Formalising Java Safety – An overview

Pieter H. Hartel
phh@ecs.soton.ac.uk

Declarative Systems and Software Engineering Group
Technical Report DSSE-TR-2000-2
February 14 2000

Department of Electronics and Computer Science
University of Southampton
Highfield, Southampton SO17 1BJ, United Kingdom

Formalising Java Safety – An overview

Pieter H. Hartel *

February 14, 2000

Abstract

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.

We have tried to avoid technical detail, and to focus on the bigger issues. The interested reader may wish to consult some of the many papers that we refer to fill in the details.

1 Introduction

Java offers interesting possibilities for building flexible and portable smart card systems. However, many design and implementation problems have been, and are being reported in the literature with respect to Java's safety [38].

Java is a safe programming language in the sense that Java programs are type safe and memory safe. The two main features that bring type and memory safety are firstly that Java does not offer pointer arithmetic; instead Java offers references to objects which cannot be manufactured by the user but only by the system. Unused objects are automatically garbage collected. The second feature is that Java is a strongly typed language, like Pascal and Ada, and unlike C and C++. Java even performs runtime checks to avoid array index errors.

Java is implemented by compiling Java programs into Java Virtual Machine (JVM) byte codes. The byte codes are stored in class files. An interpreter, the JVM, loads the class files and executes byte codes. The JVM controls access to all machine resources. Safety in Java is therefore considered language based, as opposed to operating system based.

A simple (and inefficient) implementation of safety would carry out all sorts of runtime checks. Examples include making sure that the operands of an integer add instruction are indeed integers, to check that an object is initialised before it is used, and to check that an index is within the bounds of an array. It is more efficient for the compiler to perform the type checks. However, other checks, such as the array index check cannot normally performed by the compiler, and must be delegated to the run time system. We will discuss some proposals also to avoid this kind of runtime check in Section 5.

Javas' *write once, run anywhere* philosophy adds an interesting complication by allowing compiled Java programs (in the form of class files) from any source to be loaded into the runtime environment. This means that the checks performed by the compiler loose their validity because it is easy to tamper with class files while they are stored, or in transit. Therefore Java implementations normally include a class loader and a byte code verifier [36]. The former takes care of accepting and loading JVM programs into the Java runtime environment. The latter is essentially another type checker operating on the JVM byte codes. We will not say more about the class loader here as it mainly deals with controlling name spaces and providing the hooks for a third component, the security manager to implement a form of Security in Java. Instead we will concentrate on safety.

Both the class loader and the byte code verifier do their work before execution of the code from a newly loaded class starts. We would argue that even if all the relevant checks were performed during the execution of the code, which are now performed at load and verification time, Java applets would offer more safety than they do now. The reason is that many implementation errors [17] have been, and are being uncovered in class loaders, byte code verifiers, and in particular in the complex interplay between the class loader, the byte code verifier and the run time 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 [25]. Operational procedures are often the weakest element in security [2].

2 Methodology

If Java is to be a language used to build applications that offer safety, it needs to be well defined, so that program-

*Dept. of Electronics and Computer Science, Univ. of Southampton, Email: phh@ecs.soton.ac.uk

mers understand exactly how to use the language, and that implementors know how to realise the implementation, always maintaining the safety. This requires formal specifications of the following components:

- The semantics of Java.
- The semantics of the JVM language.
- The Java to JVM compiler.
- The runtime support, that is parts of the Java API, including all `Java.*` classes. A specification of the API is needed because for example starting and stopping threads is effectuated via the Java API and not via JVM instructions.

The methodology to build these specifications and their implementations should be to:

- Construct clear and concise formal specifications of the relevant components.
- Validate the specifications by animating 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 [55].

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, exception handling, object orientation and garbage collection;
- Careful consideration has to be given to the interaction of these features. The official SUN references [28, 37] are sometimes ambiguous, inconsistent and incomplete. See for example Bertelsen [4], who provides a long list of ambiguities in the JVM specification. Curiously, other authors do find the official SUN references complete and unambiguous [22].
- 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 40 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 at relating 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. No attempts have been made at specifying the relevant parts of the Java API.

The vast majority of the studies that we have found discuss abstractions, to make 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.

2.1 Java and JVM 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:

IM Imperative core consisting of basic data, expressions and statements.

OO Object orientation, i.e. Objects, classes, interfaces, and arrays.

TY The Java type system, or byte code verification in the JVM

CL Class loading.

EH Exception handling.

MT Multi-threading, monitors, synchronisation.

GC Garbage collection.

Most researchers in the field model parts of the imperative core, and many also deal with object orientation. We will say no more about this core, as it is well understood. Instead, we will concentrate on the remaining issues. Some authors model objects and classes but not the type system. Type soundness has been studied by quite a few. The JVM implementation of exception handling uses a difficult optimisation, which is the reason why several authors have studied this in detail. Multi-threading has found favour only with few. 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. This connection is actually ignored by some authors [5].

3 Java Semantics

In this paper we concentrate on the various reports found in the literature on specifying the semantics of Java. The most interesting aspect of studying the JVM (the byte code verifier) is perhaps less relevant to Java implementations on smart cards, mainly because it is so difficult to implement a byte code verifier within the limited resources of a smart card. However, we will revisit this issue in Section 6.

Our focus is on identifying the methodological approaches and on 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 gives a complete summary, showing 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, whether the work applies to Java, the JVM or both, a characterisation of the subset studied, an indication of the style of semantics used, and whether any proofs have been reported. The styles of semantics used are Denotational Semantics (DS), Continuation Semantics (CS), Abstract State Machine Semantics (ASM), Structured Operational Semantics (SOS), Natural Semantics (NS), a Higher Order Logic (HOL) or a semantics based on that of the tool used. See Nielson and Nielson [43] for an introduction into programming language semantics.

3.1 Object Orientation

Alves-Foss and Lam [1] present a denotational semantics of most of Java (excluding multi-threading and garbage collection, but including class loading). Their specification gives detail on the various basic data types in Java. This contributes to a better understanding of those aspects of the language.

3.2 The type system

The Java type system is based on simple sub typing, but it has one novel feature: Java offers interfaces by way of creating multiple inheritance. Drossopoulou and Eisenbach were probably the first to model this feature [24]. They give a static semantics (i.e. a specification of the type system) and a dynamic semantics (i.e. an interpreter of Java programs that works with typed data) of a relatively small subset of Java. Drossopoulou and Eisenbach then state the soundness of their type system. In a separate paper, Drossopoulou et al [23] extend their subset to include exception handling. Neither paper gives proofs. Instead Syme [55] 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 uncover 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.

Nipkow and von Oheimb [45] prove type soundness of a similar subset to Drossopoulou et al. 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, Nipkow and von Oheimb were not able to validate the specifications due to the lack of support for generating executable semantics [58]. 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 make working with semantics more accessible [58, Page 151].

Glesner and Zimmermann [26] specify the type system for a small fragment of Java as an example of their work on many sorted logic in natural semantics.

3.3 Class loader

Wragg et al [62] offer a model of class loading for a relatively small subset of Java to study one of Java's more experimental features, i.e., that of binary compatibility. In Java it is possible to add methods and fields to a Java class, without having to recompile any classes that import the class being modified. The work uncovers a serious flaw in connection with interfaces.

First Author	Ref.	Small	Purpose	Base	Tool	Java/JVM	no CL	no EH	no MT	Semantics	Proof
Alves-Foss	[1]		study semantics			Java			no MT	DS	
Bertelsen	[5]		study semantics			Java		no EH	no MT	DS	
Börger	[10]		study semantics			Java	no CL			ASM	
Cenciarelli	[13]		study semantics			Java	no CL			SOS	proof
Demartini	[18]		verification		SPIN	Java				see tool	proof
Drossopoulou	[22]		prove type soundness			Java	no CL	no EH	no MT	SOS	
Drossopoulou	[23]		prove type soundness	[24]		Java	no CL		no MT	SOS	
Drossopoulou	[24]		prove type soundness	[22]		Java	no CL		no MT	SOS	
Glesner	[26]		study semantics			Java	no CL	no EH	no MT	NS	
Havelund	[31]		verification		SPIN	Java		no EH		see tool	proof
Jacobs	[32]		verification		PVS	Java			no MT	see tool	proof
Jensen	[33]		verification			Java	no CL	no EH	no MT	SOS	
Kassab	[34]		study security			Java	no CL	no EH		HOL	
Nipkow	[45]		prove type soundness		Isabelle/HOL	Java	no CL	no EH	no MT	NS	proof
Rustan	[52]		verification	[20]	ESC/Java	Java				see tool	proof
Syme	[55]		prove type soundness	[23]	DECLARE	Java	no CL		no MT	SOS	proof
van den Berg	[57]		verification	[32]	PVS, Isabelle/HOL	Java			no MT	see tool	proof
von Oheimb	[58]		prove type soundness	[45]	Isabelle/HOL	Java	no CL		no MT	NS	proof
Wallace	[59]		study semantics			Java	no CL			ASM	
Wragg	[62]		study semantics	[24]		Java	no CL		no MT	SOS	
Börger	[7]		prove compiler correct	[11],[10]		Java+JVM				ASM	
Börger	[9]		prove compiler correct	[11],[10]	ASMGofer	Java+JVM	no CL		no MT	ASM	proof sketch
Diehl	[21]		study semantics			Java+JVM	no CL	no EH	no MT	ASM	
Rose	[50]	small	prove compiler correct			Java+JVM		no EH	no MT	NS	
Staerk	[53]		prove compiler correct	[11],[10]		Java+JVM	no CL			ASM	proof

Table 1: Java semantics

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
no CL	no model of class loading
no EH	no model of exception handling
no MT	no model of multi-threading
Semantics	an indication of the style of semantics used
Proof	Whether the paper presents proofs or proof sketches

3.4 Multi-threading

Börger and Schulte [10], and Cenciarelli et al [13] model multi-threading, at the Java level. The main interest in these two papers is the study of the issues left open by the official SUN documentation. For example, threads are able to keep local copies of information, until other threads require access to the information. Various optimisations are possible to optimise the management of this information, and Cenciarelli et al. prove some optimisations correct.

This concludes our survey on formalising the semantics of Java. We now consider work on formalising the compiler in the next section.

4 The compiler

While a considerable amount of effort has been spent on specifying the semantics of various subsets of Java and/or the JVM, relatively little work has been done on the compiler. Table 1 includes summaries of the efforts described below.

Diehl [21] 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 gives 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 [50] 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.

4.1 The Abstract State Machine approach

To conclude our discussion of Java language features we mention a numbers of papers and a forthcoming book that take a more integrated approach towards the study of the Java, the JVM and the compiler. For a number of years, Börger and Schulte 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 [29]. Two earlier papers specify a modular semantics of a subset of the JVM [11], and a subset of Java [10]. 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 [7] 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 [8] 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 [9] 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 [53] revisits the specification of Java and the JVM from Börger and Schulte [11, 10]. 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. A forthcoming book [6] promises a more complete specification of Java, the JVM, the compiler, the byte code verifier as well as correctness proofs.

As to the methodology deployed, the papers by Börger and Schulte do not give details. Most importantly, the specifications are all provided in one notation (ASM), which is essential for a consistent approach. While mechanical checking of the specifications is mentioned as a challenge to the community [9], mechanical validation of the specifications is supported by (hand) translating the abstract machines into Haskell, using the ASMGofer system. This allows ASM specifications to be executed and thus to be validated.

Wallace 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 [60]. The work is purely a study of semantics.

5 Java extensions

Several authors propose to enhance the safety of Java programs by using program verification techniques. This contributes to the safety of Java programs because they may be expected to contain fewer design and implementation problems. While some of the work we report on below has not been done specifically for smart cards, it is relevant for the practical reason that programs or applets for smart cards are generally intricate but small. This is something that the tools we report on cope well with.

5.1 Model checking

Demartini et al [18], and also Havelund et al [31] 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 et al 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. For small Java programs this line of research is useful, it is difficult to see how the result might scale up to larger systems. One of the most useful features of the SPIN model checker is its ability to display scenarios leading to problems such as deadlock. 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.

Jensen et al [33] also use model checking to verify properties of Java programs, but 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 Java's sandbox, or the stack inspection introduced by Java 2 [38].

5.2 Theorem proving

Detlefs et al, using Modula 3 [20], and more recently also using Java [52], go beyond what type checking offers by requiring the programmer to annotate programs with pre- and post-conditions. The idea is that programmers do this informally anyway, so 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 than to write down than just the pre and post conditions. Instead the system derives loop invariants automatically, which 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 thus a compromise between what is achievable with automated techniques to date 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.

The aim of the LOOP project of Jacobs et al is full verification of Java programs. They use a denotational semantics based tool to translate Java into the higher order logic of widely used theorem provers (PVS [32], or Isabelle/HOL [57]). The user then expresses properties of the translated Java programs in higher order logic and drives the appropriate theorem prover to develop the proofs. Examples of properties include termination of a method, or in-variants on the fields of a class. While the theorem provers provide a degree of automation, user intervention is required for example to introduce loop variants and in-variants. The LOOP project aspires to achieve full verification of Java programs. While the project is building tools to assist the Java programmer, it is unclear how much Java programmers will be expected to know about PVS, or Isabelle/Hol and tool assisted theorem proving.

Poetzsch-Heffter and Müller [47] give an operational and an axiomatic semantics of a subset set 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. This is proposed as future work.

Verification is not restricted to Java programs. Moore [39] has built a new version of a small subset of Cohen's specification [15] of the JVM. Moore shows how the ACL2 theorem prover is capable not only of animating the semantics of simple byte code programs, but also of proving the correctness of such programs, against a specification in terms of the models underlying programs. Both Cohen and Moore's ACL2 specifications are rather verbose, as the notation used in ACL2 is Lisp.

5.3 Controlling type casts

Java extensions can be conducive to better safety. One particular example is Java's lack of polymorphism, which requires programmers to insert type casts in their programs. Consider for example to 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 like Pizza [46] and Generic Java [12] address these problems by 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.

5.4 Controlling execution time

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 to control resources. While there are languages [3] and type systems [16] that have been designed to guarantee termination, we have not been able to find efforts that apply such technology to Java.

5.5 Code certification

Necula and Lee introduced the idea of proof carrying code (PCC) [40]. This is a partly automatic verification technique for assembly level programs designed to allow a code consumer to have trust in the products of a code producer. One might argue that this would then be not a Java but more a JVM issue. However we report it here as it relies on automatic program verification techniques, like most of the other work reported in this section.

PCC works as follows (ignoring the negotiations between producer and consumer about the safety policy to be used). The producer expresses a safety property in terms of pre and post conditions on the program. In addition, the producer annotates the program, with loop invariants etc. Then 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. Since it is more difficult to generate a proof than to check it, 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.

One of the problems of the PCC approach is that the size of a proofs may be exponential in the size of the program [42]. A proof may become large because of the amount of redundancy. Necula and Lee [41] 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. In spite of this improvement, proofs may still be exponentially large.

We conclude this section with a cautionary note. Program verification requires special skills, to formulate properties, to discover appropriate loop invariants, to drive mechanical theorem provers etc. Few programmers have these skills. It is thus essential that tools are automatic, or at least require as little programmer intervention as possible.

6 Small footprint devices

Java implementations are resource hungry. For example even the smallest JVM implementations require at least 1 MB of store [61]. 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, which has been designed specially to fit into small footprint devices requires at least 128KB of RAM [54]. Please note that we are side stepping the fact that Java is not suitable for real time applications [61].

The most extreme example of a small device is probably a smart card, which typically offers a few hundred bytes of RAM and a dozen or so KB of EEPROM. The current solution for smart cards as licensed by Sun to the smart card industry is to subset Java and the JVM. Only programs written in the Java-Card subset can be run on the Java-Card VM (JCVM). This has three disadvantages:

- 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 the smallest embedded controllers 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.

The implementation of Java for smart cards is based on the Split VM concept, which pushes part of the byte code verification from the loading to the compilation/linking phase. A converter from the JVM byte codes to the JCVM format performs the byte code verification and optimises and prepares the code for loading into the device.

6.1 Byte code compression

Clausen et al [14] retain JVM byte codes, but propose to compress them for the benefit of embedded systems. The compression technique works by discovering commonly

occurring sequences of instructions, which are then replaced by a new 'macro' instruction. This requires modifications to the JVM. While the technique is reported to save up to 30% space at the cost of an increase of up 30% loading time, it remains unclear how the compression technique interacts with for example the byte code verifier.

6.2 Class file conversion

Hartel et al [30] provide a complete specification of an early version of 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.

An interesting methodological point to note is that the earlier JSP was designed essentially by starting from the full JVM, and then cutting back unwanted features. The newer KVM 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 [56]. 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 [44].

Lanet and Requet [35] 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 include:

1. A specification of the constraints imposed by the byte code verifier for a small subset (the imperative core and method calls) of the JVM.
2. A specification of the semantics of this subset of the JVM byte codes.
3. A specification of the semantics of the corresponding subset of the JCVM byte codes.
4. A proof that the specification of the JCVM subset is a *data refinement* of the JVM subset.

The subsets are 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.

Denney and Jensen [19] 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.

6.3 Byte code verification revisited

As we said before, a small footprint device (a smart card) does not have enough memory to perform byte code verification. The split VM concept 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 [49, 48] 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. They give no supporting evidence for this claim. In a separate paper [27], Posegga et al 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 appear to be somewhat disappointing, as proving theoremhood of some simple first order logic formulae may take of the order of minutes.

Rose and Rose [51] 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) is to check based on the reconstructed types, that each instruction is correctly typed. The advantages are, firstly that 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. The second advantage is that 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 annoying,

which could have been avoided if Rose and Rose had used tool support. Furthermore, exception handling has been omitted, which as we have seen before does complicate byte code verification considerably.

7 Conclusions

Java programs offer type and memory safety because of properties of the Java language. However, it has proved difficult to implement the safety features correctly. The main reason is that 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 run time system. The various components responsible for safety interact in complex ways, creating scope for design and implementation problems. Yet in spite of all the optimisations, Java programs today are still slower than C or C++ programs.

New implementation techniques are needed to make Java simpler and faster, whilst at the same time making the implementations more amenable to formal modelling. Formal models offer a way of studying the different components responsible for safety, and for studying the interactions between these components.

Not all formal methods and semantics tools (ACL2, ASMGofer, Atelier B, Coq, DECLARE, ESC/Java, Haskell, Isabelle/HOL, LETOS, PVS, SMV, SpecWare, SPIN) that have been brought to bear on Java are sufficiently automatic, or sufficiently 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. Isabelle/HOL is one of the most popular tools, but even its users complain about lacking mathematical theories and validation facilities [58]. This clearly needs improvement.

Almost all efforts that we have discussed, either to formalise parts of Java, or its implementation have uncovered ambiguities and inconsistencies in the official SUN documentation, and/or problems with the various implementations. This should be considered a clear success of applying formal techniques. However, much work remains to be done:

- On modelling garbage collection, and the Java API.
- On building more appropriate theories for programming language semantics modelling.
- On simplifying and modularising the individual components of Java implementations.
- On reducing the size of the trusted computing base, so that flaws are less likely to compromise the security of the system as a whole.
- On considering formal specification, validation and provably correct implementation as a whole, rather than in separation.
- On presenting clear and concise formalisations of systems, which are accessible to the designers and implementors of these systems.
- On using machine-readable specifications.

We believe that work in each of these areas is both interesting and will lead to novel results, as the combination of features offered by Java is rather different from other languages.

We have made an effort to survey all relevant literature on Java safety, and in particular the relation with smart cards. We have tried to make the survey as accurate as possible. However, we welcome to hear about work that we have not surveyed yet, and about errors and inaccuracies in the survey.

Acknowledgements

The help of Egon Börger and Luc Moreau is gratefully acknowledged.

References

[1] J. Alves-Foss and F. S. Lam. Dynamic denotation semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java, LNCS 1523*, pages 201–240. Springer-Verlag, Berlin, 1999.

[2] R. J. Anderson. Making smart card systems robust. In V. Cordonnier and J.-J. Quisquater, editors, *1st Int. Conf. Smart card research and advanced application (CARDIS)*, pages 1–14, Lille France, Oct 1994. Univ. de Lille, France.

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

[4] P. Bertelsen. Dynamic semantics of Java byte code. *Future Generation Computer Systems*, page to appear, 1999.

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

[6] E. Börger, J. Schmid, W. Schulte, and R. Stärk. *Java and the Java Virtual Machine: Definition, Verification, Validation, LNCS in preparation.* Springer-Verlag, Berlin, 2000.

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

[8] E. Börger and W. Schulte. Initialization problems for Java. *Software Concepts and Tools*, 20(4), 1999.

[9] E. Börger and W. Schulte. A practical method for specification and analysis of exception handling – a Java/JVM case study. *IEEE Transactions on software engineering*, page to appear, 1999.

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

[11] E. Börger and W. Schulte. Modular design for the Java virtual machine architecture. In E. Börger, editor, *Architecture Design and Validation Methods.* Springer-Verlag, Berlin, 2000.

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

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

[14] L. R. Clausen, U. P. Schultz, C. Consel, and G. Muller. Java bytecode compression for embedded systems. Rapport de recherche RR-3578, INRIA, Rennes, Dec 1998.

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

[16] K. Crary and S. Weirich. Resource bound certification. In *27th Int. Conf. Principles of programming languages (POPL)*, page to appear, Boston, Massachusetts, Jan 2000. ACM, New York. `www.cs.cmu.edu/ afs/ cs/ user/ crary/ www/ papers/`.

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

[18] C. Demartini, R. Iosif, and R. Sisto. 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/`.

[19] E. Denney and Th. Jensen. Correctness of Java card method lookup via logical relations. In D. Watt, editor, *9tht European Symp. on programming (ESOP), LNCS*, page to appear, Berlin, West Germany, Mar 2000. Springer-Verlag, Berlin.

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

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

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

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

[24] S. Drossopoulou, S. Eisenbach, and S. Khurshid. Is the Java type system sound? *Theory and practice of object systems*, 1997.

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

[26] S. Glesner and W. Zimmermann. 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*, pages 249–62. Chapman & Hall, London, 1998.

[27] R. Goré, J. Posegga, A. Slater, and H. Vogt. cardTAP: Automated deduction on a smart card. In *Joint Australian Artificial Intelligence Conf., LNAI*, Brisbane, Australia, Jul 1998. Springer Verlag, Berlin.

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

[29] Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

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

[31] K. Havelund and T. Pressburger. Model checking Java programs using pathfinder. *Software Tools for Technology Transfer*, page to appear, Mar 1999. `http://ase.arc.nasa.gov/havelund/`.

[32] B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about Java classes. In *Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 329–340, Vancouver, Canada, Oct 1998. ACM Press, New York.

[33] T. Jensen, D. Le Metayer, and T. Thorn. Verification of control flow based security properties. In *Symposium on Security and Privacy*, pages 89–103, Oakland, California, May 1999. IEEE Comput. Soc, Los Alamitos, California.

[34] L. Kassab and S. Greenwald. Towards formalizing the Java security architecture in JDK 1.2. In J.-J. Quisquater, Y. Deswarte, C. Meadows, and D. Gollmann, editors, *European Symposium on Research in Computer Security (ESORICS), LNCS 1485*, pages 191–207, Leuven-la-Neuve, Belgium, Sep 1998. Springer-Verlag, Berlin.

[35] J.-L. Lanet and A. Requet. Formal proof of smart card applets correctness. In J.-J. Quisquater and B. Schneier, editors, *3rd Int. Conf. Smart card research and advanced application (CARDIS 1998 preproceedings)*, Louvain la Neuve, Belgium, Sep 1998. Univ. Catholique de Louvain la Neuve.

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

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

[38] G. McGraw and E. W. Felten. *Securing Java: Getting down to business with mobile code*. John Wiley & Sons, Chichester, UK, second edition, 1999.

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

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

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

[42] G. C. Necula and P. Lee. Safe, untrusted agents using Proof-Carrying code. In G. Vigna, editor, *Mobile Agents and Security, LNCS 1419*. Springer-Verlag, Berlin, Jan 1998.

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

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

[45] T. Nipkow and D. von Oheimb. Java$_{light}$ is Type-Safe — definitely. In *25th Int. Conf. Principles of programming languages (POPL)*, pages 161–170, San Diego, California, Jan 1998. ACM, New York.

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

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

[48] J. Posegga and H. Vogt. Byte code verification for Java smart cards based on model checking. In J.-J. Quisquater, Y. Deswarte, C. Meadows, and D. Gollmann, editors, *European Symposium on Research in Computer Security (ESORICS), LNCS 1485*, pages 175–190, Leuven-la-Neuve, Belgium, Sep 1998. Springer-Verlag, Berlin.

[49] J. Posegga and H. Vogt. Java bytecode verification using model checking. In *OOPSLA'98 Workshop on Formal Underpinnings of Java (FUJ)*, Vancouver, BC, Canada, Nov 1998. `http://www-dse.doc.ic.ac.uk/~sue/oopsla/cfp.html`.

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

[51] E. Rose and K. H. Rose. Lightweight bytecode verification. In *OOPSLA'98 Workshop on Formal Underpinnings of Java (FUJ)*, Vancouver, BC, Canada, Nov 1998. `http://www-dse.doc.ic.ac.uk/~sue/oopsla/cfp.html`.

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

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

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

[55] D. Syme. Proving Java type soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java, LNCS 1523*, pages 83–118. Springer-Verlag, Berlin, 1999.

[56] A. Taivalsaari, B. Bush, and D. Simon. The spotless system: Implementing a JavaTMSystem for the palm connected organizer. Technical report TR-99-73, Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303 USA, Feb 1999.

[57] J. van den Berg, M. Huisman, B. Jacobs, and E. Poll. *A Type-Theoretic Memory Model for Verification of Sequential Java Programs.* Univ. Nijmegen, Dept Comp Sci, Netherlands, Nov 1999.

[58] D. von Oheimb and T. Nipkow. Machine-Checking the Java specification: Proving type safety. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java, LNCS 1523*, pages 119–156. Springer-Verlag, Berlin, 1999.

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

[60] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible security architectures for Java. Technical report 546-97, Dept. of Comp. Sci, Princeton Univ., Apr 1997.

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

[62] D. Wragg, S. Drossopolou, and S. Eisenbach. Java binary compatibility is almost correct. Research report, Dept. of Computing, Imperial College London, Feb 1998.