

Java and the Java Virtual Machine

Definition, Verification, Validation

Robert Stärk, Joachim Schmid, Egon Börger

June 7, 2001

The introduction of Jbook. This document is not for distribution.
The Home-Page of Jbook is

<http://www.inf.ethz.ch/~jbook/>

where this document and more information about Jbook is available.

1. Introduction

This book provides a structured and high-level description, together with a mathematical and an experimental analysis, of Java and of the Java Virtual Machine (JVM), including the standard compilation of Java programs to JVM code and the security critical bytecode verifier component of the JVM. The description is structured into modules (language layers and machine components), and its abstract character implies that it is truly platform-independent. It comes with a natural refinement to executable machines on which code can be tested, exploiting in particular the potential of model-based high-level testing. The analysis brings to light in what sense, and under which conditions, legal Java programs can be guaranteed to be correctly compiled, to successfully pass the bytecode verifier, and to be executed on the JVM correctly, i.e., faithfully reflecting the Java semantics and without violating any run-time checks. The method we develop for this purpose, using Abstract State Machines which one may view as code written in an abstract programming language, can be applied to other virtual machines and to other programming languages as well.

The target readers are practitioners—programmers, implementors, standardizers, lecturers, students—who need for their work a complete, correct, and at the same time transparent definition, and an executable model of the language and of the virtual machine underlying its intended implementation. As a consequence, in our models for the language and the machine, we first of all try to directly and faithfully reflect, in a complete way, as far as possible without becoming inconsistent, and in an unambiguous yet for the human reader graspable way, the intuitions and design decisions which are expressed in the reference manuals [18, 23] and underlie the current implementations of the language and the machine. We clarify various ambiguities and inconsistencies we discovered in the manuals and in the implementations, concerning fundamental notions like legal Java program, legal bytecode, verifiable bytecode, etc. Our analysis of the JVM bytecode verifier, which we relate to the static analysis of the Java parser (rules of definite assignment and reachability analysis), goes beyond the work of Stata and Abadi [34], Qian [27, 28], Freund and Mitchell [16], and O’Callahan [26].

In this introduction, we give an overview of the general goals of the book, its contents, the structuring techniques we use for decomposing Java and the JVM, and the literature we used.

For additional information on the book and updates made after its publication, see the Home Page of Jbook at <http://www.inf.ethz.ch/~jbook>.

1.1 The goals of the book

Our main goal is not to write an introduction to programming in Java or on the JVM, but to support the practitioner's correct understanding of Java programs and of what can be expected when these programs run on the virtual machine. Therefore we provide a rigorous implementation-independent (read: a mathematical) framework for the clarification of dark corners in the manuals, for the specification and evaluation of variations or extensions of the language and the virtual machine, and for the mathematical and the experimental study and comparison of present and future Java implementations. We build stepwise refined models for the language, the virtual machine, and the compiler that are abstract, but nevertheless can in a natural way be turned into executable models, which we also provide in this book, together with the necessary run-time support. As a result, our specifications of Java and the JVM are amenable to mathematical and computer-assisted verification as well as to the experimental validation of practically important properties of Java programs when executed on the JVM.

To formulate our models for Java and the JVM as consisting of components which reflect different language and security features, we use Gurevich's *Abstract State Machines* (ASMs), a form of pseudo-code, working on abstract data structures, which comes with a simple mathematical foundation [20]. The use of ASMs allowed us:

- To express the basic Java and JVM objects and operations directly, without encoding, i.e., as abstract entities and actions, at the level of abstraction in which they are best understood and analyzed by the human reader
- To uncover the modular structure which characterizes the Java language and its implementation

At the same time, one can turn ASMs in various natural ways into executable code, so that the models can be tested experimentally and validated.

With this book we also pursue a more general goal, which uses Java and the JVM only as a practically relevant and non-trivial case study. Namely, we want to illustrate that for the design and the experimental and mathematical analysis of a complex system, the ASM method is helpful for the working software system engineer and indeed scales to real-life systems.¹ Therefore

¹ For a survey of numerous other applications of the method including industrial ones, we refer the reader to [3, 4].

we also include a chapter with a textbook introduction to ASMs. We provide two versions, one written for the practitioner and the other one for the more mathematically inclined reader. We hope that the framework developed in this book shows how to make implementations of real-life complex systems amenable to rigorous high-level analysis and checkable documentation—an indispensable characteristic of every scientifically grounded engineering discipline worth its name.

The three main themes of the book, namely, definition, mathematical verification, and experimental validation of Java and the JVM, fulfill three different concerns and can be dealt with separately. The *definition* has to provide a natural understanding of Java programs and of their execution on the JVM, which can be justified as representing a faithful “ground model” of the intentions of the reference manuals, although our models disambiguate and complete them and make them coherent, where necessary. The *verification* has to clarify and to prove under which assumptions, and in which sense, the relevant design properties can be guaranteed, e.g., in this case, the type safety of syntactically well-formed Java programs, the correctness of their compilation, the soundness and completeness of the bytecode verifier, etc. The *validation* of (a refinement of the ground model to) an executable model serves to provide experimental tests of the models for programs. However, as should become clear through this book, using the ASM framework, these three concerns, namely, abstract specification, its verification, and its validation, can be combined as intimately and coherently connected parts of a rigorous yet practical approach to carrying out a real-life design and implementation project, providing objectively checkable definitions, claims, and justifications. It is a crucial feature of the method that, although abstract, it is run-time oriented. This is indispensable if one wants to come up with formulating precise and reliably implementable conditions on what “auditing” secure systems [21] may mean.

It is also crucial for the practicality of the approach that by exploiting the abstraction and refinement capabilities of ASMs, one can layer complex systems, like Java and the JVM, into several natural strata, each responsible for different aspects of system execution and of its safety, so that in the models one can study their functionality, both in isolation and when they are interacting (see the explanations below).

1.2 The contents of the book

Using an ASM-based modularization technique explained in the next section, we define a structured sequence of mathematical models for the statics and the dynamics of the programming language Java (Part I) and for the Java Virtual Machine, covering the compilation of Java programs to JVM code (Part II) and the JVM bytecode verifier (Part III). The definitions clarify some dark corners in the official descriptions in [18, 23]:

- Bytecode verification is not possible the way the manuals suggest (Fig. 16.8, Fig. 16.9, Remark 8.3.1, Remark 16.5.1, bug no. 4381996 in [14])
- A valid Java program rejected by the verifier (Fig. 16.7, bug no. 4268120 in [14])
- Verifier must use sets of, instead of single, reference types (Sect. 16.1.2, Fig. 16.10)
- Inconsistent treatment of recursive subroutines (Fig. 16.6)
- Verifier has problems with array element types (Example C.7.1)
- Inconsistent method resolution (Example 5.1.4, bug no. 4279316 in [14])
- Compilation of boolean expressions due to the incompatibility of the reachability notions for Java and for JVM code (Example 16.5.4)
- Unfortunate entanglement of embedded subroutines and object initialization (Fig. 16.19, Fig. 16.20)
- Initialization problems [10]

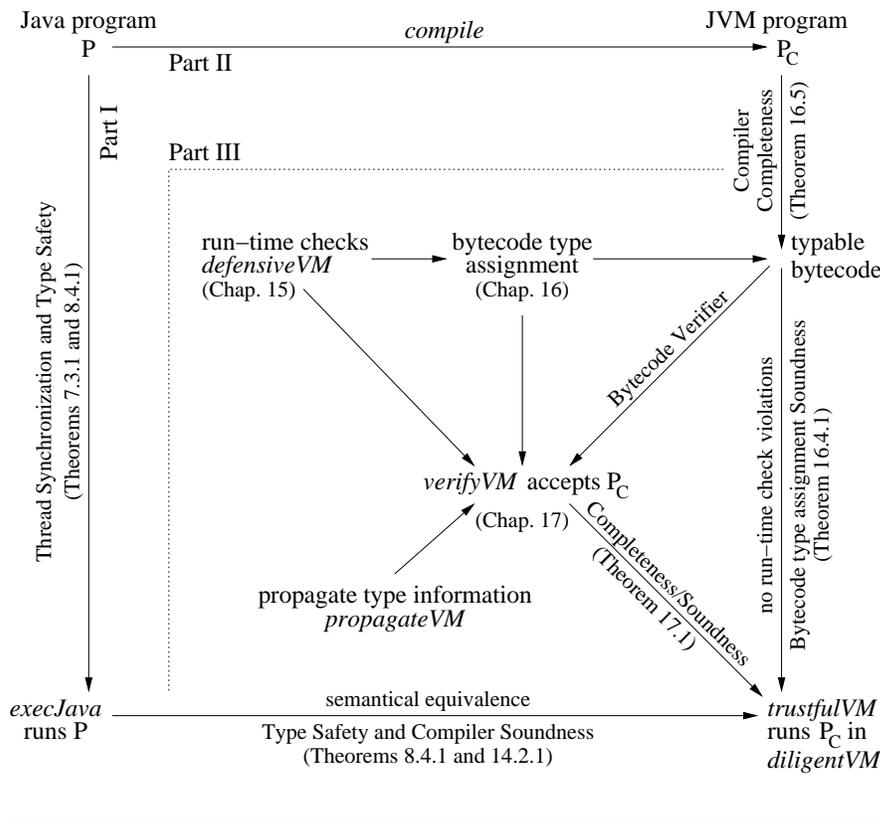
We formulate and prove some of the basic correctness and safety properties, which are claimed for Java and the JVM as a safe and secure, platform-independent, programming environment for the internet. The safety of Java programs does not rely upon the operating system. The implementation compiles Java programs to bytecode which is loaded and verified by the JVM and then executed by the JVM interpreter, letting the JVM control the access to all resources. To the traditional correctness problems for the interpretation and the compilation of programs,² this strategy adds some new correctness problems, namely, for the following JVM components (see Fig. 1.4):

- The loading mechanism which dynamically loads classes; the binary representation of a class is retrieved and installed within the JVM—relying upon some appropriate name space definition to be used by the security manager—and then prepared for execution by the JVM interpreter
- The bytecode verifier, which checks certain code properties at link-time, e.g. conditions on types and on stack bounds which one wants to be satisfied at run-time
- The access right checker, i.e., a security manager which controls the access to the file system, to network addresses, to critical windowing operations, etc.

As is well known (see [21]), many Java implementation errors have been found in the complex interplay between the JVM class loader, the bytecode verifier, and the run-time system.

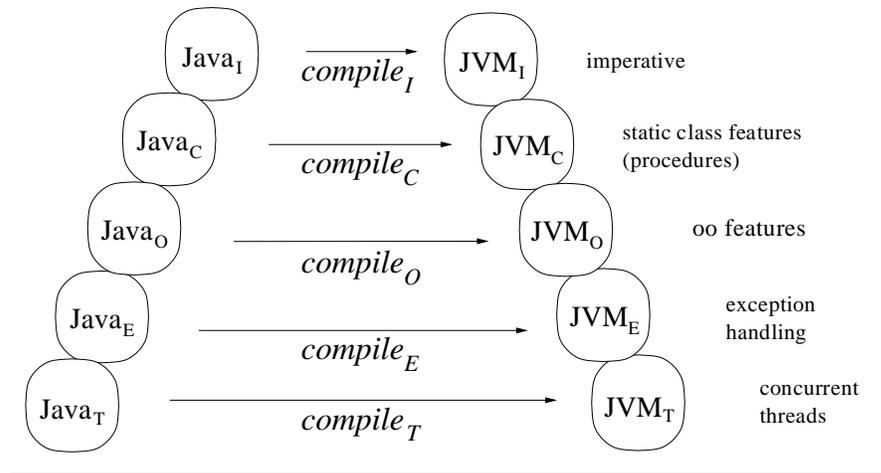
We show under what assumptions Java programs can be proved to be type safe (Theorem 8.4.1), and successfully verified (Theorem 16.5.2 and Theorem 17.1.2) and correctly executed when correctly compiled to JVM code (Theorem 14.1.1). The most difficult part of this endeavor is the rigorous

² See [5, 6] where ASMs have been used to prove the correctness of the compilation of PROLOG programs to WAM code and of imperative (OCCAM) programs with non-determinism and parallelism to Transputer code.

Fig. 1.1 Dependency Graph

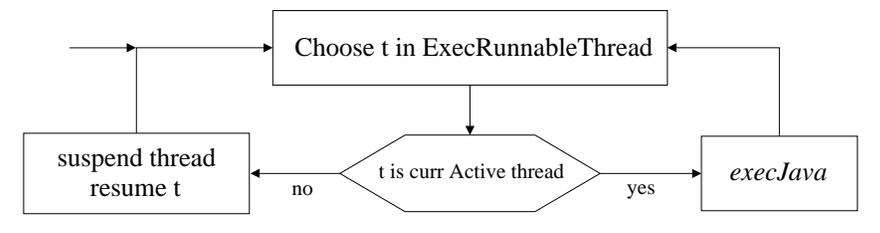
definition and verification of the bytecode verifier, which is a core part of the JVM. We define a novel bytecode verifier for which we can prove soundness (Theorem 17.1.1) and completeness (Theorem 17.1.2). We also prove that successfully verified bytecode is guaranteed to execute without violating any run-time checks (Theorem 16.4.1). We also prove the soundness of Java’s thread synchronization (Theorem 7.3.1). Figure 1.1 shows how the theorems and the three parts of this book fit together. We hope that the proofs will provide useful insight into the design of the implementation of Java on the JVM. They may guide possible machine verifications of the reasoning which supports them, the way the WAM correctness proof for the compilation of Prolog programs, which has been formulated in terms of ASMs in [6], has been machine verified in [31].

Last but not least we provide experimental support for our analysis, namely, by the validation of the models in their AsmGofer executable form. Since the executable AsmGofer specifications are mechanically transformed

Fig. 1.2 Language oriented decomposition of Java/JVM

into the \LaTeX code for the numerous models which appear in the text, the correspondence between these specifications is no longer disrupted by any manual translation. AsmGofer (see Appendix A) is an ASM programming system developed by Joachim Schmid, on the suggestion and with the initial help of Wolfram Schulte, extending TkGofer to execute ASMs which come with Haskell definable external functions. It provides a step-by-step execution of ASMs, in particular of Java/JVM programs on our Java/JVM machines, with GUIs to support debugging. The appendix which accompanies the book contains an introduction to the three graphical AsmGofer user interfaces: for Java, for the compiler from Java to bytecode, and for the JVM. The Java GUI offers debugger features and can be used to observe the behavior of Java programs during their execution. As a result, the reader can run experiments by executing Java programs on our Java machine, compiling them to bytecode and executing that bytecode on our JVM machine. For example, it can be checked that our Bytecode Verifier rejects the program found by Saraswat [30].

The CD contains the entire text of the book, numerous examples and exercises which support using the book for teaching, the sources of the executable models, and the source code for AsmGofer together with installation instructions (and also precompiled binaries of AsmGofer for several popular operating systems like Linux and Windows). The examples and exercises in the book which are provided by the CD are marked with \rightsquigarrow CD. The executable models also contain the treatment of strings which are needed to run interesting examples.

Fig. 1.3 Multiple thread Java machine `execJavaThread`

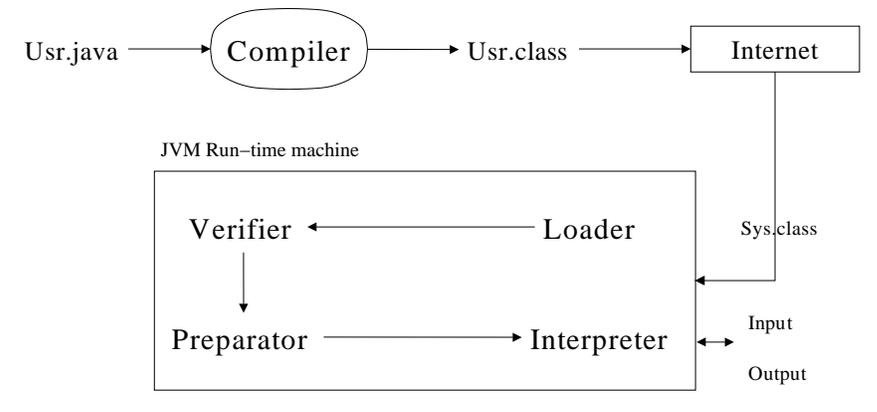
1.3 Decomposing Java and the JVM

We decompose Java and the JVM into language layers and security modules, thus splitting the overall definition and verification problem into a series of tractable subproblems. This is technically supported by the abstraction and refinement capabilities of ASMs. As a result we succeed

- To reveal the structure of the language and the virtual machine
- To control the size of the models and of the definition of the compilation scheme, which relates them
- To keep the effort of writing and understanding the proofs and the executable models, manageable

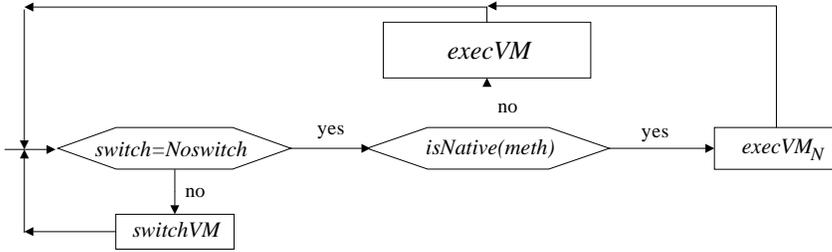
The first layering principle reflects the structure of the Java language and of the set of JVM instructions. In Part I and Part II we factor the sets of Java and of JVM instructions into five sublanguages, by isolating language features which represent milestones in the evolution of modern programming languages and of the techniques for their compilation, namely imperative (sequential control), procedural (module), object-oriented, exception handling, and concurrency features. We illustrate this in Fig. 1.2. A related structuring principle, which helps us to keep the size of the models small, consists in grouping similar instructions into one abstract instruction each, coming with appropriate parameters. This goes without leaving out any relevant language feature, given that the specializations can be regained by mere parameter expansion, a refinement step whose correctness is easily controllable instruction-wise. See Appendix C.8 for a correspondence table between our abstract JVM instructions and the real bytecode instructions.

This decomposition can be made in such a way that in the resulting sequence of machines, namely $\text{Java}_{\mathcal{I}}$, $\text{Java}_{\mathcal{C}}$, $\text{Java}_{\mathcal{O}}$, $\text{Java}_{\mathcal{E}}$, $\text{Java}_{\mathcal{T}}$ and $\text{JVM}_{\mathcal{I}}$, $\text{JVM}_{\mathcal{C}}$, $\text{JVM}_{\mathcal{O}}$, $\text{JVM}_{\mathcal{E}}$, $\text{JVM}_{\mathcal{N}}$, each ASM is a purely incremental—similar to what logicians call a conservative—extension of its predecessor, because each of them provides the semantics of the underlying language instruction by instruction. The general compilation scheme *compile* can then be defined between the corresponding submachines by a simple recursion.

Fig. 1.4 Security oriented decomposition of the JVM

Functionally we follow a well known pattern and separate the treatment of parsing, elaboration, and execution of Java programs. We describe how our Java machines, which represent abstract interpreters for arbitrary programs in the corresponding sublanguage, are supposed to receive these input programs in the form of abstract syntax trees resulting from parsing. For each Java submachine we describe separately, in Part I, the static and the dynamic part of the program semantics. We formulate the relevant static constraints of being well-formed and well-typed, which are checked during the program elaboration phase and result in corresponding annotations in the abstract syntax tree. In the main text of the book we restrict the analysis of the static constraints to what is necessary for a correct understanding of the language and for the proofs in this book. The remaining details appear in the executable version of the Java model. We formalize the dynamical program behavior by ASM transition rules, describing how the program runtime state changes through evaluating expressions and executing statements. This model allows us to rigorously define what it means for Java to be type safe, and to prove that well-formed and well-typed Java programs are indeed type safe (Theorem 8.4.1). This includes defining rules which achieve the definite assignment of variables, and to prove the soundness of such assignments. The resulting one-thread model *execJava* can be used to build a multiple-thread executable ASM *execJavaThread* which reflects the intention of [18, 23], namely to leave the specification of the particular implementation of the scheduling strategy open, by using a choice that is not further specified function (Fig. 1.3)³. For this model we can prove a correctness theorem for thread synchronization (Theorem 7.3.1).

³ The flowchart notation we use in this introduction has the expected precise meaning, see Chapter 2, so that these diagrams provide a rigorous definition, namely of so called control state ASMs.

Fig. 1.5 Decomposing trustfulVMs into execVMs and switchVMs

$$trustfulVM = execVM_I \cup execVM_C \cup execVM_O \cup execVM_E \cup execVM_N \cup execVM_D$$

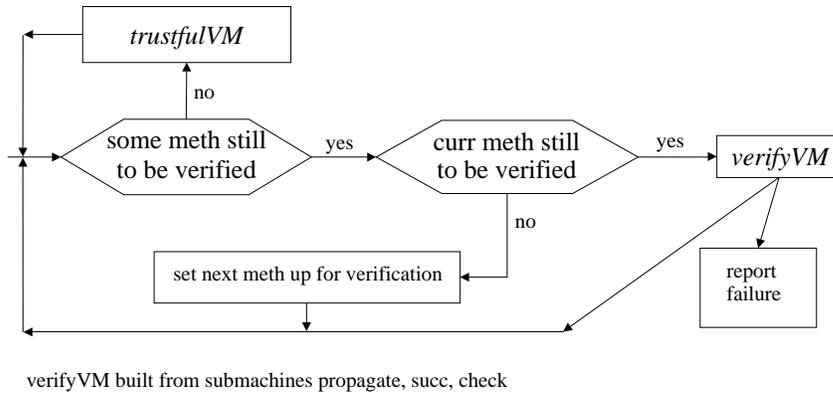
$$switchVM_D \text{ extends } switchVM_E \text{ extends } switchVM_C$$

For JVM programs, we separate the modeling of the security relevant loading (Chapter 18) and linking (i.e., preparation and verification, see Part III) from each other and from the execution (Part II), as illustrated in Fig. 1.4.

In Part II we describe the *trustful execution* of bytecode which is assumed to be successfully loaded and linked (i.e., prepared and verified to satisfy the required link-time constraints). The resulting sequence of stepwise refined trustful VMs, namely $trustfulVM_I$, $trustfulVM_C$, $trustfulVM_O$, $trustfulVM_E$, and $trustfulVM_N$, yields a succinct definition of the functionality of JVM execution in terms of language layered submachines $execVM$ and $switchVM$ (Fig. 1.5). The machine $execVM$ describes the effect of each single JVM instruction on the current frame, whereas $switchVM$ is responsible for frame stack manipulations upon method call and return, class initialization and exception capture. The machines do nothing when no instruction remains to be executed. As stated above, this piecemeal description of single Java/JVM instructions yields a simple recursive definition of a general compilation scheme for Java programs to JVM code, which allows us to incrementally prove it to be correct (see Chapter 14). This includes a correctness proof for the handling of Java exceptions in the JVM, a feature which considerably complicates the bytecode verification, in the presence of embedded subroutines, class and object initialization and concurrently working threads.

In Chapter 17 we insert this trustfully executing machine into a *diligent* JVM which, after loading the bytecode, which is stored in class files, and before executing it using the trustfully executing component $trustfulVM$, prepares and verifies the code for all methods in that class file, using a sub-machine $verifyVM$ which checks, one after the other, each method body to satisfy the required type and stack bound constraints (Fig. 1.6).

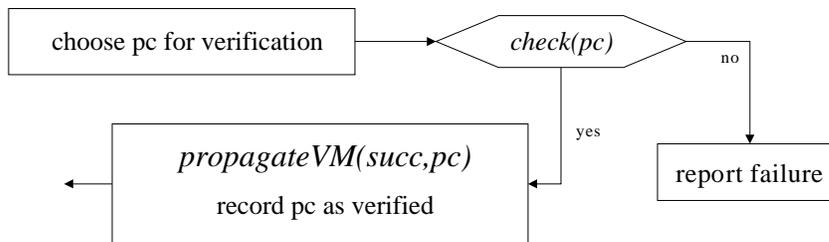
The machine $verifyVM$ is language layered, like $trustfulVM$, since it is built from a language layered submachine $propagateVM$, a language layered

Fig. 1.6 Decomposing diligent JVMs into trustfulVMs and verifyVMs

predicate *check* and a language layered function *succ*. The verifier machine chooses an instruction among those which are still to be verified, checks whether it satisfies the required constraints and either reports failure or propagates the result of the checked conditions to the successor instructions (Fig. 1.7).

The submachine *propagateVM*, together with the function *succ* in the verifying submachine *verifyVM*, defines a link-time simulation (type version) of the *trustful* VM of Part II, although the checking functionality can be better defined in terms of a run-time checking machine, see Chapter 15. The *defensive* VM we describe there, which is inspired by the work of Cohen [13], defines what to check for each JVM instruction at run-time, before its trustful execution. We formulate the constraints about types, resource bounds, references to heap objects, etc., which are required to be satisfied when the given instruction is executed (Fig. 1.8).

The reason for introducing this machine is to obtain a well motivated and clear definition of the bytecode verification functionality, a task which is best accomplished locally, in terms of run-time checks of the safe executability of single instructions. However, we formulate these run-time checking conditions referring to the types of values, instead of the values themselves, so that we can easily lift them to link-time checkable bytecode type assignments (see Chapter 16). When lifting the run-time constraints, we make sure that if a given bytecode has a type assignment, this implies that the code runs on the defensive VM without violating any run-time checks, as we can indeed prove in Theorem 16.4.1. The notion of bytecode type assignment also allows us to prove the completeness of the compilation scheme defined in Part II. Completeness here means that bytecode which is compiled from a well-formed and well-typed Java program (in a way which respects our compilation scheme), can be typed successfully, in the sense that it does have type assignments

Fig. 1.7 Decomposing verifyVMs into propagateVMs, checks, succs

$succ_I \subset succ_C \subset succ_O \subset succ_E$ and $propagate_I \subset propagate_E$

(Theorem 16.5.2). To support the inductive proof for this theorem we refine our compiler to a certifying code generator, which issues instructions together with the type information needed for the bytecode verification.

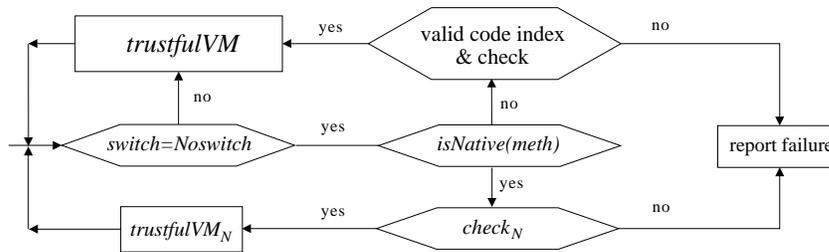
The details of the machines outlined above are explained in this book and are summarized in appendices B and C. Putting together the properties of the language layered submachines and of the security components of Java and of the JVM, one obtains a precise yet graspable statement, and an understandable (and therefore checkable) proof of the following property of Java and the JVM.

Main Theorem. Under explicitly stated conditions, any well-formed and well-typed Java program, when correctly compiled, passes the verifier and is executed on the JVM. It executes without violating any run-time checks, and is correct with respect to the expected behavior as defined by the Java machine.

For the executable versions of our machines, the formats for inputting and compiling Java programs are chosen in such a way that the ASMs for the JVM and the compiler can be combined in various ways with current implementations of Java compilers and of the JVM (see Appendix A and in particular Fig. A.1 for the details).

1.4 Sources and literature

This book is largely self-contained and presupposes only basic knowledge in object-oriented programming and about the implementation of high-level programming languages. It uses ASMs, which have a simple mathematical foundation justifying their intuitive understanding as “pseudo-code over abstract data”, so that the reader can understand them correctly and successfully without having to go through any preliminary reading. We therefore

Fig. 1.8 Decomposing defensive VMs into trustful VMs and checks

$check_D$ extends $check_N$ extends $check_E$ extends $check_O$ extends $check_C$ extends $check_I$

invite the reader to consult the formal definition of ASMs in Chapter 2 only should the necessity be felt.

The Java/JVM models in this book are completely revised—streamlined, extended and in some points corrected—versions of the models which appeared in [9, 11]. The original models were based upon the first edition of the Java and JVM specifications [18, 23], and also the models in this book still largely reflect our interpretation of the original scheme. In particular we do not treat nested and inner classes which appear in the second edition of the Java specification, which was published when the work on this book was finished. It should be noted however that the revision of [23], which appeared in 1999 in the appendix of the second edition of the JVM specification, clarifies most of the ambiguities, errors and omissions that were reported in [10].

The proofs of the theorems were developed for this book by Robert Stärk and Egon Börger, starting from the proof idea formulated for the compiler correctness theorem in [8], from its elaboration in [33] and from the proof for the correctness of exception handling in [12]. The novel subroutine call stack free bytecode verifier was developed by Robert Stärk and Joachim Schmid. Robert Stärk constructed the proof for Theorem 16.5.2 that this verifier accepts every legal Java program which is compiled respecting our compilation scheme. The AsmGofer executable versions of the models were developed for this book by Joachim Schmid and contributed considerably towards getting the models correct.

We can point the reader to a recent survey [21] of the rich literature on modeling and analyzing safety aspects of Java and the JVM. Therefore we limit ourselves to citing in this book only a few sources which had a direct impact on our own work. As stated above, the complex scheme to implement Java security through the JVM interpreter requires a class loader, a security manager and a bytecode verifier. For a detailed analysis of the class loading mechanism, which is underspecified in [18] and therefore only sketched in this book, we refer the reader to [29, 35] where also further references on this still widely open subject can be found. We hope that somebody will use and

extend our models for a complete analysis of the critical security features of Java, since the framework allows to precisely state and study the necessary system safety and security properties; the extensive literature devoted to this theme is reviewed in [21].

Draft chapters of the book have been used by Robert Stärk in his summer term 2000 course at ETH Zürich, and by Egon Börger in his *Specification Methods* course in Pisa in the fall of 2000.

References

1. K. Achatz and W. Schulte. A formal OO method inspired by Fusion and Object-Z. In J. P. Bowen, M. G. Hinchey, and D. Till, editors, *ZUM'97: The Z Formal Specification*, number 1212 in Lecture Notes in Computer Science, pages 92–111. Springer-Verlag, 1997.
2. J. Alves-Foss, editor. *Formal Syntax and Semantics of Java(tm)*. Number 1523 in Lecture Notes in Computer Science. Springer-Verlag, 1999.
3. E. Börger. High level system design and analysis using Abstract State Machines. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Current Trends in Applied Formal Methods (FM-Trends 98)*, number 1641 in Lecture Notes in Computer Science, pages 1–43. Springer-Verlag, 1999.
4. E. Börger. Abstract state machines at the cusp of the millenium. In *Abstract State Machines ASM'2000*, number 1912 in Lecture Notes in Computer Science. Springer-Verlag, 2000.
5. E. Börger and I. Durdanovic. Correctness of compiling Occam to transputer code. *The Computer Journal*, 39:52–92, 1996.
6. E. Börger and D. Rosenzweig. The WAM—definition and compiler correctness. In L. Plümer C. Beierle, editor, *Logic Programming: Formal Methods and Practical Applications*, pages 20–90. Elsevier Science B.V./North-Holland, 1995.
7. E. Börger and J. Schmid. Composition and submachine concepts. In P. G. Clote and H. Schwichtenberg, editors, *Computer Science Logic (CSL 2000)*, number 1862 in Lecture Notes in Computer Science, pages 41–60. Springer-Verlag, 2000.
8. E. Börger and W. Schulte. Defining the Java Virtual Machine as platform for provably correct Java compilation. In L. Brim, J. Gruska, and J. Zlatuska, editors, *MFCS'98*, number 1450 in Lecture Notes in Computer Science, pages 17–35. Springer-Verlag, 1998.
9. E. Börger and W. Schulte. A programmer friendly modular definition of the semantics of Java. In Alves-Foss [2], pages 353–404. Extended Abstract in: R. Berghammer and F. Simon, editors, *Programming Languages and Fundamentals of Programming*, University of Kiel (Germany) TR 9717, 1997, pages 175–181.
10. E. Börger and W. Schulte. Initialization problems for Java. *Software – Principles and Tools*, 19(4):175–178, 2000.
11. E. Börger and W. Schulte. Modular design for the Java Virtual Machine architecture. In E. Börger, editor, *Architecture Design and Validation Methods*, pages 297–357. Springer-Verlag, 2000.
12. E. Börger and W. Schulte. A practical method for specification and analysis of exception handling — a Java JVM case study. *IEEE Transactions of Software Engineering*, 26(10), 2000.
13. R. M. Cohen. Defensive Java Virtual Machine version 0.5 alpha release. <http://www.cli.com/software/djvm/>, 1997.

14. Java Developer Connection. Bug parade, 1999. <http://developer.java.sun.com/developer/bugParade/>.
15. M. Dahm. JavaClass. Technical report, FU Berlin, 2000. <http://www.inf.fu-berlin.de/~dahm/JavaClass/>.
16. S. N. Freund and J. C. Mitchell. Specification and verification of Java bytecode subroutines and exceptions. Technical Report CS-TN-99-91, Stanford University, 1999.
17. S. N. Freund and J. C. Mitchell. The type system for object initialization in the Java bytecode language. *ACM Transactions on Programming Languages and Systems*, 21(6):1196–1250, 1999.
18. J. Gosling, B. Joy, and G. Steele. *The Java(tm) Language Specification*. Addison Wesley, 1996.
19. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java(tm) Language Specification*. Addison Wesley, second edition, 2000.
20. Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
21. P. H. Hartel and L. Moreau. Formalising the safety of Java, the Java Virtual Machine and Java Card. 2001. Submitted to ACM Computing Surveys.
22. M. P. Jones. Gofer distribution 2.30, 1993. <http://www.cse.ogi.edu/~mpj/goferarc/>.
23. T. Lindholm and F. Yellin. *The Java(tm) Virtual Machine Specification*. Addison Wesley, second edition, 1999.
24. J. Meyer and T. Downing. *Java Virtual Machine*. O’Reilly & Associates, Inc., 1997.
25. Sun Microsystems. Connected, limited device configuration, specification 1.0, Java 2 platform micro edition, 2000.
26. R. O’Callahan. A simple, comprehensive type system for Java bytecode subroutines. In *POPL ’99. Proc. 26th ACM Symposium on Principles of Programming Languages*, pages 70–78, 1999.
27. Z. Qian. A formal specification of Java(tm) Virtual Machine for objects, methods and subroutines. In Alves-Foss [2], pages 271–311.
28. Z. Qian. Standard fixpoint iteration for Java bytecode verification. *ACM Transactions on Programming Languages and Systems*, 200? To appear.
29. Z. Qian, A. Goldberg, and A. Coglio. A formal specification of Java class loading. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Application (OOPSLA-00)*, volume 35 of *ACM Sigplan Notices*, pages 325–336. ACM Press, 2000.
30. V. Saraswat. Java is not type-safe. Technical report, AT&T, Research, 1997. <http://www.research.att.com/~vj/bug.html>.
31. G. Schellhorn. *Verifikation abstrakter Zustandsmaschinen*. PhD thesis, University of Ulm, 1999. For an English version see <http://www.informatik.uni-ulm.de/pm/kiv/papers/verif-asms-english.ps.gz>.
32. J. Schmid. Executing ASM specifications with AsmGofer, 1999. Web pages at <http://www.tydo.de/AsmGofer>.
33. R. Stärk. Formal foundations of Java. Course notes, University of Fribourg, 1999.
34. R. Stata and M. Abadi. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems*, 21(1):90–137, 1999.
35. A. Tozawa and M. Hagya. Formalization and analysis of class loading in Java. *Higher Order and Symbolic Computation*, 200? To appear.
36. T. Vullinghs, W. Schulte, and T. Schwinn. An introduction to TkGofer, 1996. Web pages at <http://pplab.kaist.ac.kr/seminar/haha/tkgofer2.0-html/user.html>.