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

Pieter H. Hartel † Luc Moreau ‡

March 25, 2001

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, and thence to achieve ITSEC level 6 or Common Criteria level 7 certification for Java programs.

Keywords: D.2.4 Software/Program Verification, D.3.1 Formal Definitions and Theory, C.3 Special-purpose and application-based systems (Smart cards)

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 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, by studying the formal semantics of Java.

While type safety and memory safety are necessary, they are not sufficient to make Java programs secure [71, 44]. For a Java program to be secure we make two main requirements [37]. The first is that Java programs can be restricted to accessing only certain resources, as defined by an appropriate security policy [75]. 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 [49]. 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 ter-
minology and concepts of safety and security in programming languages may be found in Volpano and Smith [180]. We should also like to point out that the boundary between safety and security is sometimes a little vague [174]. 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 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 [79, 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 [79, 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 [79, 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 [71] or applications [113] 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 [118].

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 [51], 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 [60].
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% [158, 190]. 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 the 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 standard [95] and the more recent Common Criteria [140] 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 [171].

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 necessary 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 support 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 [78, 114] are sometimes ambiguous, inconsistent and incomplete. See for example Bertelsen [20], 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 [47] and the LOOP
team [144]. (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 as-
sumptions 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 cer-
tain 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 Se-
manitics (CS), Denotational Semantics (DS), Natural Se-
manitics (NS), Operational Semantics (OS), Structural
Operational Semantics (SOS), or a semantic embedding
in a higher order logic (HOL).

We refer the reader to Nielson and Nielson [132] for an
introduction into programming language semantics, and
to Nipkow et al. [135] for a brief introduction into the
notation of embedding the semantics of a programming lan-
guage 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 fea-
tures of Java in detail: dynamic class loading, the mem-
ory 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 spe-
cification methods, and in particular the accompanying sup-
port 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
whether the work is particularly relevant to small foot-
print 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 [10] present a
denotational semantics of a Java subset (excluding multi-
threading and garbage collection, but including class load-
ing). The specification gives considerable detail on the
various basic data types in Java. The semantics is not
used for any particular purpose. An alternative deno-
tational semantics is offered by Cenciarelli [35] 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 [191] and Jones [102] to
model the semantics of byte codes.

Attali et al. [16] discuss a reasonably complete exe-
cutable, operational semantics of Java built using the Cen-
taur system. The specification includes concurrency, but
it omits exceptions, arrays, and packages. The aim of
this work is to use the Centaur system to generate an ap-
propriate 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 com-
mon behaviour without sharing any implementation.
Drossopoulou and Eisenbach modelled these features [60],
by giving a static semantics and a small-step dynamic
semantics of a fairly large subset of sequential Java.
Drossopoulou and Eisenbach then state the soundness
of a type system based on simple subtyping using sub-
ject reduction. In a separate paper, Drossopoulou et
al. [58] extend their subset to include exception handling.
Results were proved by hand but neither paper gives
proofs. Instead Syme [171] encodes some of the mod-
els 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.
<table>
<thead>
<tr>
<th>First Author</th>
<th>Ref.</th>
<th>Small Purpose</th>
<th>Base</th>
<th>Tool</th>
<th>CL</th>
<th>EH</th>
<th>MT</th>
<th>Semantics</th>
<th>Proof</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abrahám-Mumm</td>
<td>[4]</td>
<td>prove Hoare logic sound</td>
<td></td>
<td></td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>OS</td>
<td>yes</td>
</tr>
<tr>
<td>Alves-Foss</td>
<td>[10]</td>
<td>study semantics</td>
<td></td>
<td></td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>DS</td>
<td>no</td>
</tr>
<tr>
<td>Attali</td>
<td>[16]</td>
<td>executable specification</td>
<td></td>
<td>Centaur</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>NS+SOS</td>
<td>no</td>
</tr>
<tr>
<td>Bertelsen</td>
<td>[21]</td>
<td>study semantics</td>
<td></td>
<td></td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>DS</td>
<td>no</td>
</tr>
<tr>
<td>Börger</td>
<td>[27]</td>
<td>study semantics</td>
<td></td>
<td></td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>ASM</td>
<td>no</td>
</tr>
<tr>
<td>Cenciarelli</td>
<td>[36]</td>
<td>study semantics</td>
<td></td>
<td></td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>SOS</td>
<td>yes</td>
</tr>
<tr>
<td>Cenciarelli</td>
<td>[35]</td>
<td>study semantics</td>
<td></td>
<td></td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>DS</td>
<td>no</td>
</tr>
<tr>
<td>Coscia</td>
<td>[48, 47]</td>
<td>verification</td>
<td>[57]</td>
<td></td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>NS+SOS</td>
<td>yes</td>
</tr>
<tr>
<td>Demartini</td>
<td>[52]</td>
<td>verification</td>
<td>SPIN</td>
<td></td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>embed</td>
<td>yes</td>
</tr>
<tr>
<td>Drossopoulou</td>
<td>[60, 57, 58, 61]</td>
<td>prove type soundness</td>
<td></td>
<td></td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>CR</td>
<td>no</td>
</tr>
<tr>
<td>Drossopoulou</td>
<td>[62, 56]</td>
<td>study semantics</td>
<td></td>
<td></td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>CR</td>
<td>yes</td>
</tr>
<tr>
<td>Glesner</td>
<td>[73]</td>
<td>study semantics</td>
<td></td>
<td></td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>AS</td>
<td>yes</td>
</tr>
<tr>
<td>Gottmaker</td>
<td>[76]</td>
<td>memory model</td>
<td>[8]</td>
<td></td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>AS</td>
<td>no</td>
</tr>
<tr>
<td>Havelund</td>
<td>[86]</td>
<td>verification</td>
<td>SPIN</td>
<td></td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>embed</td>
<td>yes</td>
</tr>
<tr>
<td>Igarashi</td>
<td>[93]</td>
<td>prove type soundness</td>
<td></td>
<td></td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>NS</td>
<td>yes</td>
</tr>
<tr>
<td>Igarashi</td>
<td>[94]</td>
<td>calculus</td>
<td></td>
<td></td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>SOS</td>
<td>yes</td>
</tr>
<tr>
<td>Jacobs</td>
<td>[99]</td>
<td>verification</td>
<td>PVS</td>
<td></td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>embed</td>
<td>yes</td>
</tr>
<tr>
<td>Jensen</td>
<td>[101]</td>
<td>verification</td>
<td></td>
<td></td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>SOS</td>
<td>no</td>
</tr>
<tr>
<td>Kassab</td>
<td>[103]</td>
<td>study security</td>
<td></td>
<td></td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>HOL</td>
<td>no</td>
</tr>
<tr>
<td>Leino</td>
<td>[112]</td>
<td>verification</td>
<td>[54]</td>
<td>ESC/Java</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>embed</td>
<td>yes</td>
</tr>
<tr>
<td>Maessen</td>
<td>[115]</td>
<td>memory model</td>
<td></td>
<td></td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>algebraic</td>
<td>no</td>
</tr>
<tr>
<td>Manson</td>
<td>[117]</td>
<td>memory model</td>
<td></td>
<td></td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>OS</td>
<td>no</td>
</tr>
<tr>
<td>Nipkow</td>
<td>[134]</td>
<td>prove type soundness</td>
<td></td>
<td>Isabelle/HOL</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>NS</td>
<td>yes</td>
</tr>
<tr>
<td>Poetsch-Heftner</td>
<td>[143]</td>
<td>prove Hoare logic sound</td>
<td></td>
<td>Isabelle/HOL</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>NS</td>
<td>yes</td>
</tr>
<tr>
<td>Skalka</td>
<td>[160]</td>
<td>study stack inspection</td>
<td></td>
<td></td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>OS</td>
<td>no</td>
</tr>
<tr>
<td>Stärk</td>
<td>[164]</td>
<td>study of semantics, verification</td>
<td>[27]</td>
<td>AsmGofer</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>ASM</td>
<td>yes</td>
</tr>
<tr>
<td>Synek</td>
<td>[171]</td>
<td>prove type soundness</td>
<td>[58]</td>
<td>DECLARE</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>SOS</td>
<td>yes</td>
</tr>
<tr>
<td>van den Berg</td>
<td>[177]</td>
<td>verification</td>
<td>[99]</td>
<td>PVS, Isabelle/HOL</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>embed</td>
<td>yes</td>
</tr>
<tr>
<td>von Oheimb</td>
<td>[182]</td>
<td>prove type soundness</td>
<td>[134]</td>
<td>Isabelle/HOL</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>NS</td>
<td>yes</td>
</tr>
<tr>
<td>von Oheimb</td>
<td>[181]</td>
<td>soundness, completeness</td>
<td>[182]</td>
<td>Isabelle/HOL</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>AS</td>
<td>yes</td>
</tr>
<tr>
<td>Wallace</td>
<td>[184]</td>
<td>study semantics</td>
<td></td>
<td></td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>ASM</td>
<td>no</td>
</tr>
<tr>
<td>Wallach</td>
<td>[186]</td>
<td>study stack inspection</td>
<td>[3]</td>
<td></td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>modal logic</td>
<td>yes</td>
</tr>
<tr>
<td>Wallach</td>
<td>[185]</td>
<td>study stack inspection</td>
<td></td>
<td></td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>translation</td>
<td>yes</td>
</tr>
</tbody>
</table>
in the hand written proofs of Drossopoulou and Eisenbach.

Drossopoulou and Valkeych [61] propose a semantics of a Java subset with both checked and unchecked exceptions. The semantics relies on Felleisen's context rewriting technique to model the dynamic behaviour of exceptions. As far as typing is concerned, exceptions are regarded as "effects" [173]; 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 [73] 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 [93] 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 [134] prove type soundness of their Java_{light} subset, which is similar to the subset used by Drossopoulou et al. [60]. 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 [182]. 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 [182, Page 151]. To complement the operational semantics of Java_{light} [182], von Oheimb presents an axiomatic semantics [181], 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 [94] 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 Flat et al. [64] 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. Errorful type casts will eventually cause unexpected runtime exceptions. Java extensions such as Pizza [137] and Generic Java [32] 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 define 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. [94] 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 [7], NexGen [33], PolyJ [126], 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 if preexisting binaries that previously linked without error will continue to link without error (cf. [78, 79, Ch. 13]). More specifically, a list of important binary compatible changes supported by Java is enumerated in the language specification. Drossopoulou, Eisenbach and
Wragg [62, 59] 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. [62] study the problem of separate compilation and binary compatibility on the significant subset of Java described in [60], 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 [62]. 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. [62, 59].

In contrast to their first formalisation of binary compatibility based the language and typing judgements of [60], Drossopoulou et al. 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 [59]. The authors discuss a number of interpretations of the language specification [78, 79]; 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 [62] are established using the new axiomatic definitions.

In a subsequent paper, Drossopoulou [56] 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 [60] 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 [152], and does not integrate the notion of link compatibility [62].

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 to transfer the contents of its working copy of a variable into the master copy of vice-versa [79, Ch. 17.1].

These rules are not straightforward to understand because they rely on a double indirectness, 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. [36] 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 [79, 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.)

Gontmaker and Schuster investigate the Java memory model in isolation (independently of the language itself) [76]. They provide a non-operational, trace based characterisation of the memory model, which they compare with existing memory models [8]. Gontmaker 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) [8] 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 [12] 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 Allam [38] 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 [13, 14]. 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 [147] wrote a critique of the Java memory model and he argues that it requires Coherence (Cache Consistency) [8], 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, Strings may not be thread safe, and it may be difficult to efficiently implement their thread safety on shared memory multiprocessors with weak memory models. Alternative memory models, proposed by Manson and Pugh [117] and Maessen et al. [115], 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; [117] contains an operational description of memory related operations, whereas [115] 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. [36] 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 [9, 24, 40, 189], but may break the semantics of programs over the new proposed memory models.

Coscia and Reggio [48, 47] 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 [4] present an operational and an axiomatic semantics of a subset of Java. The purpose of their work is to provide a framework 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 et al. [103] 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 [185] develop an abstract model of stack inspection in terms of a belief logic, known as ABPL (Abadi, Burrows, Lampson and Plotkin) [3]. 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 [185] 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 [160] 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 [87] 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 [123], 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 [2] and Cenciarelli et al. [36], 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).
<table>
<thead>
<tr>
<th>First Author</th>
<th>Ref.</th>
<th>Small</th>
<th>Purpose</th>
<th>Base</th>
<th>Tool</th>
<th>CL</th>
<th>EH</th>
<th>MT</th>
<th>Semantics</th>
<th>Proof</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bertelsen</td>
<td>[19]</td>
<td>yes</td>
<td>study semantics</td>
<td></td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>OS</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Bertelsen</td>
<td>[20]</td>
<td>yes</td>
<td>study semantics</td>
<td>[19]</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>OS</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Biglardi</td>
<td>[23]</td>
<td>yes</td>
<td>study semantics</td>
<td>[165]</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>OS</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Börger</td>
<td>[28]</td>
<td>yes</td>
<td>study semantics</td>
<td></td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>ASM</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Coglio</td>
<td>[42]</td>
<td>yes</td>
<td>prove type soundness</td>
<td>[74]</td>
<td>SpecWare</td>
<td>yes</td>
<td>no</td>
<td>embed</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Cohen</td>
<td>[43]</td>
<td>yes</td>
<td>study semantics</td>
<td></td>
<td>ACL2</td>
<td>yes</td>
<td>no</td>
<td>embed</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Dean</td>
<td>[50]</td>
<td>yes</td>
<td>study semantics</td>
<td></td>
<td>PVS</td>
<td>yes</td>
<td>no</td>
<td>embed</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Denny</td>
<td>[53]</td>
<td>yes</td>
<td>prove converter correct</td>
<td>[19]</td>
<td>Coq</td>
<td>no</td>
<td>no</td>
<td>OS</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Fong</td>
<td>[66]</td>
<td>yes</td>
<td>study semantics</td>
<td></td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>HOL</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Freund</td>
<td>[67]</td>
<td>yes</td>
<td>study semantics</td>
<td>[68], [165]</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>OS</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Freund</td>
<td>[69, 70]</td>
<td></td>
<td>study semantics</td>
<td>[165]</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>OS</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Goldberg</td>
<td>[74]</td>
<td>yes</td>
<td>prove type soundness</td>
<td></td>
<td>SpecWare</td>
<td>yes</td>
<td>no</td>
<td>embed</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Hagiya</td>
<td>[82]</td>
<td>yes</td>
<td>study semantics</td>
<td>[165]</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>OS</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Hagiya</td>
<td>[175, 176]</td>
<td></td>
<td>study semantics</td>
<td>[165]</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>OS</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Hartel</td>
<td>[84]</td>
<td>yes</td>
<td>study semantics</td>
<td></td>
<td>LETOS</td>
<td>yes</td>
<td>no</td>
<td>OS</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Jensen</td>
<td>[100]</td>
<td>yes</td>
<td>study semantics</td>
<td></td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>OS</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Jones</td>
<td>[102]</td>
<td>yes</td>
<td>prove type soundness</td>
<td></td>
<td>Haskell</td>
<td>no</td>
<td>yes</td>
<td>CS</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Klein</td>
<td>[106]</td>
<td>yes</td>
<td>prove type soundness</td>
<td>[156]</td>
<td>Isabelle</td>
<td>no</td>
<td>no</td>
<td>NS</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Lanet</td>
<td>[110]</td>
<td>yes</td>
<td>prove converter correct</td>
<td>[150]</td>
<td>Atelier B</td>
<td>no</td>
<td>no</td>
<td>embed</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Moore</td>
<td>[121]</td>
<td>yes</td>
<td>verification</td>
<td>[43]</td>
<td>ACL2</td>
<td>no</td>
<td>no</td>
<td>embed</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>O'Callahan</td>
<td>[136]</td>
<td>yes</td>
<td>study semantics</td>
<td>[165]</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>CS</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Posegga</td>
<td>[18]</td>
<td>yes</td>
<td>prove type soundness</td>
<td>[25]</td>
<td>SMV</td>
<td>no</td>
<td>yes</td>
<td>tool</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Pusch</td>
<td>[148]</td>
<td>yes</td>
<td>prove type soundness</td>
<td>[150]</td>
<td>Isabelle/HOL</td>
<td>no</td>
<td>yes</td>
<td>OS</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Pusch</td>
<td>[149]</td>
<td>yes</td>
<td>prove type soundness</td>
<td>[148]</td>
<td>Isabelle/HOL</td>
<td>yes</td>
<td>no</td>
<td>OS</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Qian</td>
<td>[150]</td>
<td>yes</td>
<td>prove type soundness</td>
<td>[148]</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>OS</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Qian</td>
<td>[152]</td>
<td>yes</td>
<td>prove type soundness</td>
<td>[150]</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>OS</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Requet</td>
<td>[154]</td>
<td>yes</td>
<td>prove type soundness</td>
<td>[165]</td>
<td>Atelier B</td>
<td>no</td>
<td>no</td>
<td>OS</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Rose</td>
<td>[156]</td>
<td>yes</td>
<td>prove type soundness</td>
<td>[155]</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>NS</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Stata</td>
<td>[165, 166]</td>
<td></td>
<td>study semantics</td>
<td></td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>OS</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Stephenson</td>
<td>[167]</td>
<td>yes</td>
<td>study semantics</td>
<td></td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>DS</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Wallach</td>
<td>[186]</td>
<td>yes</td>
<td>study security</td>
<td></td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>HOL</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Yelland</td>
<td>[191]</td>
<td>yes</td>
<td>prove type soundness</td>
<td></td>
<td>Haskell</td>
<td>yes</td>
<td>no</td>
<td>CS</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 2: JVM Language– For Legend see Table 3
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 [2], who 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 [19, 20], and Stephenson [167] 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 [43]. 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.

Java statement:

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

Compiled JVM code:

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

Figure 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).

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 [165, 166] 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 [114] and easier to understand and reason about than Sun’s reference implementation. Stata and Abadi’s initial publication [165] considered a small but representative set 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 [68] used the same typing framework to investigate the problem of object initialisation. Then, Freund and Mitchell [70] integrated in a single framework the handling of subroutines and object initialisation. The same authors again [69] added objects, classes, interfaces, arrays, exceptions and double types. Using the same approach, Hagiya and Tozawa [175, 176] 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 [23] 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 ille-
Alternative approaches to formalise the byte code verification process have been proposed. Goldberg [74], and Qian [150] 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 [42] extend their previous work towards a provably correct implementation of their specifications, using SpecWare [162]. 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 [151] 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 [148, 149] offers an alternative to Qian’s work [150], 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 [102] 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 this 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 [191] presents a continuation semantics of a small subset of the JVM (the μVM) using the functional programming language Haskell. Using monads [183] and Rémy’s encoding of sub typing in Haskell’s type system [153] (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 [169]; 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:

<table>
<thead>
<tr>
<th>local type</th>
<th>purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 ref</td>
<td>to refer to the current object this</td>
</tr>
<tr>
<td>1 int</td>
<td>to represent the int argument of the method bar</td>
</tr>
<tr>
<td>2 int</td>
<td>to save the int type result of this.foo()</td>
</tr>
<tr>
<td>3 ref</td>
<td>to save the exception reference</td>
</tr>
<tr>
<td>4 address</td>
<td>to save the address to which the subroutine at address 27 should return</td>
</tr>
</tbody>
</table>

Each local 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 local 2 to store the return address at statement 21, and to reload the return address again from local 2 at statement 25, as indicated in comments in the JVM code of Figure 2. This would remove the need for local 3. While verifying the byte code of the subroutine at address 27, the verifier must now take into account that local 2 may either have type int (when the call originates from 10), or that local 2 may be a reference (when the call originates from 22). This optimisation is safe because local 2 is not actually used in the subroutine. We say that local 2 is a polymorphic variable of the subroutine.

The second potential problem with JVM subroutines is that the jar (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 in-
Java Method:

```java
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
```

Figure 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).

The JVM however supports returns of multiple subroutines with a single `ret` instruction, and this facility is formalised by Freund and Mitchel [70].

Concerned about the byte code verification complexity, Freund [67] 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 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 [82].

O’Callahan provides yet another solution to the problem by using a polymorphic type system for subroutines [136]. 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 [6], 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 synchronisations, 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 [114]. Bigiardi and Laneve [23] extend Stata and Abadi [165] and Freund and Mitchell [68] 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 [50] 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 [157] 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 [113]. 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: (i) a distinction is introduced between the initiating and defining loaders of a class; (ii) a loaded class cache per loader maintains temporal namespace consistency; (iii) and namespace consistency between delegating loaders is preserved by a set of constraints. Loading constraints are triples \( (L_1, L_2, N) \), 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. [100] 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 [157]. The formalisation of Jensen et al. is criticised for containing some inaccuracies, both inadvertent and deliberate ones [31]. 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 [175, 176] 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 175. 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 [152] 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 formalisation [175, 176]. Subtype constraints have the form \( (L, N_1, N_2) \) 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 [41], describe examples of type safety violation that the proposed formalisation avoids.

Fong et al. are concerned that byte code verification,
class loading and linking are too tightly coupled, to allow for the three components to be verified effectively [66]. 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 [107] 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 [72] 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' Slim binaries [165] 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 TIL [124]. The JVM offers less polymorphism than TIL but supports subroutines absent from TIL. 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 process, and examine some of the proposed static analysis 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 bytecode 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 [2] 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 [55] gives the compilation schemes for a subset of the Java that excludes exception handling, multithreading and garbage collection to the corresponding subset of the JVM. He also offers an operational semantics
<table>
<thead>
<tr>
<th>First Author</th>
<th>Ref</th>
<th>Small</th>
<th>Purpose</th>
<th>Base</th>
<th>Tool</th>
<th>CL</th>
<th>EH</th>
<th>MT</th>
<th>Semantics</th>
<th>Proof</th>
</tr>
</thead>
<tbody>
<tr>
<td>Börger</td>
<td>[29]</td>
<td>yes</td>
<td>prove compiler correct</td>
<td>[28],[27]</td>
<td>AsmGofer</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>ASM</td>
<td>yes</td>
</tr>
<tr>
<td>Choi</td>
<td>[40]</td>
<td></td>
<td>static analysis</td>
<td></td>
<td></td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>non standard</td>
<td>yes</td>
</tr>
<tr>
<td>Diehl</td>
<td>[55]</td>
<td></td>
<td>study semantics</td>
<td></td>
<td></td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>ASM</td>
<td>no</td>
</tr>
<tr>
<td>Flanagan</td>
<td>[63]</td>
<td></td>
<td>static analysis</td>
<td></td>
<td></td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>CR</td>
<td>no</td>
</tr>
<tr>
<td>Nipkow</td>
<td>[135]</td>
<td>yes</td>
<td>prove type soundness</td>
<td>[149]</td>
<td>Isabelle</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>embed</td>
<td>yes</td>
</tr>
<tr>
<td>Rose</td>
<td>[155]</td>
<td>yes</td>
<td>prove compiler correct</td>
<td></td>
<td></td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>NS</td>
<td>no</td>
</tr>
<tr>
<td>Stärk</td>
<td>[163]</td>
<td>yes</td>
<td>prove compiler correct</td>
<td>[28],[27]</td>
<td>AsmGofer</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>ASM</td>
<td>yes</td>
</tr>
<tr>
<td>Stärk</td>
<td>[164]</td>
<td>yes</td>
<td>study of semantics, verification</td>
<td>[27]</td>
<td>AsmGofer</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>ASM</td>
<td>yes</td>
</tr>
<tr>
<td>Whaley</td>
<td>[189]</td>
<td></td>
<td>static analysis</td>
<td></td>
<td></td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>non standard</td>
<td>no</td>
</tr>
</tbody>
</table>

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

Table 2: Combined Java and JVM languages
of this JVM subset. No specification of the Java subset is given, thus missing the opportunity to prove the compilation schemes correct.

Rose [155] 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. [135] 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 [80]. Two earlier papers specify a modular semantics of a subset of the JVM [28], and a subset of Java [27]. 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 [25] 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 [26] 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 [29] 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 [163] revisits the specification of Java and the JVM from Börger and Schulte [28, 27]. 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 [164] 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 bytecode 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 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 [184]. 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 [189] 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. [161] 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. [40]. 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 [63] present a static analysis of Java programs able to detect race conditions in multi-threaded programs. Their analysis is based on a new type system, which like an “effect system” [173], specifies the locks required by expressions. Their formalisation is based on an extension of Flat et al.’s CLASSICJAVA with fork and synchronized statements [64]. 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 [17] rename the standard AppletClassLoader to IpplletClassLoader, and then insert a new AppletClassLoader that subclasses IpplletClassLoader. The new class pops up a security dialogue, offering the user full control.

Shin and Mitchell not only rename classes but also methods. This is necessary because final classes cannot be subclassed [159]. They show how some common attacks can be avoided using their method.

Malkhi and Reiter 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 [116]. 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 [67].

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. [92] 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 bytecode interpreters, and were rapidly replaced by just-in-time compilers able to transform bytecode into assembly code. We are now observing the emergence of a new generation of optimising compilers such as Jalapeno [161]. Moreira et al. [122] 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 [81], 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. [52], and also Havelund et al. [86] 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. 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. [101] 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 [118].

The Bandera project [46] 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 [189] 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 these 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 [88, 188].

7.2 Theorem Proving

Dettefs et al., using Modula 3 [54], and more recently also Java [112], 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 Dettefs 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 [143] 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 [121] has built a new version of a small subset of Cohen’s specification [43] 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 [11] and type systems [49] 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 [178] translates the program and the specifications into the Higher Order Logic (HOL) of a theorem prover, such as PVS [99], or Isabelle/HOL [177]. 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 [177]. Jacobs [96] 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 [90]. 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 [97].

LOOP specifications are embedded in Java programs by way of annotations in the Java Modelling Language (JML) [111]. JML is designed to minimise 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 [98].

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 preconditions, which, if satisfied, ensure 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 [145].

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. [179] 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” [91].

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 and Lee introduced the idea of proof carrying code (PCC) [127]. 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 used with compilers for ML [127], type safe C [128] and more recently Java [45].

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 as-
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 [130]. A proof may become large because of the amount of redundancy. In a paper, Necula and Lee [129] 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 has shown that oracle based techniques allow for even smaller proofs. For a given benchmark proofs are on average 12% of the code size [131]. 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 proposal for code certification [108] 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 pro-
gram verification techniques are needed to derive strong guarantees that programs have certain desired proper-
ties. However, Program verification requires special skills, to formulate properties, to discover appropriate condi-
tions, 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 re-
quire manual intervention but generally cope better with larger problems. Unfortunately, relatively few program-
ners are trained in program verification, thus hampering the widespread use of the techniques.

A number of groups are working on program verifica-
tion techniques that are specifically adapted to Java. Us-
ing such specialised tools and techniques relieves the pro-
grammer 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 sem-
antics 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 ex-
plicit 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 [104], ASM [80], B method [5], Centaur [30], Coq, DE-
CLARE, ESC/Java [112], FDR, Haskell, Isabelle [142],
HOL, JML [111], LETOS [83], PVS [141], SMV [119],
SpecWare [162], SPIN [89]. Not all of those tools are suf-
ciently 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 [182]. This dearly 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 [39]. 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 re-
cently 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, is-

suers, and users [109].

Java Card is not plain Java, because Java implemen-
tations are too resource hungry for smart cards. A JVM
implementation requires at least 1 MB of store [187]. This makes Java acceptable for use in PCs and capacious em-
bedded 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 [169]. 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 7816-4 standard [1] 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 [120] show how the security of the Java Card object sharing model can be broken. Oestreicher [138, 139] 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 present a critique of the Java Card safety and security mechanisms [85].

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 novelities 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. [15] 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. [84] 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 [83].

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 [169] 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 [172]. 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 func-
tionality to the core VM. Unfortunately, no details are provided in the paper [133].

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 (O) 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 reference to the SIO to the client, which can then begin to invoke methods on the object. We have glossed over some details here [39], but it should be apparent that the protocol is non-trivial. A formal investigation into the properties of the protocol is thus useful.

Motré [125] 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. [22] 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.

Laet and Requet [110] 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 Laet 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 [34]. The scalability of the method, covering about 100 different byte codes, is investigated in a later paper [154]. Even though the work is “not representative of the tricky parts of the full instruction set”, the effort required is considerable.

Denney and Jensen [53] 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 [18, 146] 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 [77], 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 appears to be somewhat disappointing, as proving the theorems of some simple first order logic formulae may take of the order of minutes.

With lightweight byte code verification, Rose and Rose [156] 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 [165].

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 [170] 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 [106] 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 [65]. 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 not 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. The novel features of Java have been investigated in particular. 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 APIs 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 loaded libraries. 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 Cards 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-requisite 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.

Acknowledgements

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

References


