm for Java programs. Using a data flow approach they construct a graph which characterises how variables refer to other objects in the program. Using such information, they perform a transformation that removes unnecessary synchronisations. Additionally, they determine when objects may be allocated in the local stack instead of the shared heap. An interesting aspect of their approach is its compositional nature, according to which they analyse methods independently of their callers and the methods they call. Such a compositional approach is essential for component-based systems where portions of the code may be loaded dynamically; Sreedhar *et al.* [2000] also share such a preoccupation in the design of their Jalapeno compiler. An alternative analysis used for similar optimisations is described by Choi *et al.* [1999]. In some benchmarks, these analyses remove between 20–80% of synchronisation operations, and are able to perform stack allocations for 20–95% of the objects. Overall performance improvement in the range 2–20% were observed.

Flanagan and Freund [2000] present a static analysis of Java programs able to detect race conditions in multithreaded programs. Their analysis is based on a new type system, which like an "effect system" [Talpin and Jouvelot 1992], specifies the locks required by expressions. Their formalisation is based on an extension of Flatt *et al.*'s ClassicJava with `fork` and `synchronized` statements [1999]. No inference algorithm is offered, so the programmer is required to annotate the program with lock-related type information.

6.4 Byte Code Modification

Several proposals have been made to enhance the security of Java programs by automatically modifying incoming applets, some standard classes or both to restrict the damage that can be done by rogue applets. Balfanz and Felten [1997] rename the standard `AppletClassLoader` to `XppletClassLoader`, and then insert a new `AppletClassLoader` that subclasses `XppletClassLoader`. The new class pops up a security dialogue, offering the user full control.

Shin and Mitchell [1998] not only rename classes but also methods. This is necessary because final classes cannot be subclassed. They show how some common attacks can be avoided using their method.

Malkhi and Reiter [2000] go another step further by acknowledging that the best defense is to separate a potential attacker from its victim. Their system modifies applets, so that they may be run on a separate system called the playground. An applet may then freely access the resources of the 'playground', to which no real harm can be done. Finally, byte code rewriting has also been proposed to 'macro expand' subroutines with the aim of avoiding subroutine polymorphism, and all the modelling problems this brings about [Freund 1998].

6.5 Byte Code Annotation

The class file format is quite flexible in the sense that it allows extra information to be supplied with compiled Java programs. In parallel to the stream of JVM byte codes, Hummel *et al.* [1997] provide a stream of annotations that enable the JVM to recreate in a cost effective way structured, high-level compiler information. One particular application of the technique is to provide indications for register allocation, which would otherwise require detailed information and analysis of programs. A significant advantage of byte code annotation is that the extra information in class files can safely be ignored by implementations that do not support it. The main disadvantage is that there is no guarantee that the annotations actually correspond to the byte codes, thus creating potential security loopholes.

6.6 Discussion

It has proved difficult to implement Java safety features correctly. Indeed, building a Java system with acceptable performance requires various optimisations, which basically distribute the implementation of safety features throughout the compiler and different parts of the runtime system. The various components responsible for safety interact in complex ways, creating scope for design and implementation problems.

The first implementations of Java were based on byte-code interpreters, and were rapidly replaced by just-in-time compilers able to transform byte codes into assembly code. We are now observing the emergence of a new generation of optimising compilers such as Jalapeno [Sreedhar et al. 2000]. Moreira *et al.* [2000] show that by applying aggressive optimisations for Java's precise exception model, array bound checking, and floating point semantics, Java can become competitive in performance with Fortran for numeric applications. However, proving the correctness of such compilers is a daunting task, due to the levels of refinements that would be required. Such a whole process has rarely been performed for a complete language: we are only able to mention the verified implementation VLisp [Wand and Guttman 1995], including compilation to byte code, byte code verification and memory management for the language Scheme, which is definitely smaller than Java.

It is not uncommon for static analyses to require whole programs as input, but such an approach is not suitable for programs loading code dynamically. We have seen a couple of papers addressing this problem, and we expect this topic to be the subject of further investigation as component-based systems become more prevalent.

Commercial interests understandably prefer preserving the compatibility with existing Java installations, but differences between the JVM and the Java language do not constitute the most natural route to build safe and secure systems. Some of the alternative approaches may turn out to be valuable in the future.

7. JAVA VERIFICATION

Program verification contributes to the safety of Java programs because formally verified programs contain fewer design and implementation problems. We review approaches to program verification based on model checking and on theorem proving. We discuss these two techniques in the following sections. The LOOP project aims to develop a comprehensive method and tools to verify Java programs and the Java API. This is discussed in a separate section (7.3). Code certification is a program verification technique that is particularly appropriate for mobile code. We discuss this technique in Section 7.4.

7.1 Model Checking

Demartini *et al.* [1998], and also Havelund and Pressburger [2000] describe how core features of Java can be mapped onto the Promela language of the SPIN model checker. Both model multi-threading and objects, Havelund and Pressburger also model exceptions. Both approaches model the objects using Promela's arrays, with one array element per instance of the class. The resulting models quickly grow too large to model check effectively. Both approaches only check for safety properties (e.g. assertions, deadlock), and do not provide support for the checking of liveness properties. One of the most useful features of the SPIN model checker is its ability to display scenarios leading to problems such as deadlocks. Demartini *et al.* take care to relate these scenarios back to the original Java sources, making their tool more user friendly than that of Havelund *et al.*. For small programs a naive mapping from Java to the input language of a model checker is useful, it is difficult to see how the results might scale up to larger systems.

Jensen *et al.* [1999] use model checking to verify properties of Java programs. To address the scalability issue, they use a more abstract approach than Demartini *et al.*, or Havelund *et al.*. In the proposal by Jensen *et al.*, static analysis techniques are used to reduce a Java program to a control flow graph with only three operations: method calls, method returns and assertions. A simple operational semantics of the three operations defines the state transitions of the abstract Java program, and linear temporal logic formulae define the properties of the system. As an example of use, Jensen *et al.* show how the system can be used to model stack inspection and the Java sandbox [McGraw and Felten 1999].

The Bandera project [Corbett et al. 2000] provides semi-automated tools for mapping Java programs to the input languages of SPIN and SMV. The three main tool components are a program slicer, an abstraction engine, and a two way translation system. The first two components both serve to automate the creation of models with smaller state spaces than the original Java programs. The slicer removes code and data irrelevant to the properties of interest. Under user guidance, the abstraction engine replaces concrete data sets by smaller abstract data sets. The translation system maps Java into the native notation of the model checker, and it maps counter examples found by the model checker back into Java. The strength of the Bandera system is its automated support for creating scalable models.

Stiles [1998] summarises various research projects proposing the combined use of CSP and Java. The idea is that a concurrent program is best specified first in CSP, for which a considerable body of theory, as well as some state-of-the-art modelling tools exist. One of those tools, the model checker FDR is then used to verify the specification. Finally, the specification is translated into Java using one of several Java classes implementing CSP style communication [Hilderink et al. 1997; Welch 1997].

7.2 Theorem Proving

Detlefs *et al.* [1998], using Modula 3, and more recently also Java [Leino et al. 1999], go beyond the verification offered by type checking and require the programmer to annotate programs with pre- and post-conditions. They observe that programmers do this informally anyway, and claim that it is not a big step to ask them to annotate their programs formally. The compiler is then able to generate and prove the verification conditions (using a form of Dijkstra's weakest pre-condition calculus) that need to be satisfied for the pre- and post-conditions to hold. The system of Detlefs *et al.* does not require the programmer to annotate programs with loop invariants and variants, which most programmers would find harder to write than just pre- and post-conditions. Instead the system derives loop invariants automatically, which, as a consequence, are weaker than those provided by humans. Alternatively, the system may be directed to assume that loops are executed at most once, thus giving rise to conservative approximations to the real behaviour of loops. The system is a compromise between what is achievable with automated techniques and what programmers are able to provide. The system is therefore more powerful than a type checker, but less powerful than programming with full verification.

Poetzsch-Heffter and Müller [1999] give an operational and an axiomatic semantics of a subset of Java (the imperative core and method calls). They then prove the soundness of the axiomatic semantics with respect to the operational semantics. Their axiomatic semantics can thus be used to as a basis for the verification of Java programs. Both types of semantics are also embedded in HOL, so that mechanical checking of the soundness proof would be feasible in future.

Verification is not restricted to Java programs. Moore [1999] has built a new version of a small subset of Cohen's [1997] specification of the JVM. Moore shows how the ACL2 theorem prover is capable not only of executing simple byte code programs, but also of proving the correctness of such programs with respect to a specification.

If Java safety would be able to guarantee that computations terminate, and within certain bounds, then the denial of service attack would be prevented, which is clearly a desirable safety goal. However, execution time is probably one of the most difficult resources to control. While there are languages [Anderson and Witty 1978] and type systems [Crary and Weirich 2000] that have been designed to guarantee termination, we have not been able to find efforts that apply such technology to Java.

7.3 The LOOP Project

The aim of the LOOP project is full verification of Java programs. The LOOP methodology allows the Java programmer to annotate a program with appropriate specifications. The LOOP compiler [van den Berg and Jacobs 2001] translates the program and the specifications into the Higher Order Logic (HOL) of a theorem prover, such as PVS [Jacobs et al. 1998], or Isabelle/HOL [van den Berg et al. 2000].

The user then drives the tool to prove that the programs satisfy the specifications provided by the annotations. Examples of properties include guaranteed termination of a method, invariants on the fields of a class, or the absence of exceptions. While the theorem provers provide a degree of automation, user guidance is often required for example to decide on proof strategies.

The LOOP embedding of Java programs into a HOL is based on a set of theories representing the denotational semantics of sequential Java [van den Berg et al. 2000]. Jacobs [2001] presents a coalgebraic view of exceptions, emphasizing the state-based aspect of computations, and its suitability for reasoning in a Hoare-style logic. This formalisation has highlighted an implicit assumption in the first specification of Java that "a thrown exception is never the null-reference". The semantics of Java have been validated by showing a close, almost literal correspondence between the natural language specification of the Java Language specification and the appropriate theories [Huisman 2001]. With a denotational semantics it is not always easy to model irregular control flows in a clear and concise fashion. Therefore Jacobs explores the use of a monad to structure the semantics of break statements, exception handling etc [Jacobs and Poll 2000].

LOOP specifications are embedded in Java programs by way of annotations in the Java Modelling Language (JML) [Leavens et al. 1999]. JML is designed to mimimise notational burden by using the syntax of the host language (Java) as possible for the specifications. Compiled JML specifications are represented in HOL by a set of tailor made proof rules [Jacobs and Poll 2001].

The LOOP tool and methodology has been applied extensively to the smart card API for Java Card (See Section 8). In the first case study, the methods in each class have been annotated with a lightweight specifications stating the precondition, which, if satisfied guarantee the absence of unexpected exceptions. This is useful information for the Java Card programmer, because unexpected exceptions are a safety hazard, and also for the compiler because handlers may be omitted. The Application Protocol Data Unit (APDU) Class has been fully specified [Poll et al. 2001].

The Application Identifier (AID) class maintains a unique manufacturer ID and a suffix as a string of a certain length. The AID string is used to identify applets and as such forms an important aspect of the security management in a Java Card. Therefore van den Berg *et al.* [2000] prove that the length of the string satisfies certain constraints as a basis for further reasoning about applet security.

A third case study is the verification of a non-trivial invariant for Java's Vector class, stating that "the number of elements in the array of a vector never exceed its capacity" [Huisman et al. 2001].

While there are many more useful properties that could be specified and proved, a solid foundation for the verification of the Java Card API and applets has been laid.

7.4 Code Certification

Necula [1997] introduced the idea of proof carrying code (PCC). This is a partly automatic verification technique for assembly level programs designed to allow a code consumer to have trust in a code producer. PCC has been been used with compilers for ML [Necula 1997], type safe C [Necula and Lee 1998a] and more recently Java [Colby et al. 2000].

PCC works as follows. Suppose a consumer wishes to receive code from a producer. The consumer establishes a safety policy for code it is willing to accept, and communicates this policy to the producer. The latter might be a compiler, Web site or other source that is not trusted by the consumer. The producer uses the safety policy to annotate the code destined for the consumer with a safety property in terms of loop invariants, pre- and post-conditions. The producer generates a proof of the safety property, either by hand, or using a mechanical proof assistant. The consumer receives the code and the proof, and mechanically checks that the proof is consistent with the program, and therefore that the program satisfies the safety property. It is more difficult to generate a proof than to check a proof. Therefore, separating the two phases has a significant benefit: The consumer does not need to trust the producer, or the means by which the producer creates the code and the proof. Instead, the consumer relies only on a small trusted infrastructure consisting of what is essentially a type checker. This is reported to be no more than 5 pages of C code in size.

Initially, the PCC approach suffered from proof sizes exponential in the size of programs [Necula and Lee 1998c]. A proof may become large because of the amount of redundancy. In a later paper, Necula and Lee [1998b] show that it is possible to reduce a proof of size n to a proof of size $\sqrt{n}$ by avoiding some redundancy. They also give practical examples of small programs (e.g. quick sort) with acceptable proof sizes. Recently, Necula and Rahul [2001] have shown that oracle based techniques allow for even smaller proofs. For a given benchmark proofs are on average 12% of the code size. These results apply to the same kind of verification as offered by the Java byte code verifier, but in the case of PCC for Intel x86 machine code rather than Java byte codes. The proofs of more powerful safety properties would still be larger.

Kozen's [1998] proposal for code certification does not suffer from large proof sizes, but it is also strictly less powerful than that of Necula and Lee. For example, Kozen's technique cannot make a distinction between different elements of an array. Kozen's notion of code certification is roughly as powerful as that of Java byte code verification. The differences are firstly that Kozen maintains the structural information in compiled code that is absent from JVM byte codes. This greatly simplifies the verification process. Secondly, Kozen targets native machine code, while the JVM offers portability.

7.5 Discussion

An informal argument may sometimes suffice to show that (safety) properties of programs hold. Formal program verification techniques are needed to derive strong guarantees that programs have certain desired properties. However, program verification requires special skills, to formulate properties, to discover appropriate conditions, such as loop invariants, and to drive specialist tools. Model checking techniques tend to be more automatic but have difficulty coping with problems of increasing size. Computer supported theorem proving techniques often require manual intervention but generally cope better with larger problems. Unfortunately, relatively few programmers are trained in program verification, thus hampering the widespread use of the techniques.

A number of groups are working on program verification techniques that are specifically adapted to Java. Using such specialised tools and techniques relieves the programmer from having to deal with many irrelevant issues because the tools cope with the details. To achieve this, considerable effort must be spent embedding the semantics of Java in program verification tools. Of course, some of the prior effort invested in specifying the semantics of Java is implicitly reused in the development of program verification tools. However, we have not seen much explicit reuse of programming language semantics for the purpose of building verification tools. This may be due to the fact that program verification is the domain of the formal methods community whereas the main practitioners of semantics are within the programming languages community.

Many formal methods and semantics tools have been used to study aspects of Java: ACL2 [Kaufmann and Moore 1996], ASM [Gurevich 1995], B method [Abrial 1996], Centaur [Borras et al. 1988], Coq, DECLARE, ESC/Java [Leino et al. 1999], FDR, Haskell, Isabelle [Paulson 1994], HOL, JML [Leavens et al. 1999], LETOS [Hartel 1999], PVS [Owre et al. 1995], SMV [McMillan 1993], SpecWare [Srinivas and Jullig 1995], SPIN [Holzmann 1997]. Not all of those tools are sufficiently automatic, or adequately equipped with the right mathematical theories to prove safety properties of Java programs.

There is no clear winner amongst the various methods and tools used. The Abstract State Machines has been used to build the most comprehensive set of specifications, complete with experimental and mathematical analysis. Isabelle/HOL is one of the most popular tools, but even its users complain about lacking mathematical theories and validation facilities [von Oheimb and Nipkow 1999]. This clearly needs improvement.

8. JAVA ON SMART CARDS

In this section of our survey we present a case study on Java Card, the smart card 'dialect' of Java [Chen 2000]. Java Card offers an important application domain of the tools and techniques that we have discussed, because smart cards are used for business critical applications. These include banking, access control, and health care, and more recently the SIM card in the GSM mobile phone. The safety (and security) of Java applications on smart cards is of prime importance to the smart card manufacturers, issuers, and users [Lanet 2000].

Java Card is not plain Java, because Java implementations are too resource hungry for smart cards. A JVM implementation requires at least 1 MB of store [Webb 1999]. This makes Java acceptable for use in PCs and capacious embedded controllers but less than ideal for use in small footprint devices, such as mobile phones, and PDAs. Even the K Virtual Machine (KVM), which has been designed specially to fit into small footprint devices requires at least 128KB of RAM [Sun 1999]. This is still too large, as a smart card typically offers a few hundred bytes of RAM and a dozen or so KB of EEPROM.

To reflect the limited computing resources inherent to smart cards, the Java Card VM and API impose restrictions. For example, there is no support for threads, garbage collection, or real numbers. While the standard word size for Java is 32 bits, for the Java Card VM this is 16 bits. Instead of relying on middle-ware products, the Java Card API includes a simple transaction facility built in the VM.

The Java Card API includes an interface to the ISO [1995] standard 7816-4 for the Application Protocol Data Unit (APDU) format of communication between smart card and terminal. Using this rather low-level protocol in Java is somewhat cumbersome, but Java Card applications are fully compatible with legacy terminals. Without this compatibility, an evolutionary approach for introducing Java Card technology into the market place would not work. Finally, Java Card implementations do not provide generic auditing facilities, which makes it difficult to evaluate the effectiveness of the Java Card security mechanisms. Instead, it is left to the Java Card application programmers to ensure that appropriate information logging is maintained.

The differences listed above create a number of serious disadvantages to the successful deployment of Java Card:

—A semantics of standard Java, or a formal method tailored to standard Java is not directly applicable to Java Card. Instead some of the formalisation needs to be revisited.

—The full potential and flexibility of client-server software development cannot be realised because developers need to be aware of the platform on which their code is going to run (i.e. on or off card).

—Java applets running on smart cards cannot be verified appropriately before they are run because the full byte code verifier is too large. Current stopgap measures include digital signing of pre-verified byte codes.

—The freedom of code migration is restricted because not all platforms support full Java.

In addition to those inherited from Java and the JVM, Java Card has some safety and security problems of its own. For example Montgomery and Krishna [1999] show how the security of the Java Card object sharing model can be broken. Oestreicher [1999, 1999] discusses the Java Card memory model. The model is not obviously flawed, but it is rather baroque, and therefore a potential source of security problems when used incorrectly. Hartel and de Jong [2000] present a critique of the Java Card safety and security mechanisms.

In the remainder of this section we present work based on the core semantics of Java Card and the Java Card Virtual Machine (JCVM). Then we discuss the two novelties of Java Card. The first is the applet firewall, which offers a mechanism for controlled sharing of objects between applets (Section 8.3). The second innovation is the class file converter, which reduces the amount of space needed to represent loaded classes (Section 8.4). We then revisit byte code verification in the specific setting of the JCVM.

8.1 Java Card Core Semantics

Attali *et al.* [2000] leverage the Centaur tool suite to build a programming environment for Java Card. Java Card programs can be entered using a syntax directed editor, which knows for example which methods must be implemented by an applet. This helps Java Card programmers over the initial hurdle of getting to know the details of Java Card. The programming environment is able to interpret Java Card programs. The novelty is in the analyzer, which extracts the APDU commands from a Java Card program. The analyzer then generates a terminal application, which is able to send the required APDU commands to the simulated Java Card. This enables developers to explore the behaviour of their applets at a reasonably high level of abstraction. The paper does not explain how the APDU analysis works. The tool does not cope with some of the extensions that Java Card offers over Java, such as transactions, and object sharing.

8.2 JCVM Core Semantics

Hartel *et al.* [1999] provide a complete specification of a precursor to the JCVM, the Java Secure Processor (JSP). The JSP subset excludes multi-threading, garbage collection and exception handling, mainly because the limited resources on a smart card would not be able to support these features. The specifications have been validated using the LETOS tool [Hartel 1999].

An interesting methodological point to note is that the JSP and also the JCVM were designed essentially by starting from the full JVM, and then cutting back unwanted features. The newer KVM [Sun 1999] on the other hand has been designed from scratch, adding features as required. This latter method is more likely to yield a coherent result and is therefore recommended [Taivalsaari et al. 1999]. The developers of the picoPERC version of the JVM take a different and promising looking approach. They offer a core VM (still requiring 64KB) and provide tools to add further functionality to the core VM. Unfortunately, no details are provided in the paper [Nilsen 1998].

8.3 Applet Firewall

In Java Card the only entity that can ‘own’ anything is a context. A context is associated with a package; contexts are unique. The applets defined within a package are owned by the context of the package. All objects created by an applet are also owned by the context of (the package of) the applet. The Java Card Runtime Environment (JCRE) is represented by a pseudo context, which is the owner of any objects not created by an applet (i.e. system objects).

Objects owned by the same context may be shared freely. Objects owned by different contexts can only be shared if the sharable interface object protocol, which we explain below, is followed. Ownership is thus a relation between contexts and objects. It would have been more natural to develop an ownership relation between objects, but that has not been the choice of the Java Card designers.

In addition to the notion of ownership, Java Card defines the notion of object access. An applet may grant another applet access, again subject to the sharable interface object protocol. An owner may invoke methods on an owned object, read and write fields etc. An applet that has access to a shared object (i.e. an object owned by someone else) may invoke methods on the shared object, but the applet may not access fields of the shared object.

The sharable interface object protocol works as follows. Suppose that a server applet wishes to grants a client applet access to an object owned by the server. The server must 1) define a new public interface (`I`) that extends the standard Java Card interface `Sharable`, 2) define a class (`C`) that implements the interface `I`, and 3) create an object `SIO` of class `C`.

The client identifies the server by its Applet Identifier (AID), and asks the JCRE for a reference to the servers SIO object. The JCRE then invokes a method defined by the server, which enables the latter to decide whether to grant or deny access to the client. If access is granted, the JCRE returns the refence to the SIO to the client, which can then begin to invoke methods on the object. We have glossed over some details here [Chen 2000], but it should be apparent that the protocol is non-trivial. A formal investigation into the properties of the protocol is thus useful.

Motré [2000] describes a formal model (using the B-method) of the applet firewall. In the model a number of simplifications have been made, such as the omission of looping constructs from the JCVM component. The main results of the work are twofold. First, the constructed model has been verified entirely. Second, to reduce the number of proof obligations that need to be discharged during verification, it is argued that it is a good idea to break the specification up in a number of separate B machines. Spending more effort to build a specification that extends beyond the firewall component, and relating the specification back to the reference implementation of Java Card would be valuable future work.

Bieber *et al.* [1999] use the SMV model checker to establish the conditions under which unexpected information leaks between applets can be excluded. Their approach is based on abstract interpretation of Java Card byte codes, where the domain of the interpretation is a partial order based on levels of security. Bieber *et al.* perform their analysis at the byte code level, which makes it particularly difficult to keep the state space sufficiently small.

8.4 Class File Conversion

Smart cards are too small to contain all the information of Java's class files, or the byte code verifier. Therefore, the Java Card Architecture has a special component, the class file converter. This component takes standard class files and converts them into smaller called Card APplet (CAP) files. The class file converter also implements the byte code verifier. This means that a smart card cannot rely on its byte code verifier to ensure that CAP files are well formed and type correct. Instead cryptographic techniques are used to ensure that the contents of CAP files can be trusted.

Lanet and Requet [1998] use the B-method (and the associated toolkit 'Atelier B') to study one particular aspect of the conversion from JVM to JCVM code. This is the optimisation that replaces JVM instructions with `int` type arguments by JCVM instructions that take `byte`, `short` or `int` as appropriate. Their results for a small subset (the imperative core and method calls) of the JVM byte codes and the corresponding JCVM byte codes include:

(1) A specification of the constraints imposed by the byte code verifier for the JVM subset.

(2) A specification of the semantics of the JVM subset.

(3) A specification of the semantics of the JCVM subset.

(4) A proof that the specification of the JCVM subset is a *data refinement* of the JVM subset.

The subset is small, and the differences between the JCVM and the JVM are small. However, the work by Lanet and Requet shows how the B-method can be used successfully, and succinctly to make the proof.

The same authors use the B-method to prove the JCVM type system sound. The idea is to specify a JCVM with runtime type checks first. This specification is then refined into a byte code verifier and a byte code interpreter without runtime type checks. The correctness proof of the refinement then constitutes a soundness proof of the type system. An early paper takes a small subset of the JCVM byte codes into account [Casset and Lanet 1999]. The scalability of the method, covering about 100 different byte codes, is investigated in a later paper [Requet 2000]. Even though the work is “not representative of the tricky parts of the full instruction set”, the effort required is considerable.

Denney and Jensen [2000] study an aspect that is complementary to that studied by Lanet and Requet. The former study the conversion of JVM class files to JCVM class files by a ‘tokenisation’ process. This replaces names in the class files by more compact representations, thus reducing the size of the class files as well as speeding up the loading process. Denney and Jensen take essentially the same four steps as Lanet and Requet above. However, Denney and Jensen use the Coq theorem prover to mechanically check their proofs. They also use an elegant method to parameterise their operational semantics over name resolution. Therefore, only one operational semantics is required, that is abstract with respect to the actual name resolution method, and thus common to both the JVM and JCVM subsets.

8.5 Lightweight Byte Code Verification

A small footprint device does not have enough memory to perform byte code verification. Sun’s split VM concept for Java Card stipulates off-line verification, and signing the results digitally. When loading the code, all that needs to be checked is the signature, not the code itself. This places considerable trust in digital signatures: once the underlying keys are compromised, verified byte code becomes worthless.

Instead of a verifier based on type checking, Posegga and Vogt [1998], and Basin *et al.* [1999] propose to use a model checker to perform off-line byte code verification for smart cards. Their argument is that a tried and tested model checker (SMV) is easier to trust than a Java byte code verifier, but they give no evidence for this claim. In a separate paper, Goré *et al.* [1998] propose to implement a tiny proof checker on a smart card. The proof checker would then be able to reason about trust policies set by the user. The result appears to be somewhat disappointing, as proving theoremhood of some simple first order logic formulae may take of the order of minutes.

With lightweight byte code verification, Rose and Rose [1998] do not wish to rely on digital signatures for the safety of byte code verification on smart cards. Instead they use Necula and Lee’s proof carrying code method to ‘split’ the byte code verifier as follows. The first step (the verification) is to reconstruct the types associated with all local variables and stack locations of JVM code. The second step (the certification) uses the reconstructed types to check that each instruction is correctly typed. The advantages are twofold. First, the certification process is simple, so that it is feasible to implement it on a smart card; the more complex verification can be carried out on a host. Second, only the certification needs to be trusted, not the verification. This makes the trusted infrastructure smaller than in a standard Java implementation. Rose and Rose show that for a small subset of the JVM, consisting essentially of parts of the imperative core with method calls, certification is sound and complete. This means that the separated verifier and checker agree exactly with the original byte code verifier. The paper contains some errors, which could have been avoided if Rose and Rose had used tool support. Furthermore, exception handling has been omitted, because it would complicate byte code verification considerably [Stata and Abadi 1998].

Sun's KVM has a footprint at least 10 times larger than Java Card. This makes it possible to perform proper on-line byte code verification, but on simplified (pre-verified) class files. Indeed the Connected Limited Device version of the KVM [Sun 2000] implements byte code verification as proposed by Rose and Rose (above). The pre-verifier removes all subroutines (by inlining) and adds stack maps to the class file. These stack maps record the types of the locals at each branch target. The on-line verifier makes a single linear pass through the byte codes to check type consistency. Unfortunately, no formalisations of the KVM byte code verifier have been reported in the literature.

Klein and Nipkow [2000] prove a slight variant of lightweight byte code verification correct using Isabelle. Their presentation is clearer than the original by Rose and Rose.

8.6 Discussion

Java Card is at the same time a subset of Java (due to the omission of multi-threading, garbage collection etc), and a super set of Java (due to the addition of a model for controlled object sharing etc). Theories and tools developed for standard Java are therefore not directly applicable to Java Card. Special effort is required to develop tools and theories that can cope with the added features. More seriously, extra efforts are required to adapt or redevelop tools and theories to handle standard features that have been modified for Java Card. For example class loading in Java Card is package based as opposed to class based, and changing final static fields of primitive types is a binary compatible change in Java, but not in Java Card.

We believe that some of the incompatibility problems could be avoided. For example, to specify a KVM, one gives a *configuration* to determine which VM features are supported, and which API's. In addition, a *profile* states which additional, application domain specific API's are present. In a sense both the configuration and the profile are informal specifications of how a core VM is to be extended. Theoreticians have developed efficient methods for extending a semantics, stating the conditions under which theorems proved about a semantics carry over to an extension of the semantics [Fokkink and Verhoef 1998]. Clearly there is scope for applying such powerful theoretical results to the practical development of small Java implementations.

Looking to the future one might expect smart cards to become more powerful, but at the same costs. This will create a demand for more mainstream Java features to be made available on Java Cards, such as garbage collection, or a security manager. Apparently, most vendors of Java Cards already provide a form of garbage collection. The opportunity to create a novel, high integrity Java Card is thus here. We believe that enough work has been done to form a solid basis for a high integrity version of Java Card.

9. CONCLUSIONS

Our survey indicates that the formalisation of Java is actively researched by both the programming language and formal methods communities. In particular, the novel features of Java have been investigated. Initial studies focused on specific components in isolation of the rest of the system, but more recent work has been attempting to propose general frameworks encompassing several components of the Java architecture (such as class loading, byte code verification, preparation and reference resolution). With reason, the formalisation has focused on the core of the language, but garbage collection and API's such as RMI have almost been ignored by the community. In the future, we would expect general frameworks including all components of the architecture and all aspects of the language to emerge.

A general note on Java and Java Card documentation is that sometimes high-level concepts are expressed in low-level terms. For example, the Java security model talks about stack inspection, and the Java Card security model discusses which byte codes access objects. Java programmers should be able to understand all relevant issues in Java terms. We believe that with an appropriate formalisation of Java to hand, explanations in terms of implementation level concepts could be avoided.

Recent work on static analyses for Java optimising compilers has highlighted the need for techniques to analyse programs with dynamically loadable components. We expect this topic to be the subject of further investigation as component-based systems and distributed development become more prevalent.

Most formalisations are still written by hand and properties are still proved on paper. While such a methodology has successfully identified flaws in the Java specification or its implementation, we do not believe that manual methods scale up reliably to a complete formalisation of Java. There is a need for machine-readable formalisation, which can be analysed and reused by researchers. Furthermore, clarity and conciseness will make formalisations more accessible to systems designers and implementors, who may be able to give feedback. In particular, obtaining the highest level of certification for Java Card requires a complete formalisation down to a proof of implementation correctness. Such tasks need automated tools, and endeavors such as the LOOP projects are a step in the right direction. To be successful, formal specification, validation and provably correct implementation should be considered as a whole, rather than in separation; additionally, modular formalisation and proof techniques would help facilitate the separation of work between various research teams.

Formalising Java safety is a pre-requirement to building safe and secure applications. We see the need for (semi-)automatic tools that help programmers to understand the behaviour of their programs in terms of the permissions required to control the access of data; such tools will have to work at the source code level and with dynamically loaded libraries in order to be useful. Equational reasoning on programs, escape analysis, models of permissions and access control will be required. Additionally, security-specific aspects, which we have not surveyed in this paper, such as encryption and auditing, will also have to be addressed.

ACKNOWLEDGMENTS

The help and comments of Egon Börger, Luc Bougé, Bart Jacobs, Robert Stärk and the anonymous referees is gratefully acknowledged.

REFERENCES

Abadi, M. 1999. Protection in Programming-Language translations. In *Secure Internet Programming: Security Issues for Mobile and Distributed Objects, LNCS 1603*, pp. 19–34. Springer-Verlag, Berlin.

Abadi, M., Burrows, M., Lampson, B., and Plotkin, G. 1993. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems 15*, 4 (Sep), 706–734.

Ábrahám-Mumm, E. and de Boer, F. S. 2000. Proof-Outlines for threads in Java. In C. Palamidessi Ed., *11th Concurrency Theory (CONCUR), LNCS 1877* (University Park, Pennsylvania, Aug 2000), pp. 229–242. Springer-Verlag, Berlin.

Abrial, J.-R. 1996. *The B-Book: Assigning Programs to Meanings.* Cambridge Univ. Press, UK.

Agesen, O., Detlefs, D., and Moss, J. E. B. 1998. Garbage collection and local variable Type-Precision and liveness in Java virtual machines. In *Programming Language Design and Implementation (PLDI)* (Montreal, Canada, Jun 1998), pp. 269–279. ACM press, New York.

Agesen, O., Freund, S. N., and Mitchell, J. C. 1997. Adding type parameterization to the Java(TM) language. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pp. 49–65. ACM SIGPLAN NOTICES, 32(10).

Ahamad, M., Bazzi, R., John, R., Kohli, P., and Neiger, G. 1993. The power of processor consistency. In *5th Annual Symp. on Parallel Algorithms and Architectures (SPAA)* (Velen, Germany, Jun 1993), pp. 251–260. ACM, New York.

Aldrich, J., Chambers, C., Sirer, E., and Eggers, S. 1999. Static analyses for eliminating unnecessary synchronization from Java programs. In A. Cortesi and G. Filè Eds., *6th Static Analysis Symp, (SAS), LNCS 1694* (Venice, Italy, Sep 1999), pp. 19–38. Springer-Verlag, Berlin.

Alves-Foss, J. and Lam, F. S. 1999. Dynamic denotational semantics of Java. In J. Alves-Foss Ed., *Formal Syntax and Semantics of Java, LNCS 1523*, pp. 201–240. Springer-Verlag, Berlin.

Anderson, T. and Witty, R. W. 1978. Safe programming. *BIT 18*, 1–8.

Antoniu, G., Bougé, L., Hatcher, P., MacBeth, M., McGuigan, K., and Namyst, R. 2000. Compiling multithreaded Java bytecode for distributed execution. In A. Bode, T. Ludwig, W. Karl, and R. Wismüller Eds., *6th Int. Euro-Par 2000 - Parallel Processing Conf, LNCS 1900* (Munich, Germany, Aug 2000), pp. 1039–1052. Springer-Verlag, Berlin. `www.ens-lyon.fr/ ~bouge/ Biblio/ Antoniu/ AntBouHatBetGuiNam00EuroPar.ps.gz`.

Aridor, Y., Factor, M., and Teperman, A. 1999. cJVM: A single system image of a JVM on a cluster. In *Parallel Processing (ICPP'99)* (Wakamatsu, Japan, Sep 1999), pp. 21–24. IEEE Computer Society Press, Los Alamitos.

Aridor, Y., Factor, M., Teperman, A., Eilam, T., and Schuster, A. 2000. A high performance cluster JVM presenting a pure single system image. In *Conf. Java Grande* (San Francisco, California, Jun 2000), pp. 168–177. ACM, New York.

Attali, I., Caromel, D., Courbis, C., Henrio, L., and Nilsson, H. 2000. Smart tools for Java cards. In J. Domingo-Ferrer, D. Chan, and A. Watson Eds., *4th Int. IFIP wg 8.8 Conf. Smart card research and advanced application (CARDIS)* (Bristol, UK, Sep 2000), pp. 155–174. Kluwer Academic Publishers, Boston.

Attali, I., Caromel, D., and Russo, M. 1998. A formal executable semantics for Java. In *OOPSLA'98 Workshop on Formal Underpinnings of Java (FUJ)* (Vancouver, Canada, Nov 1998), pp. Paper 11. Dept. of Computing, Imperial College London. `www-dse.doc.ic.ac.uk/ ~sue/ oopsla/ cfp.html`.

Balfanz, D. and Felten, E. W. 1997. A Java filter. Technical Report 567-97 (Sep), Dept. of Compt. Science, Princeton Univ.

Basin, D., Friedrichs, S., Posegga, J., and Vogt, H. 1999. Java bytecode verification using model checking. In R. Alur and T. Henzinger Eds., *11th Computer Aided Verification (CAV), LNCS 1633* (New Brunswick, NJ, 1999), pp. 491–494. Springer-Verlag, Berlin.

Bertelsen, P. 1997. Semantics of Java byte code. Technical report (Mar), Technical Univ. of Denmark. `www.dina.kvl.dk/ ~pmb/`.

Bertelsen, P. 2000. Dynamic semantics of Java byte code. *Future Generation Computer Systems 16*, 7 (May), 841–850.

Bertelsen, P. and Anderson, S. 1996. The semantics of a core language derived from Java. Technical report (Sep), Technical Univ. of Denmark. `www.dina.kvl.dk/ ~pmb/`.

Bieber, P., Cazin, J., Wiels, V., Zanon, G., Girard, P., and Lanet, J.-L. 1999. Electronic purse applet certification. In S. Schneider and P. Ryan Eds., *Workshop on secure architectures and information flow* (Royal Holloway, London, Dec 1999). Electronics Notes in Theoretical Computer Science, 32.

Bigliardi, G. and Laneve, C. 2000. A type system for JVM threads. Technical Report UBLCS-2000-06 (Jun), University of Bologna.

Bogda, J. and Hölzle, U. 1999. Removing unnecessary synchronizations in Java. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)* (Denver, Colorado, Dec 1999), pp. 35–46. ACM-Press, New York.

Börger, E. and Schulte, W. 1998. Defining the Java virtual machine as platform for provably correct Java compilation. In L. Brim, J. Gruska, and J. Zlatuska Eds., *23rd Int. Symp. Mathematical Foundations of Computer Science (MFCS) LNCS 1450* (Brno, Czech Republic, Aug 1998), pp. 17–35. Springer-Verlag, Berlin.

Börger, E. and Schulte, W. 1999a. Initialization problems for Java. *Software Concepts and Tools 20*, 4, 175–179.

Börger, E. and Schulte, W. 1999b. A programmer friendly module definition of the semantics of Java. In J. Alves-Foss Ed., *Formal Syntax and Semantics of Java, LNCS 1523*, pp. 353–404. Springer-Verlag, Berlin.

Börger, E. and Schulte, W. 2000a. Modular design for the Java virtual machine architecture. In E. Börger Ed., *Architecture Design and Validation Methods*, pp. 297–357. Springer-Verlag, Berlin.

Börger, E. and Schulte, W. 2000b. A practical method for specification and analysis of exception handling – a Java/JVM case study. *IEEE Transactions on software engineering 26*, 9 (Sep), 872–887.

Borras, P., Clément, D., Despeyroux, T., Incerpi, J., Kahn, G., Lang, B., and Pascual, V. 1988. Centaur: the system. In *Third Annual Symp. on Software Development Environments (SDE3)* (Boston, USA, 1988), pp. 14–24. ACM, New York.

Bracha, G. 1999. *A Critique of* Security and Dynamic Loading in Java: A Formalisation. Sun Java Software. `http:// java.sun.com/ people/ gbracha/ critique-jmt.html`.

Bracha, G., Odersky, M., Stoutamire, D., and Wadler, P. 1998. GJ: Extending the Java programming language with type parameters. Technical report (Mar), Bell Labs, Lucent Technologies. `http:// cm.bell-labs.com/ cm/ cs/ who/ wadler/ pizza/ gj/`.

Cartwright, R. and Jr., G. L. S. 1998. Compatible genericity with run-time types for the Java programming language. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)* (Vancouver, Canada, Nov 1998), pp. 201–215. ACM press, New York.

Casset, L. and Lanet, J.-L. 1999. A formal specification of the Java bytecode semantics using the B method. Technischer Bericht 251 (Jun), Fernuniversität Hagen, Lisbon, Portugal.

Cenciarelli, P. 1999. Towards a modular denotational semantics of Java. Technischer Bericht 251 (Jun), Fernuniversität Hagen, Lisbon, Portugal.

Cenciarelli, P., Knapp, A., Reus, B., and Wirsing, M. 1999. An event based structural operational semantics of multi threaded Java. In J. Alves-Foss Ed., *Formal Syntax and Semantics of Java, LNCS 1523*, pp. 157–200. Springer-Verlag, Berlin.

Chen, X. and Allan, V. 1998. MultiJav: A distributed shared memory system based on multiple Java virtual machines. In *Conference on Parallel and Distributed Processing Techniques and Applications (PDTA'98)* (Las Vegas, Nevada, Jun 1998).

Chen, Z. 2000. *Java Card Technology for Smart Cards: Architecture and programmer's guide*. Addison Wesley, Reading, Massachusetts.

Choi, J.-D., Gupta, M., Serrano, M., Sreedhar, V., and Midkiff, S. 1999. Escape analysis for Java. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)* (Denver, Colorado, Dec 1999), pp. 1–19. ACM Press, New York.

Coglio, A. and Goldberg, A. 2000. *Type Safety in the JVM: Some Problems in JDK 1.2.2 and Proposed Solutions*. Kestrel Institute, 3260 Hillview Avenue, Palo Alto, CA 94304. www.kestrel.edu/ java..

Coglio, A., Goldberg, A., and Qian, Z. 1998. Toward a Provably-Correct implementation of the JVM bytecode verifier. In *OOPSLA'98 Workshop on Formal Underpinnings of Java (FUJ)* (Vancouver, Canada, Nov 1998), pp. Paper 6. Dept. of Computing, Imperial College London. www-dse.doc.ic.ac.uk/ ~sue/ oopsla/ cfp.html.

Cohen, R. M. 1997. The defensive Java virtual machine specification version 0.5. Technical report (May), Computational Logic Inc, Austin, Texas. www.cli.com/.

Cohen, R. M. 1998. Formal underpinnings of Java: Some requirements. In *OOPSLA'98 Workshop on Formal Underpinnings of Java (FUJ)* (Vancouver, Canada, Nov 1998), pp. Paper 1. Dept. of Computing, Imperial College London. www-dse.doc.ic.ac.uk/ ~sue/ oopsla/ cfp.html.

Colby, C., Lee, P., Necula, G. C., Blau, F., Cline, K., and Plesko, M. 2000. A certifying compiler for Java. In *Programming Language Design and Implementation (PLDI)* (Vancouver, British Columbia, Canada, Jun 2000), pp. 95–107. ACM, New York.

Corbett, J. C., Dwyer, M. B., Hatcliff, J., Laubach, S., Pasareanu, C. S., Robby, and Zheng, H. 2000. Bandera: extracting finite-state models from Java source code. In *22nd international conference on on Software engineering* (Limerick Ireland, Jun 2000), pp. 439–448. ACM, New York.

Coscia, E. and Reggio, G. 1998a. An operational semantics for Java. Technical report (Nov), DISI, Univ. of Genova, Italy. www.disi.unige.it/ person/ CosciaE/ publications.html.

Coscia, E. and Reggio, G. 1998b. A proposal for a semantics of a subset of Multi-Threaded "good" Java programs. In *OOPSLA'98 Workshop on Formal Underpinnings of Java (FUJ)* (Vancouver, Canada, Nov 1998), pp. Paper 10. Dept. of Computing, Imperial College London. www-dse.doc.ic.ac.uk/ ~sue/ oopsla/ cfp.html.

Crary, K. and Weirich, S. 2000. Resource bound certification. In *27th Principles of programming languages (POPL)* (Boston, Massachusetts, Jan 2000), pp. 184–198. ACM, New York. www.cs.cmu.edu/ afs/ cs/ user/ crary/ www/ papers/.

Dean, D. 1997. The security of static typing with dynamic linking. In *4th Computer and Communications Security* (Zurich, Switzerland, Apr 1997), pp. 18–27. ACM, New york.

Dean, D., Felten, E. W., and Wallach, D. S. 1996. Java security: From HotJava to netscape and beyond. In *Symp. on Security and privacy* (Oakland, California, May 1996), pp. 190–200. IEEE Computer Society Press, Los Alamitos, California.

Demartini, C., Iosif, R., and Sisto, R. 1998. Modeling and validation of Java multithreading applications using SPIN. In *4th Spin workshop* (Paris, France, Nov 1998). http:// netlib.bell-labs.com/ netlib/ spin/ ws98/.

Denney, E. and Jensen, T. 2000. Correctness of Java card method lookup via logical relations. In E. Smolka Ed., *9th European Symp. on programming (ESOP), LNCS 1782* (Berlin, West Germany, Mar 2000), pp. 104–118. Springer-Verlag, Berlin.

Detlefs, D. L., Leino, K. R. M., Nelson, G., and Saxe, J. B. 1998. Extended static checking. SRC Research report 159 (Dec), Compaq Systems Research Center, Palo Alto, California.

Diehl, S. 1998. A formal introduction to the compilation of Java. *Software—practice and experience 28*, 3 (Mar), 297–327.

Drossopoulou, S. 2000. Towards an abstract model of Java dynamic linking and verification. In *ACM SIGPLAN Workshop on Types in Compilation (TIC)* (Montreal, Canada, Sep 2000), pp. Paper 19. Computer Science Department, Carnegie Mellon University.

Drossopoulou, S. and Eisenbach, S. 1997. Java is type safe – probably. In M. Aksit and S. Matsuoka Eds., *11th European Conference on Object Oriented Programming, ECOOP, LNCS 1241* (Jyväskylä, Finland, Jun 1997), pp. 389–418. Springer Verlag, Berlin.

Drossopoulou, S. and Eisenbach, S. 1999. Describing the semantics of Java and proving type soundness. In J. Alves-Foss Ed., *Formal Syntax and Semantics of Java, LNCS 1523*, pp. 41–82. Springer-Verlag, Berlin.

Drossopoulou, S., Eisenbach, S., and Khurshid, S. 1999. Is the Java type system sound? *Theory and practice of object systems 5*, 1, 3–24.

Drossopoulou, S., Eisenbach, S., and Wragg, D. 1999. A fragment calculus - towards a model of separate compilation, linking and binary compatibility. In *Logic in Computer Science (LICS)* (Trento, Italy, Jul 1999), pp. 147–156. IEEE Computer Society Press.

Drossopoulou, S. and Valkevych, T. 2000. *Java Exceptions Throw no Surprises.* Department of Computing, Imperial College, London.

Drossopoulou, S., Wragg, D., and Eisenbach, S. 1998. What is Java binary compatibility? In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)* (Vancouver, Canada, Oct 1998), pp. 341–361. Sigplan Notices, 33(10).

Felleisen, M. and Hieb, R. 1992. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science 2*, 4, 235–271.

Flanagan, C. and Freund, S. 2000. Type-based race detection for Java. In *Programming Language Design and Implementation (PLDI)* (Vancouver, Canada, Jun 2000), pp. 219–232. ACM-Press, New York.

Flatt, M., Krisnamurthi, S., and Felleisen, M. 1999. A programmer's reduction semantics for classes and mixins. In J. Alves-Foss Ed., *Formal Syntax and Semantics of Java, LNCS 1523*, pp. 241–270. Springer-Verlag, Berlin.

Fokkink, W. J. and Verhoef, C. 1998. A conservative look at operational semantics with variable binding. *Information and Computation 146*, 1, 24–54. `http:// adam.wins.uva.nl/ ~x/ conimex.ps`.

Fong, P. W. L. and Cameron, R. D. 1998. Proof linking: An architecture for modular verification of dynamically-linked mobile code. In *6th SIGSOFT Int. Symp. on the Foundations of Software Engineering* (Orlando, Florida, Nov 1998), pp. 222–230. ACM press, New York.

Freund, S. N. 1998. The costs and benefits of Java bytecode subroutines. In *OOPSLA'98 Workshop on Formal Underpinnings of Java (FUJ)* (Vancouver, Canada, Nov 1998), pp. Paper 2. Dept. of Computing, Imperial College London. `www-dse.doc.ic.ac.uk/ ~sue/ oopsla/ cfp.html`.

Freund, S. N. and Mitchell, J. C. 1998. A type system for object initialization in the Java bytecode language. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)* (Vancouver, Canada, Oct 1998), pp. 310–328. ACM Press, New York.

Freund, S. N. and Mitchell, J. C. 1999a. A formal framework for the Java bytecode language and verifier. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)* (Denver, Colorado, Dec 1999), pp. 147–166. ACM press, New York.

Freund, S. N. and Mitchell, J. C. 1999b. A type system for object initialization in the Java bytecode language. *ACM Transactions on Programming Languages and Systems 21*, 6 (Nov), 1196–1250.

Fritzinger, J. S. and Mueller, M. 1996. *Java Security.* Sun Micro systems Inc, Mountain View, California.

Gagnon, E. and Hendren, L. 1999. Intra-procedural inference of static types. Technical Report 1999-1 (Mar), Sable Group, McGill University, Montréal, Canada.

www.sable.mcgill.ca.

Glesner, S. and Zimmermann, W. 1998. Using many-sorted natural semantics to specify and generate semantic analysis. In *TC2 WG2.4 Working Conference on Systems Implementation 2000: Languages, Methods and Tools*, pp. 249–62. Chapman & Hall, London.

Goldberg, A. 1998. A specification of Java loading and bytecode verification. In *5th Conf. Computer and Communications Security* (San Francisco, California, Nov 1998), pp. 49–58. ACM Press, New York. www.kestrel.edu/ HTML/ people/ goldberg/.

Gong, L. 1998. Secure Java class loading. *IEEE-Internet-Computing 2*, 6 (Nov), 56–61.

Gontmakher, A. and Schuster, A. 2000. Java consistency: Non-operational characterizations for Java memory behavior. *ACM Transactions on Computer Systems 18*, 4 (Nov), 333–386.

Goré, R., Posegga, J., Slater, A., and Vogt, H. 1998. cardTAP: The first theorem prover on a smart card. In *15th Automated Deduction (CADE), LNCS 1421* (Lindau, Germany, Jul 1998), pp. 47–50. Springer Verlag, Berlin.

Gosling, J., Joy, B., and Steele, G. 1996. *The Java Language Specification*. Addison Wesley, Reading, Massachusetts.

Gosling, J., Joy, B., Steele, G., and Bracha, G. 2000. *The Java Language Specification* (second ed.). Addison Wesley, Reading, Massachusetts, Reading, Massachusetts.

Gurevich, Y. 1995. Evolving algebras 1993: Lipari guide. In E. Börger Ed., *Specification and Validation Methods*, pp. 9–36. Oxford University Press.

Hagiya, M. and Tozawa, A. 1998. On a new method for dataflow analysis of Java virtual machine subroutines. In G. Levi Ed., *Int. Static Analysis Symp. (SAS), LNCS 1503* (Pisa, Italy, Sep 1998), pp. 17–32. Springer-Verlag, Berlin.

Hartel, P. H. 1999. LETOS – a lightweight execution tool for operational semantics. *Software—practice and experience 29*, 15 (Sep), 1379–1416. www.ecs.soton.ac.uk/ ~phh/ letos.html.

Hartel, P. H., Butler, M. J., and Levy, M. 1999. The operational semantics of a Java secure processor. In J. Alves-Foss Ed., *Formal Syntax and Semantics of Java, LNCS 1523*, pp. 313–352. Springer-Verlag, Berlin. www.dsse.ecs.soton.ac.uk/ techreports/ 98-1.html.

Hartel, P. H. and de Jong Frz, E. K. 2000. A programming and a modelling perspective on the evaluation of Java card implementations. In I. Attali and T. Jensen Eds., *1st Java on Smart Cards: Programming and Security (e-Smart), LNCS 2041* (Cannes, France, Sep 2000), pp. 52–72. Springer-Verlag, Berlin. www.dsse.ecs.soton.ac.uk/ techreports/ 2000-8.html.

Havelund, K. and Pressburger, T. 2000. Model checking Java programs using pathfinder. *Software Tools for Technology Transfer 2*, 4, 366–381. http:// ase.arc.nasa.gov/ havelund/.

Hayes, B. 1992. Finalization in the collector interface. In Y. Bekkers and J. Cohen Eds., *Memory Management (IWMM), LNCS 637* (St. Malo, France, Sep 1992), pp. 277–298. Springer-Verlag, Berlin.

Hilderink, G., Broeking, J., Vervoort, W., and Bakkers, A. 1997. Communicating Java threads. In *20th World Occam and Transputer User Group Technical Meeting* (Enschede The Netherlands, Apr 1997), pp. 48–76. IOS Press, Amsterdam. www.rt.el.utwente.nl/ javapp/.

Holzmann, G. J. 1997. The model checker SPIN. *IEEE Transactions on software engineering 23*, 5, 279–295. http:// cm.bell-labs.com/ cm/ cs/ who/ gerard/.

Huisman, M. 2001. *Reasoning about Java progams in higher order logic with PVS and Isabelle*. Ph. D. thesis, Univ. of Nijmegen, The Netherlands.

Huisman, M., Jacobs, B., and van den Berg, J. 2001. A case study in class library verification: Java's vector class. *Software tools for technology transfer*, to appear.

Hummel, J., Azavedo, A., Kolson, D., and Nicolau, A. 1997. Annotating the Java bytecodes in support of optimization. *Concurrency: Practice and Experience 9*, 11 (Nov), 1003–1016. www.eecs.uic.edu/ ~jhummel/ papers.html.

IGARASHI, A. AND PIERCE, B. 2000. On inner classes. In E. BERTINO Ed., *14th European Conference on Object Oriented Programming, ECOOP, LNCS 1850* (Sophia Antipolis, France, Jun 2000), pp. 129–153. Springer-Verlag, Berlin. `http://web.yl.is.s.u-tokyo.ac.jp/ ~igarashi/`.

IGARISHI, A., PIERCE, B., AND WADLER, P. 1999. Featherweight Java: A minimal core calculus for Java and GJ. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)* (Denver, Colorado, Oct 1999), pp. 132–146. ACM Press, New York.

ISO/IEC. 1995. *7816-4:1995 Information technology–Identification cards–Integrated circuit(s) cards with contacts part4: Inter-Industry commands for interchange*. Int. Standards Organization.

ITSEC. 1993. *Evaluation criteria for IT security – part 3: Assurance of IT systems* (version 1.2 ed.). INFOSEC central office, Brussels, Belgium.

JACOBS, B. 2001. A formalisation of Java's exception mechanism. In D. SANDS Ed., *10th European Symp. on programming (ESOP), LNCS 2028* (Genova, Italy, Apr 2001), pp. 284–301. Springer-Verlag, Berlin.

JACOBS, B. AND POLL, E. 2000. A monad for basic Java semantics. In T. RUS Ed., *Algebraic Methodology and Software Technology (AMAST), LNCS 1816*, pp. 150–164. Springer-Verlag.

JACOBS, B. AND POLL, E. 2001. A logic for the Java modeling language JML. In H. HUSSMANN Ed., *4th Fundamental Approaches to Software Engineering (FASE), LNCS 2029* (Genova, Italy, Apr 2001), pp. 284–299. Springer-Verlag, Berlin.

JACOBS, B., VAN DEN BERG, J., HUISMAN, M., VAN BERKUM, M., HENSEL, U., AND TEWS, H. 1998. Reasoning about classes in Java (preliminary report). In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)* (Vancouver, Canada, Oct 1998), pp. 329–340. ACM Press, New York.

JENSEN, T., METAYER, D. L., AND THORN, T. 1998. Security and dynamic class loading in Java: A formalisation. In *Computer Languages*, pp. 4–15. IEEE Comput. Soc. Press, Los Alamitos, California.

JENSEN, T., METAYER, D. L., AND THORN, T. 1999. Verification of control flow based security properties. In *Symp. on Security and Privacy* (Oakland, California, May 1999), pp. 89–103. IEEE Comput. Soc, Los Alamitos, California.

JONES, M. 1998. The functions of Java bytecode. In *OOPSLA'98 Workshop on Formal Underpinnings of Java (FUJ)* (Vancouver, Canada, Nov 1998), pp. Paper 3. Dept. of Computing, Imperial College London. `www-dse.doc.ic.ac.uk/ ~sue/ oopsla/ cfp.html`.

KASSAB, L. AND GREENWALD, S. 1998. Towards formalizing the Java security architecture in JDK 1.2. In J.-J. QUISQUATER, Y. DESWARTE, C. MEADOWS, AND D. GOLLMANN Eds., *European Symp. on Research in Computer Security (ESORICS), LNCS 1485* (Louvain-la-Neuve, Belgium, Sep 1998), pp. 191–207. Springer-Verlag, Berlin.

KAUFMANN, M. AND MOORE, J. S. 1996. ACL2: An industrial strength version of nqthm. In *11th Annual Conf. on Computer Assurance (COMPASS)* (Gaithersburg, MD, Jun 1996), pp. 23–34. IEEE Computer Society Press, Los Alamitos, California.

KISTLER, T. AND FRANZ, M. 1996. A Tree-Based alternative to Java Byte-Codes. Technical Report 96-58 (Dec), Depart. of Information and Computer Science, Univ. of California, Irvine.

KLEIN, G. AND NIPKOW, T. 2000. Verified lightweight bytecode verification. In S. DROSSOPOULKOU, S. EISENBACH, B. JACOBS, G. T. LEAVENS, P. MÜLLER, AND A. POETZSCH-HEFFTER Eds., *ECOOP'2000 Workshop on Formal Techniques for Java Programs* (Sophia Antipolis, France, Jun 2000), pp. 35–42. Fernuniversität Hagen.

KNOBLOCK, T. B. AND REHOF, J. 2000. Type elaboration and subtype completion for Java bytecode. In *27th Principles of programming languages (POPL)* (Boston, Massachusetts, Jan 2000), pp. 228–242. ACM, New York.

KOZEN, D. 1998. Efficient code certification. Tech. Report 98-1661 (Jan), Dept. of Comp. Sci, Cornell Univ, Ithaca, New York. `www.cs.cornell.edu/ kozen/ papers/ cert.ps`.

LANET, J.-L. 2000. Are smart cards the ideal domain for applying formal methods? In

International Conference of Z and B Users (ZB), LNCS 1878 (York, UK, Sep 2000), pp. 363–374. Springer Verlag, Berlin.

Lanet, J.-L. and Requet, A. 1998. Formal proof of smart card applets correctness. In J.-J. Quisquater and B. Schneier Eds., *3rd Smart card research and advanced application (CARDIS), LNCS 1820* (Louvain la Neuve, Belgium, Sep 1998), pp. 85–97. Springer-Verlag, Berlin.

Leavens, G. T., Baker, A. L., and Ruby, C. 1999. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds Eds., *Behavioral Specifications of Business and Systems*, pp. 175–188. Kluwer Academic Publishers, Boston/Dordrecht/London.

Leino, K. R. M., Saxe, J. B., and Stata, R. 1999. Checking Java programs via guarded commands. SRC Research report 1999-002 (May), Compaq Systems Research Center, Palo Alto, California.

Liang, S. and Bracha, G. 1998. Dynamic class loading in the Java virtual machine. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)* (Vancouver, Canada, Oct 1998), pp. 36–44. Sigplan Notices, 33(10).

Lindholm, T. and Yellin, F. 1996. *The Java Virtual Machine Specification*. Addison Wesley, Reading, Massachusetts.

Maessen, J., Arvind, and Shen, X. 2000. Improving the Java memory model using CRF. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)* (Minneapolis, Minnesota, Oct 2000), pp. 1–12. ACM, New York.

Malkhi, D. and Reiter, M. K. 2000. Secure execution of Java applets using a remote playground. *IEEE Transactions on Software Engineering 26*, 12 (Dec), 1197 –1209. `www.research.att.com/ ~dalia/`.

Manson, J. and Pugh, W. 2001. Semantics of multithreaded Java. Technical report (Jan), Dept. of Computer Science, University of Maryland.

McGraw, G. and Felten, E. W. 1999. *Securing Java: Getting down to business with mobile code* (second ed.). John Wiley & Sons, Chichester, UK.

McMillan, K. L. 1993. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston.

Montgomery, M. and Krishna, K. 1999. Secure object sharing in Java card. In *USENIX Workshop on Smartcard Technology (Smartcard '99)* (Chicago, Illinois, 1999), pp. 119–127. USENIX Assoc, Berkeley, California.

Moore, J. S. 1999. Proving theorems about Java-like byte code. In E.-R. Olderog and B. Steffen Eds., *Correct System Design – Recent Insights and Advances, LNCS 1710*, pp. 139–162. Springer-Verlag, Berlin.

Moreira, J. E., Midkiff, S. P., and Gupta, M. 2000. From flop to megaflops: Java for technical computing. *ACM Transactions on Programming Languages and Systems 22*, 2 (Mar), 265–295.

Morrisett, G., Felleisen, M., and Harper, R. 1995. Abstract models of memory management. Technical report CMU-CS-95-110 (Jan), School of Comp. Sci, Carnegie Mellon Univ.

Morrisett, G., Walker, D., Crary, K., and Glew, N. 1999. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems 21*, 3 (May), 528–569.

Motré, S. 2000. Formal model and implementation of the Java card dynamic security policy. In *Approches Formelles dans l'Assistance au Développement de Logiciels - AFADL'2000* (Grenoble, France, Jan 2000). `http:// www-lsr.imag.fr/ afadl`.

Myers, A., Bank, J., and Liskov, B. 1997. Parametrized types for Java. In *24th Principles of programming languages (POPL)* (Paris, France, Jan 1997), pp. 132–145. ACM-Press, New York.

NCSC. 1985. *Trusted Computer System Evaluation Criteria (Orange Book)*. U. S. Dept. of Defense, National Computer Security Center. `www.boran.com/ security/ tcsec.html`.

Necula, G. C. 1997. Proof-carrying code. In *24th Principles of programming languages (POPL)* (Paris, France, Jan 1997), pp. 106–119. ACM, New York.

Necula, G. C. and Lee, P. 1998a. The design and implementation of a certifying compiler. In *Programming Language Design and Implementation (PLDI)* (Montreal, Canada, Jun 1998), pp. 333–344. ACM, New York.

Necula, G. C. and Lee, P. 1998b. Efficient representation and validation of proofs. In *13th Logic in Computer Science (LICS)* (Indianapolis, Indiana, Jun 1998), pp. 93–104. IEEE Computer Society Press. `www.cs.cmu.edu/ ~necula/ lics98.ps.gz`.

Necula, G. C. and Lee, P. 1998c. Safe, untrusted agents using Proof-Carrying code. In G. Vigna Ed., *Mobile Agents and Security, LNCS 1419*, pp. 61–91. Springer-Verlag, Berlin.

Necula, G. C. and Rahul, S. P. 2001. Oracle-Based checking of untrusted software. In *28th Principles of programming languages (POPL)* (London, UK, Jan 2001), pp. 142–154. ACM, New York.

Nielson, H. R. and Nielson, F. 1991. *Semantics with applications: A formal introduction.* John Wiley & Sons, Chichester, UK.

Nilsen, K. 1998. picoPERC: a small-footprint dialect of Java. *Dr.-Dobb's Journal 23*, 3 (Mar), 50–54.

Nipkow, T. and von Oheimb, D. 1998. Java$_{\ell ight}$ is Type-Safe — definitely. In *25th Principles of programming languages (POPL)* (San Diego, California, Jan 1998), pp. 161–170. ACM, New York.

Nipkow, T., von Oheimb, D., and Pusch, C. 2000. μJava: Embedding a programming language in a theorem prover. In F. L. Bauer and R. Steinbrüggen Eds., *Foundations of Secure Computation. Proc. Int. Summer School Marktoberdorf*, pp. 117–144. IOS Press, Amsterdam.

NIST. 1999. *Common Criteria for Information Technology Security Evaluation.* U. S. Dept. of Commerce, National Bureau of Standards and Technology. `http:// csrc.nist.gov/ cc/`.

O'Callahan, R. 1999. A simple, comprehensive type system for Java bytecode subroutines. In *26th Principles of programming languages (POPL)* (San Antonio, Texas, Jan 1999), pp. 70–78. ACM, New York.

Odersky, M. and Wadler, P. 1997. Pizza into Java: Translating theory into practice. In *24th Principles of programming languages (POPL)* (Paris, France, Jan 1997), pp. 146–159. ACM, New York.

Oestreicher, M. 1999. Transactions in Java card. In *15th Annual Computer Security Applications Conference (ACSAC)* (Phoenix, Arizona, Dec 1999), pp. 291–298. IEEE Comput. Soc, Los Alamitos, California. `www.acsac.org/ 1999/ abstracts/ thu-b-1500-marcus.html`.

Oestreicher, M. and Krishna, K. 1999. Object lifetimes in Java card. In *USENIX Workshop on Smartcard Technology (Smartcard '99)* (Chicago, Illinois, 1999), pp. 129–37. USENIX Assoc, Berkeley, California.

Owre, S., Rushby, J., Shankar, N., and von Henke, F. 1995. Formal verification for Fault-Tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering 21*, 2 (Feb), 107–125.

Paulson, L. C. 1994. *Isabelle: a generic theorem prover, LNCS 828.* Springer-Verlag, New York.

Poetzsch-Heffter, A. and Muller, P. 1999. A programming logic for sequential Java. In *8th European Symp. on programming (ESOP), LNCS 1576*, pp. 162–176. Springer-Verlag, Berlin.

Poll, E., van den Berg, J., and Jacobs, B. 2000. Specification of the JavaCard API in JML. In J. Domingo-Ferrer and A. Watson Eds., *Fourth Smart Card Research and Advanced Application Conference (CARDIS)* (Bristol, UK, Sep 2000), pp. 135–154. Kluwer Academic Publishers, Boston/Dordrecht/London.

Poll, E., van den Berg, J., and Jacobs, B. 2001. Formal specification of the JavaCard API in JML: the APDU class. *Computer Networks 36*, 4 (Jul), 407–421.

Posegga, J. and Vogt, H. 1998. Byte code verification for Java smart cards based on model checking. In J.-J. Quisquater, Y. Deswarte, C. Meadows, and D. Gollmann Eds., *European Symp. on Research in Computer Security (ESORICS), LNCS 1485*

(Louvain-la-Neuve, Belgium, Sep 1998), pp. 175–190. Springer-Verlag, Berlin.

Pugh, W. 2000. The Java memory model is fatally flawed. *Concurrency: Practice and Experience 12*, 1, 1–11.

Pusch, C. 1998. Formalizing the Java virtual machine in isabelle/HOL. Technical report TUM-I9816, Institut für Informatik, Technische Univ. München.

Pusch, C. 1999. Proving the soundness of a Java bytecode verifier specification in isabelle/HOL. In W. Rance-Cleaveland Ed., *5th Tools and Algorithms for Construction and Analysis of Systems (TACAS), LNCS 1579* (Amsterdam, The Netherlands, 1999), pp. 89–103. Springer Verlag, Berlin. `www4.informatik.tu-muenchen.de/ ~pusch/ pubs/ TACAS99.html`.

Qian, Z. 1999a. A formal specification of Java(tm) virtual machine instructions objects, methods and subroutines. In J. Alves-Foss Ed., *Formal Syntax and Semantics of Java, LNCS 1523*, pp. 271–312. Springer-Verlag, Berlin.

Qian, Z. 1999b. *Standard Fixpoint Iteration for Java Bytecode Verification.* Kestrel Institute, Palo Alto, CA 94304.

Qian, Z., Goldberg, A., and Coglio, A. 2000. A formal specification of JavaTM class loading. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)* (Minneapolis, Minnesota, Oct 2000), pp. 325–336. ACM, New York.

Rémy, D. 1989. Records and variants as a natural extension of ML. In *16th Principles of programming languages (POPL)* (Austin, Texas, Jan 1989), pp. 77–88. ACM, New York.

Requet, A. 2000. A B model for ensuring soundness of the Java card virtual machine. In S. Gnesi, I. Schieferdecker, and A. Rennoch Eds., *5th International ERCIM Workshop on Formal Methods for Industrial Critical Systems (FMICS)* (Berlin, Mar 2000), pp. 29–26. GMD.

Rose, E. 1998. Towards secure bytecode verification on a Java card. Master's thesis, DIKU, Univ. of Copenhagen.

Rose, E. and Rose, K. H. 1998. Lightweight bytecode verification. In *OOPSLA'98 Workshop on Formal Underpinnings of Java (FUJ)* (Vancouver, Canada, Nov 1998), pp. Paper 7. Dept. of Computing, Imperial College London. `www-dse.doc.ic.ac.uk/ ~sue/ oopsla/ cfp.html`.

Saraswat, V. 1997. Java is not type-safe. Technical report (Aug), AT&T Research, Florham Park, New Jersey. `www.research.att.com/ ~vj/ bug.html`.

Serrano, M. 1995. Control flow analysis: a functional languages compilation paradigm. In *10th Symp. on Applied Computing* (Nashville, Tennessee, USA, Feb 1995), pp. 118–122. ACM press, New York.

Shin, I. and Mitchell, J. C. 1998. Java bytecode modification and applet security. Technical report, Comp. Sci Dept, Stanford Univ.

Skalka, C. and Smith, S. 2000. Static enforcement of security with types. In *5th SIGPLAN on on Functional programming (ICFP)* (Montreal, Canada, Sep 2000), pp. 34–45. ACM, New York.

Sreedhar, V. C., Burke, M., and Choi, J.-D. 2000. A framework for interprocedural analysis and optimization in the presence of dynamic class loading. In *Programming Language Design and Implementation (PLDI)* (Vancouver, British Columbia, Canada, Jun 2000), pp. 176–207. ACM press, New York.

Srinivas, Y. V. and Jullig, R. 1995. Specware: Formal support for composing software. In *Conf. Mathematics of Program Construction (MPCS), LNCS 947* (Kloster Irsee, Germany, Jul 1995), pp. 399–422. Springer-Verlag, Berlin.

Stärk, R. 1998. *Foundations of Java – Lecture Notes for Computer Science Students.* University of Fribourg, Switzerland. `www.inf.ethz.ch/ personal/ staerk/ java/`.

Stärk, R., Schmid, J., and Börger, E. 2001. *Java and the Java Virtual Machine: Definition, Verification, Validation.* Springer-Verlag, Berlin.

Stata, R. and Abadi, M. 1998. A type system for Java bytecode subroutines. In *25th Principles of programming languages (POPL)* (San Diego, California, Jan 1998), pp. 149–160. ACM, New York.

STATA, R. AND ABADI, M. 1999. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems 21*, 1, 90–37.

STEPHENSON, K. 1998. Towards an algebraic specification of the Java virtual machine. In B. MOLLER AND J. V. TUCKER Eds., *Prospects for hardware foundations. ESPRIT working group 8533. NADA - new hardware design methods survey chapters, LNCS 1546*, pp. 236–277. Springer-Verlag, Berlin.

STILES, G. S. 1998. Safe and verifiable design of multithreaded Java programs with CSP and FDR. In *OOPSLA'98 Workshop on Formal Underpinnings of Java (FUJ)* (Vancouver, Canada, Nov 1998), pp. Paper 12. Dept. of Computing, Imperial College London. `www-dse.doc.ic.ac.uk/ ~sue/ oopsla/ cfp.html`.

SUN. 1999. *The K Virtual Machine (KVM) – A white paper.* Sun Micro systems Inc, Mountain View, California. `http:// java.sun.com/ products/ kvm/`.

SUN. 2000. *Connected Limited Devicde Specification version 1.0, Java Platform 2 Micro Edition.* Sun Micro systems Inc, Palo Alto, California. `http:// java.sun.com/ aboutJava/ communityprocess/ final/ jsr030/ index.html`.

SYME, D. 1999. Proving Java type soundness. In J. ALVES-FOSS Ed., *Formal Syntax and Semantics of Java, LNCS 1523*, pp. 83–118. Springer-Verlag, Berlin.

TAIVALSAARI, A., BUSH, B., AND SIMON, D. 1999. The spotless system: Implementing a JavaTMSystem for the palm connected organizer. Technical report TR-99-73 (Feb), Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303 USA.

TALPIN, J.-P. AND JOUVELOT, P. 1992. Polymorphic type, region and effect inference. *J. Functional Programming 2*, 3 (Jul), 245–271.

THORN, T. 1997. Programming languages for mobile code. *ACM Computing Surveys 29*, 3 (Sep), 213–239.

TOZAWA, A. AND HAGIYA, M. 1999a. Careful analysis of type spoofing. In C. H. CAP Ed., *JIT'99 Java-Informations-Tage*, pp. 290–296. Informatik aktuell, Springer-Verlag.

TOZAWA, A. AND HAGIYA, M. 1999b. *Formalization and Analysis of Class Loading in Java.* Graduate School of Science, University of Tokyo.

VAN DEN BERG, J., HUISMAN, M., JACOBS, B., AND POLL, E. 2000. A type-theoretic memory model for verification of sequential Java programs. In D. BERT AND C. CHOPPY Eds., *Recent Trends in Algebraic Development Techniques, LNCS 1827*, pp. 1–21. Springer-Verlag, Berlin.

VAN DEN BERG, J. AND JACOBS, B. 2001. The LOOP compiler for Java and JML. In T. MARGARIA AND W. YI Eds., *7th Tools and algorithms for the construction and analysis of systems (TACAS), LNCS 2031* (Genova, Italy, Apr 2001), pp. 299–312. Springer-Verlag, Berlin.

VAN DEN BERG, J., JACOBS, B., AND POLL, E. 2000. Formal specification and verification of JavaCard's application identifier class. In I. ATTALI AND T. JENSEN Eds., *1st Java on Smart Cards: Programming and Security (e-Smart), LNCS 2041* (Cannes, France, Sep 2000), pp. 173–150. Springer-Verlag, Berlin.

VOLPANO, D. AND SMITH, G. 1998. Language issues in mobile program security. In G. VIGNA Ed., *Mobile agents and security, LNCS 1419*, pp. 25–43. Springer-Verlag, Berlin.

VON OHEIMB, D. 2000. Axiomatic semantics for Java$_{\ell ight}$. In S. DROSSOPOULKOU, S. EISENBACH, B. JACOBS, G. T. LEAVENS, P. MÜLLER, AND A. POETZSCH-HEFFTER Eds., *ECOOP'2000 Workshop on Formal Techniques for Java Programs* (Sophia Antipolis, France, Jun 2000), pp. 88–95. Fernuniversität Hagen. `http:// isabelle.in.tum.de/ Bali/ papers/ ECOOP00.html`.

VON OHEIMB, D. AND NIPKOW, T. 1999. Machine-Checking the Java specification: Proving type safety. In J. ALVES-FOSS Ed., *Formal Syntax and Semantics of Java, LNCS 1523*, pp. 119–156. Springer-Verlag, Berlin.

WADLER, P. L. 1990. Comprehending monads. In *Lisp and functional programming* (Nice, France, Jul 1990), pp. 61–78. ACM, New York.

WALLACE, C. 1997. The semantics of the Java programming language: Preliminary version. Technical Report CSE-TR-355-97, University of Michigan EECS Department.

WALLACH, D. S. 1999. *A new Approach to Mobile Code Security*. Ph. D. thesis, Priceton University.

WALLACH, D. S. AND FELTEN, E. W. 1998. Understanding Java stack inspection. In *Symp. on Security and Privacy* (Oakland, California, May 1998), pp. 52–63. IEEE Computer Society Press, Los Alamitos, California.

WAND, M. AND GUTTMAN, J. D. Eds. 1995. *VLISP: A Verified Implementation of Scheme*. Kluwer, Boston. `ftp:// ftp.ccs.neu.edu/ pub/ people/ wand/ vlisp/ lasc`.

WEBB, W. 1999. Embedded Java: An uncertain future. *Electrical Design News 44*, 10 (May), 89–96. `http:// 209.67.241.58/ reg/ 1999/ 051399/ 10df2.htm`.

WELCH, P. H. 1997. Java threads in light of occam/CSP. In A. BAKKERS Ed., *Parallel Programming and Java, WoTUG 20* (Twente, Netherlands, Apr 1997), pp. 282–309. Concurrent Systems Engineering Series, IOS Press, Amsterdam.

WHALEY, J. AND RINARD, M. 1999. Compositional pointer and escape analysis for Java programs. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)* (Denver, Colorado, Dec 1999), pp. 187–206. ACM Press, New York.

WRIGHT, A. K. 1994. *Practical Soft Typing*. Ph. D. thesis, Rice University, Houston, Texas.

YELLAND, P. 1999. A compositional account of the Java virtual machine. In *26th Principles of programming languages (POPL)* (San Antonio, Texas, Jan 1999), pp. 57–69. ACM, New York.