Design for Reuse via Structuring Techniques for ASMs

Egon Börger

Università di Pisa, Dipartimento di Informatica, I-56125 Pisa, Italy
boerger@di.unipi.it

Abstract. Gurevich's [26] Abstract State Machines (ASMs), characterized by the parallel execution of abstract atomic actions in a global state, have been equipped in [13] with a refinement by standard composition concepts for structuring large machines that allows reusing machine components. Among these concepts are parameterized (possibly recursive) sub-ASMs. Here we illustrate their power for incremental and modular system design by unfolding, via appropriate ASM components, the architecture of the Java Virtual Machine (JVM), resulting from the language layering in combination with the functional decomposition of the JVM into loader, verifier, and interpreter. We survey the ASM models for Java and the JVM that appear in [34], together with the mathematical and experimental analysis they support.

1 The Method: Structuring ASMs by Submachines

Although it was by a foundational concern, namely of reconsidering Turing's thesis in the light of the problem of the semantics of computer programs, that Gurevich was led to formulate the idea of Abstract State Machines[1], it did not take a long time that the concept was recognized to be of practical importance. ASMs were soon successfully applied for the modeling and a rigorous analysis of a variety of complex real-life computing systems: programming languages and their implementations, processor architectures, protocols, embedded software, etc., see [5, 6] for a historical account. The first industrial application showed up as early as 1990 in the ASM model defining the semantics of PROLOG [2, 3, 10], which became the official ISO standard [28] and has been run for experimentation at Quintus[2], see [4] for a survey of these early applications of ASMs in the context of logic programming. By now a powerful method has been built around the

[1] In embryo the notion appeared under the name of dynamic/evolving structures/algebras in a Technical Report in 1984 [22]; a year later in a note to the American Mathematical Society [23]; I learnt it in the Spring of 1987 from the simple examples which appeared later in [24] to illustrate the concept, see [6] for a more detailed historical account. The first complete definition, which essentially remained stable since then, appeared in [26] and in a preliminary form in [25].

[2] Before, in the summer of 1990 in a diploma thesis at the University of Dortmund [30], Angelika Kappel had developed the first tool to make such ASMs and in particular that abstract PROLOG machine executable.

concept of ASM, which supports industrial system design by rigorous high-level modeling that is seamlessly linked to executable code, namely by mathematically verifiable, experimentally validatable, and objectively documentable refinement steps. Here are some highlights:

- The reengineering of a central component in a large software package for constructing and validating timetables for railway systems, work done at Siemens from May 1998 to March 1999. A high-level ASM model for the component was built, compiled to C++ and successfully integrated into the existing software system which since then is in operation at Vienna subways [14]
- The ASM definition of the International Telecommunication Union standard for SDL2000 [29]
- The investigation (verification and validation) of Java and its implementation by the Java Virtual Machine in terms of ASM models and their AsmGofer executable refinements for the language and the VM [34]
- The recent ASM model for the UPnP architecture at Microsoft [15]

For the impressive up-to-date list of annotated references to ASM publications and tools the reader may consult the ASM website [27].

One of the reasons for the simplicity of Gurevich's notion of Abstract State Machine—its mathematical content can be explained in less than an hour, see Chapter 2 of [34] for a textbook definition starting from scratch—lies in the fact that its definition uses only conditional assignments, so-called *rules* of form

$$\textbf{if } \mathit{Condition} \textbf{ then } f(t_1, \ldots, t_n) := t$$

expressing guarded atomic actions that yield updates in a well-defined (a global) state. In this respect ASMs are similar to Abrial's Abstract Machines [1] that are expressed by non-executable pseudo-code without sequencing or loop (Abstract Machine Notation, AMN). It is true that this leaves the freedom—so necessary for *high-level* system design and analysis—to introduce during the modeling process any control or data structure whatsoever that may turn out to be suitable for the application under study. However, the other side of the coin is that this forces the designer to specify standard control or data structures and standard component based design structures over and over again, namely when it comes to implement the specifications, thus making effective reuse difficult. For some time it was felt as a challenge to combine, in a practically viable manner, the simplicity of the parallel execution model of atomic actions in a global state with the structuring capabilities of modules and components as part of a large system architecture, whose execution implies duration and scheduling.

In [13] a solution has been developed that naturally extends the characteristic ASM notion of synchronous parallel execution of multiple atomic actions (read: rules) by allowing as rules also calling and execution of submachines, technically speaking *named, parameterized*, possibly recursive, ASMs. This definition gently embeds the result of executing an a priori unlimited number n of *micro steps*—namely steps of a submachine that has been called for execution in a given state—into the *macro step* semantics of the calling ASM, which is defined as the overall result of the simultaneous execution of all its rules in the given state. The same treatment covers also the classical control constructs for *sequentialization* and *iteration*[3] and opens the way to structuring large ASMs by making use of instantiatable machine components. Whereas for the AMN of the B method Abrial explicitly excludes e.g. sequencing and loop from the specification of abstract machines [1, pg. 373], we took a more pragmatic approach and defined these control constructs, and more generally the notion of ASM submachine in such a way that they can be used coherently in two ways, depending on what is needed, namely to provide black-box descriptions of the behavior of components or glass-box views of their implementation (refinement).

In the present survey we illustrate that this notion of submachines, which has been implemented in AsmGofer [33][4], suffices for a hierarchical decomposition of the Java Virtual Machine into components for the loader, the verifier, and the interpreter, each of them split into subcomponents for the five principal language layers (imperative core, static classes, object oriented features, exception handling and concurrency). We can do this in such a way that adding a component corresponds to what in logic is called extending a theory conservatively. This incremental design approach is the basis for a transparent yet far reaching mathematical analysis of Java and its implementation on the JVM (correctness and completeness proofs for the compilation, the bytecode verification, and the execution, i.e. interpretation), which appears in [34].

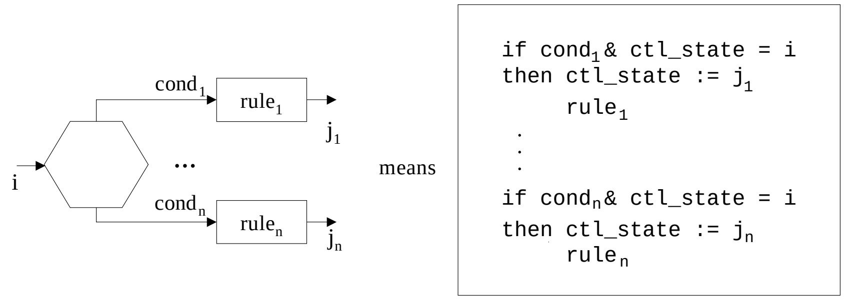
Graphical notation. Before we proceed in the next section to explain the problem of a mathematically transparent model for Java and its implementation on the JVM, and the solution offered in [34], we review here the basic graphical (UML like) notation we will use for defining structured ASMs. To describe the overall structure of the JVM we only need special ASMs that resemble the classical Finite State Machines (FSMs) in that their execution is governed by a set of internal or control states (often also called *modes*) which split the machine into finitely many submachines. Formally these ASMs, which I have called *control state* ASMs in [5], are defined and pictorially depicted as shown in Fig. 1, with *transition rules* of form

$$\textbf{if } \mathit{Condition} \textbf{ then } f(t_1, \ldots, t_n) := t$$

whose execution is to be understood as *changing* (or defining, if there was none) the value of the function f at the given parameters. Note that in a given control state i, these machines do nothing when no condition $cond_j$ is satisfied.

[3] The atomicity of this ASM iteration constructor is the key for a rigorous definition of the semantics of event triggered exiting from compound actions of UML activity and state machine diagrams, where the intended instantaneous effect of exiting has to be combined with the request to exit nested diagrams sequentially following the subdiagram order, see [7, 8].

[4] In [13] we also incorporate into standard ASMs a syntax oriented form of information hiding, namely through the notion of *local machine state*, of machines with *return values* and of *error handling* machines.



Assume disjoint cond_i. Usually the "control states" are notationally suppressed.

Fig. 1. Control state ASM diagrams

The notion of ASM *states*, differently from FSMs, is the classical notion of mathematical *structures* where data come as abstract objects, i.e., as elements of sets (domains, one for each category of data) that are equipped with basic operations (partial *functions*) and predicates (attributes or relations). The notion of ASM *run* is the classical notion of computation of transition systems. An ASM computation step in a given state consists in executing *simultaneously* all updates of all transition rules whose guard is true in the state, if these updates are consistent.

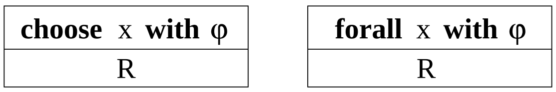
The synchronous parallelism inherent in the simultaneous execution of all ASM rules is enhanced by the following concise notation for the simultaneous execution of an ASM rule R for each x satisfying a given condition ϕ:

forall x **with** ϕ **do** R

A frequently encountered kind of functions whose detailed specification is left open are choice functions, used to abstract from details of static or dynamic scheduling strategies. ASMs support the following concise notation for an abstract specification of such strategies:

choose x **with** ϕ **do** R

meaning to execute rule R with an arbitrary x chosen among those satisfying the selection property ϕ. If there exists no such x, nothing is done. For **choose** and **forall** rules we also use graphical notations of the following form:



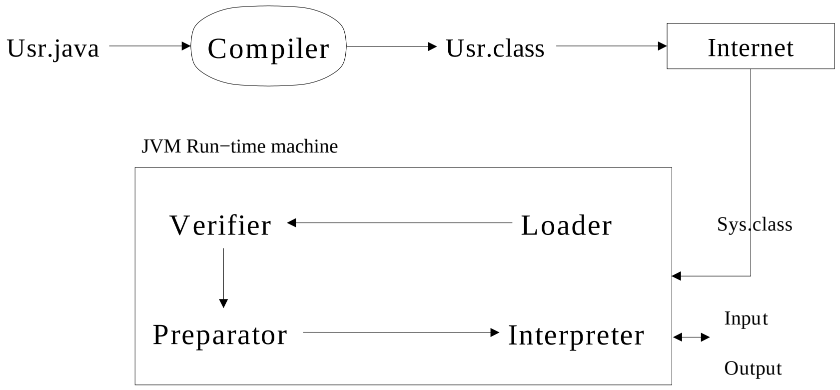


Fig. 2. Security oriented decomposition of the JVM

2 The Java/JVM Modeling and Analysis Problem

The scientific problem to solve was to investigate in which sense and to what extent one can provide a rigorous justification of the claim that Java and the JVM provide a safe and secure, platform independent programming environment for the internet. This claim goes beyond the traditional correctness problem for language compilation and the interpretation of the compiled code on a virtual or real machine, a classical problem which has been studied extensively for other source code languages and compiler target machines, including some work where ASMs are used as modeling device (e.g. [12, 9, 18, 19]. Not only is the problem of trusted (i.e. fully correct) realistic compilation not yet solved (see [16, 17] for a thorough discussion), the case of Java and its implementation on the JVM adds further problems, partly due to the fact that the access to resources by the executed code is controlled not by the operating system, but by the JVM that interprets this code, namely dynamically loaded and verified bytecode. As a result one has at least three new correctness and completeness problems, as illustrated in Fig. 2, namely concerning:

- 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.

The goal of the project was to provide an abstract (read: platform independent), rigorous but transparent, modular definition of Java and the JVM that can be used as a basis for a mathematical and an experimental analysis of the above claim. First of all this modeling work should reflect SUN's design decisions, it should provide for the two manuals [20, 21, 31] what in [5, 11] has been called a *ground model*, i.e. a sufficiently rigorous and complete, provably consistent, mathematical model that faithfully represents the given natural language descriptions. Secondly it should offer a correct high-level understanding of

- the source language, to be practically useful for Java programmers,
- the virtual machine, to offer the implementors a rigorous, implementation independent basis for the documentation, the analysis, and the comparison of implementations.

We tried to achieve the goal by constructing stepwise refined ASM models of Java, the JVM (including the loader and the bytecode verifier), and a Java-to-JVM compiler, which are abstract, but nevertheless can in a natural way be turned into executable validatable models, and for which we can prove the following theorem.

> **Main Theorem.** Under conditions that are explicitly stated in [34], any well-formed and well-typed Java program, when compiled satisfying the properties listed for the compiler, passes the bytecode verifier and is executed on the JVM. During this execution, none of the run-time checks of the properties that have been analyzed by the verifier is violated, and the generated bytecode is interpreted correctly with respect to the expected source code behavior as defined by the Java ASM.

In the course of proving the theorem, we were led to clarify various ambiguities and inconsistencies we discovered in the Java/JVM 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), led us to define a novel (subroutine call stack free) bytecode verifier which goes beyond previous work in the literature.

In the next section we explain the dependency graph which surveys how we split the proof of the main theorem in subproofs for the JVM components.

3 Decomposition of Java/JVM into Components

To make such a complex modeling and analysis problem tractable one has to split it into a series of manageable subproblems. To this end we construct the ASM for the JVM out of submachines for its security relevant components—the ones which appear in Fig. 2: loader, verifier, preparator, interpreter—and define each component incrementally via a series of submachines, put together by parallel composition and forming a sequence of conservative extensions, which is guided by the layering of Java and of the set of JVM instructions into increasingly richer sublanguages.

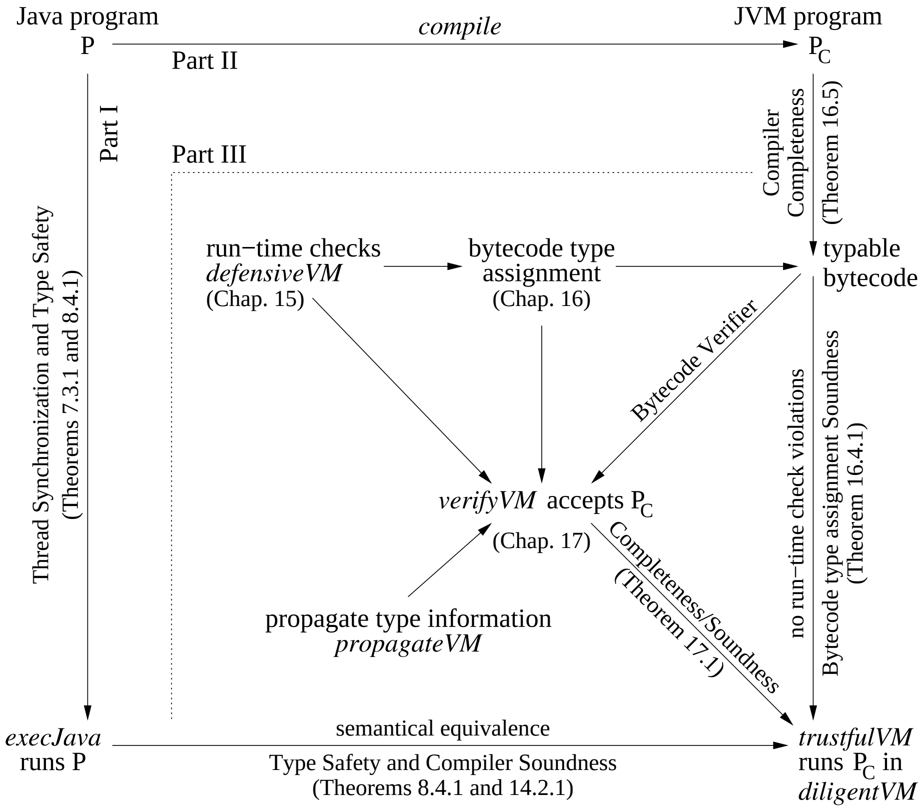


Fig. 3. Dependency Graph

Components for Language Layers. Since this language layering is common to all JVM components, we explain it first. 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. This decomposition can be made in such a way that in the resulting sequence of machines, 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. We illustrate this in Fig. 4.

A related structuring principle, which helped us to keep the size of the models small, consists in grouping similar instructions into one abstract instruction each, coming with appropriate parameters. These parameters become parameters of

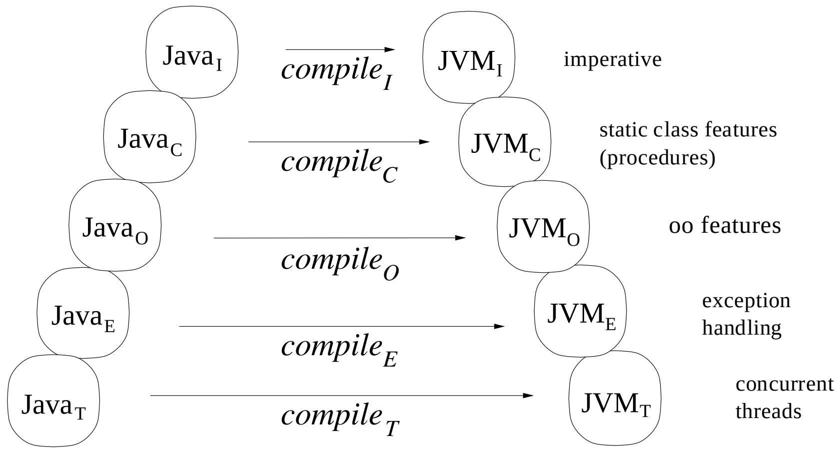


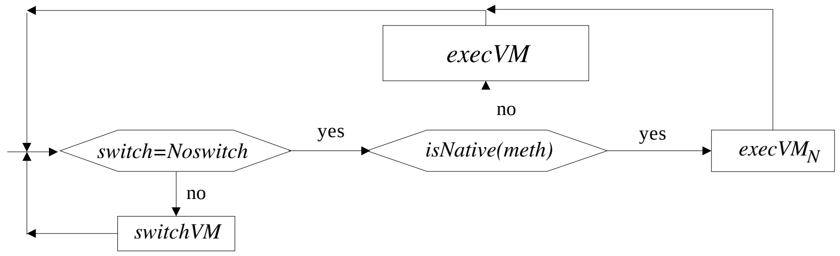
Fig. 4. Language oriented decomposition of Java/JVM

the corresponding ASM rules describing the semantical effect of those instructions. 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.

Execution Component. We now turn to explain the vertical components of the ASM model for the JVM. In one component we describe the *trustful execution* of bytecode that 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 $trustfulVM_T$, yields a succinct definition of the functionality of JVM execution in terms of language layered submachines *execVM* and *switchVM* (Fig. 5).

The language layered 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. This piecemeal description of single JVM instructions can be done similarly for the instructions provided in Java, yielding a succinct definition of the semantics of Java in terms of language layered submachines $Java_I$, $Java_C$, $Java_O$, $Java_E$, and $Java_T$. Exploiting the correspondence between these components for the Java/JVM machines yields a simple recursive definition of a compilation scheme for Java programs to JVM code, see Fig. 4, the detailed definition is in Part II of [34]. The conservativity of the component extensions allowed us to incrementally prove this compilation scheme to be correct, as is expressed by the following theorem.

Theorem 1 (Correctness of the compiler). *The ASMs for Java and the JVM, running through given Java code and its compilation to JVM code, pro-*



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

$switchVM_D$ extends $switchVM_E$ extends $switchVM_C$

Fig. 5. Decomposing trustfulVMs into execVMs and switchVMs

duce in corresponding method code segments the same values for (local, global, heap) variables and the same results of intermediate calculations, for the current method as well as for the method calls still to be completed.

The proof 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. Obviously, the statement of the theorem as phrased here is vague. In fact, it is part of the modeling and analysis work to provide a precise meaning of this intuitive statement, expressing that runs of the Java machine on a Java program and the corresponding runs of the JVM machine on the compiled program are equivalent. It took us 10 pages to make the underlying notion of corresponding runs and of their equivalence sufficiently precise to be able to carry out a proof for the correctness theorem, see Chapter 14 of [34]. The 83 case distinctions of that 24 pages long proof are not a bizarre effect of our modeling, but directly derive from—indeed are structured into—the situations which do occur during a Java computation for expression evaluation and statement execution, treated separately for each of the five language layers. This is a strength of the method that by localizing the proof obligations one has a key to modularize the overall proof: each new expression or statement feature will bring with it a clearly identifiable group of new cases to consider for definition (modeling) and proof (verification).

It was crucial for the compiler correctness proof to go through to take into account also some structural static constraints about Java runs, in particular conditions under which it can be proved that well-formed and well-typed Java programs are type safe, including the so called definite assignment rules for variables and the reachability analysis for statements. In fact we were led to correct some inconsistencies in those rules as defined in SUN's manuals (see below).

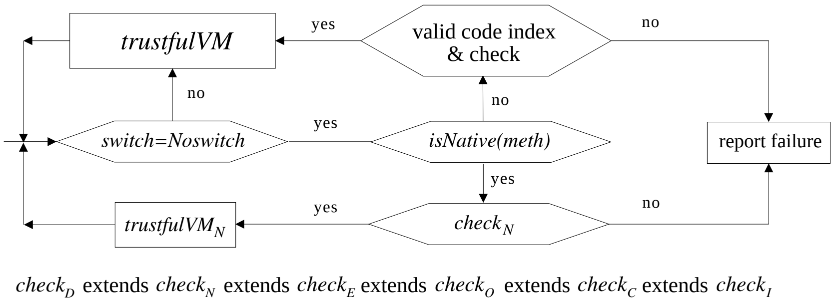


Fig. 6. Decomposing defensiveVMs into trustfulVMs and checks

Checking Component. The second group of language layered component machines we define are auxiliary machines whose parallel composition constitutes the *defensiveVM*. Their purpose is to define the verifier functionality in run-time terms of *trustfulVM* execution from a language layered component *check* . Since it is difficult to obtain a well motivated and clear definition of the bytecode verification functionality, we tried to accomplish also that task locally: guided by the language structure that allows to successively refine the checking conditions—from the imperative to the dynamic submachine—we took advantage from knowing for each type of instruction some run-time conditions which can guarantee its safe executability. To be more precise, as the architectural definition in Fig. 6 shows, the *defensiveVM* checks at run-time, before every execution step, the structural constraints which describe the verifier functionality (restrictions on run-time data: argument types, valid return addresses, resource bounds) guaranteeing safe execution. (Note that the static constraints on the well-formedness of the bytecode in Java class files are checked at link-time.) The detailed definition is given in Chapter 15 of [34]. For this new ASM *defensiveVM*, by its construction out of its component *trustfulVM*, one has the following theorem.

Theorem 2 (Correctness of defensive checking). *If the defensiveVM executes a program P successfully, then so does the trustfulVM, with the same semantical effect.*

Since we formulate the run-time *check*ing conditions referring to the types of values in registers and on the operand stack, instead of the values themselves, we can lift them to link-time checkable *bytecode type assignments*, i.e. assignments of certain type frames to code indices of method bodies. When lifting the run-time constraints, we make sure that if a given bytecode has a type assignment, then the code runs on the defensive VM without violating any of the run-time *check* conditions. For example, at run-time the values of the operands and the values stored in local variables belong to the assigned types; if there is a verify type assigned to a local variable, then at run-time the local variable contains a value which belongs to that verify type; if the type is a primitive type, then the value is of exactly that type; if the type is a reference type, then the value is a pointer to an object or array which is compatible with that type; the same is true for the verify types assigned to the operand stack, etc. The main difficulty is due to the subroutines, more precisely to the *Jsr(s)* and *Ret(x)* instructions which are used in the JVM to implement the *finally* block of Java *try* statements in the exception handling mechanism of Java. The problem is to correctly capture what is the type of return addresses from subroutines; as a matter of fact concerning this point we have identified in Chapter 16 of [34] a certain number of problems and inconsistencies in current implementations of the bytecode verifier. The outcome of this analysis is the following theorem, whose proof documents for all the cases that can occur for the single instructions in the given run why typable code can be safely executed.

Theorem 3 (Soundness of Bytecode Type Assignments). *Typable bytecode satisfies at run-time a set of invariants guaranteeing that when the code is run on the defensiveVM, it does not violate any of the dynamic constraints defined in the check component.*

The notion of bytecode type assignment also allows us to prove the completeness of the compilation scheme mentioned above. Completeness here means that bytecode which is compiled from a well-formed and well-typed Java program in a way that respects our compilation scheme, can be typed successfully, in the sense that it does have type assignments. More precisely we prove the general statement below, which implies the correctness of our Java-to-JVM compiler. We refine our compiler to a certifying code generator, which issues instructions together with the type information needed for the bytecode verification. Hence, the result of the extended compilation is not only a sequence of bytecode instructions but a sequence of triples $(instr, regT, opdT)$, where $(regT, opdT)$ is what we call a type frame for the instruction $instr$. We then prove that the so generated type frames satisfy the conditions for bytecode type assignments. This is yet another example of structuring definition and proof by conservative (purely incremental) extension.

When working on this proof, we detected a not so obvious inconsistency in the design of the Java programming language, namely an incompatibility of the reachability notions for the language and the JVM, related to the treatment of boolean expressions and the rules for the definite assignement of variables. The program in Fig. 7
shows that bytecode verification is not possible the way SUN's manuals suggest: although valid, the program is rejected by any bytecode verifier we have tried including JDK 1.2, JDK 1.3, Netscape 4.73-4.76, Microsoft VM for Java 5.0 and 5.5 and the Kimera Verifier (http://kimera.cs.washington.edu/). The problem is that in the eyes of the verifier the variable `i` is unusable at the end of the method at the `return i` instruction, whereas according to 16.2.14 in [21] the variable `i` is definitely assigned after the try statement. Our rules of definite assignment for the try statement are stronger and therefore the program is already rejected by our compiler. In [34] we exhibit another program that illustrates a similar

```
class Test {
  static int m(boolean b) {
    int i;
    try {
      if (b) return 1;
      i = 2;
    } finally { if (b) i = 3; }
    return i;
  }
}
```

Fig. 7. A valid Java program rejected by all known verifiers

problem for labeled statements. In conclusion, one can avoid this inconsistency by slightly restricting the class of valid programs by sharpening the rules for definite assignment for *finally* and for labeled statements. As a result we could establish the following desirable property for the class of certifying compilers.

Theorem 4 (Compiler Completeness Theorem). *The family of type frames generated by the certifying compiler for the body of a method μ is a bytecode type assignment for μ.*

As a corollary, the Java-to-JVM compiler we define is correct since it is extended conservatively by a certifying compiler.

Bytecode Verifier Component. Having distilled the bytecode verifier functionality in the notion of bytecode type assignment, we are ready to extend the *trustfulVM* by a new component, a link-time bytecode verifier. Before *trustfulVM* can run a method in a class that has been loaded, for each method in that class the verifier attempts to compute a—in fact a most specific—bytecode type assignment for the method. The (architecture of the) resulting machine *diligentVM* is defined in Fig. 8.

One has to show that the *verifyVM* component is sound and complete, which is expressed by the following two theorems that we can prove for our novel (subroutine call stack free) bytecode verifier.

Theorem 5 (Bytecode Verifier Soundness). *During the computation of the verifier for any given method body, the bytecode type frames computed so far satisfy the conditions for bytecode type assignments. verifyVM terminates, either rejecting the code with a type failure detection (in case the method body is not typable) or accepting it and issuing a bytecode type assignment for it.*

Theorem 6 (Bytecode Verifier Completeness). *If a method body has a bytecode type assignment, then verifyVM accepts the code and during the verification process the type frames computed so far by verifyVM are more specific than that bytecode type assignment.*

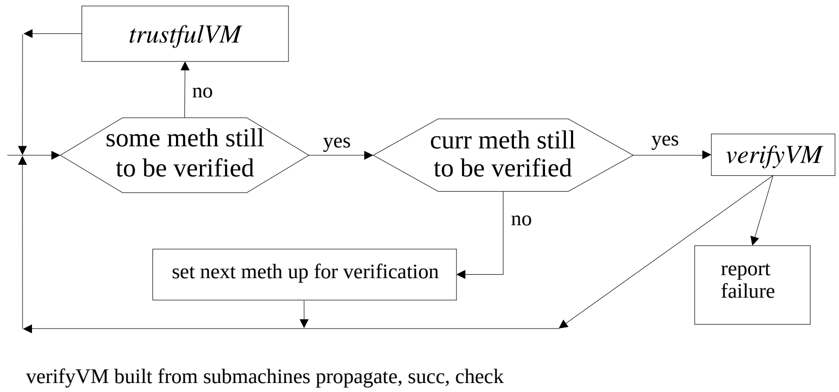


Fig. 8. Decomposing diligent JVMs into trustfulVMs and verifyVMs

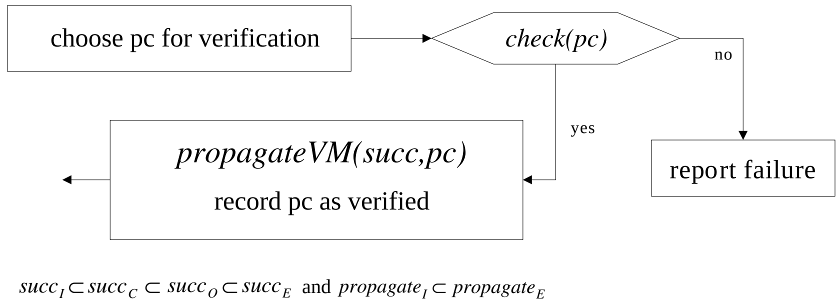


Fig. 9. Decomposing verifyVMs into propagateVMs, checks, succs

Components of the Bytecode Verifier. To compute a bytecode type assignment for a given method, *verifyVM* at each step chooses a still to be verified code index *pc*, starting at code index 0, to *check* the type conditions there. Upon successful *check*, as defined for the *defensiveVM*, the verifier marks for further verification steps the indices of all *successors* of *pc* that can be reached by the computation, trying to *propagate* the type frame computed at *pc* to each possible immediate *successor* of *pc*. This provides the architecture of the machine *verifyVM*, built out of three components *check, propagate, succ* as defined in Fig. 9.

At this point it should not any more come as a surprise to the reader that the two new components of *verifyVM*, namely the ASM *propagateVM* and the function *succ*, are language layered similarly to the predicate *check* defined already above as part of *defensiveVM*. A further reuse of previously defined machines stems from the fact that the submachine *propagateVM*, together with the

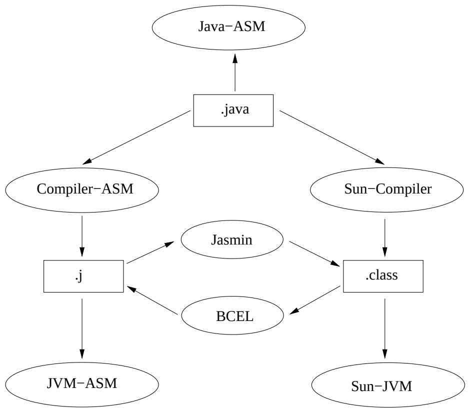


Fig. 10. Relationship between different machines

function *succ*, defines a link-time simulation (type version) of the *trustfulVM* illustrated above.

In a similar way the loading mechanism can be introduced by refining the components *execVM* and *switchVM*, see Chapter 18 in [34].

The modular component-based structure of both definitions and proofs explained above for Java and the JVM is reassumed in Fig. 3, showing how the components and the proofs of their basic properties fit together to establish the desired property for the compilation and safe execution of arbitrary Java programs on the *dynamicVM*, as expressed above in the Main Theorem.

AsmGofer executable refinements. The experimentation with the AsmGofer executable refinements of the models outlined above was crucial to get the models and the proofs of our theorems right. AsmGofer is an ASM programming system developed by Joachim Schmid and available at www.tydo.de/AsmGofer. It extends TkGofer to execute ASMs which come with Haskell definable external functions. It provides step-by-step execution and comes with GUIs to support debugging of Java/JVM programs. First of all it allows to execute the Java source code in our Java ASM and to observe that execution—there is no counterpart for this in SUN's development environment, but similar work has been done independently, using the Centaur system, by Marjorie Russo in her recent PhD thesis [32]. Furthermore one can compile Java programs to bytecode which can be executed either on our ASM for JVM or (using *Jasmin* for the conversion to binary class format) on SUN's implementation. More generally, 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, as illustrated in Fig. 10.

References

1. J. R. Abrial. *The B-Book. Assigning Programs to Meanings.* Cambridge University Press, 1996.
2. E. Börger. A logical operational semantics for full Prolog. Part I: Selection core and control. In E. Börger, H. Kleine-Büning, and M. Richter, editors, *CSL 89*, number 440 in Lecture Notes in Computer Science, pages 36–64. Springer-Verlag, 1989.
3. E. Börger. A logical operational semantics for full Prolog. Part II: Built-in predicates for database manipulations. In B. Rovan, editor, *MFCS'90. Mathematical Foundations of Computer Science*, number 452 in Lecture Notes in Computer Science, pages 1–14. Springer-Verlag, 1990.
4. E. Börger. Logic programming: The evolving algebra approach. In B. Pehrson and I. Simon, editors, *IFIP 13th World Computer Congress 1994. Volume I: Technology and Foundations*, pages 391–395. Elsevier, Amsterdam, 1994.
5. 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.
6. E. Börger. Abstract State Machines at the cusp of the millenium. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines. Theory and Applications*, number 1912 in Lecture Notes in Computer Science, pages 1–8. Springer-Verlag, 2000.
7. E. Börger, A. Cavarra, and E. Riccobene. An ASM semantics for UML Activity Diagrams. In T. Rust, editor, *Proc. AMAST 2000*, number 1912 in Lecture Notes in Computer Science, pages 361–366. Springer-Verlag, 2000.
8. E. Börger, A. Cavarra, and E. Riccobene. Modeling the Dynamics of UML State Machines. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines. Theory and Applications*, number 1912 in Lecture Notes in Computer Science, pages 223–241. Springer-Verlag, 2000.
9. E. Börger and K. Dässler. Prolog: DIN papers for discussion. In *ISO/IEC JTCI SC22 WG17 Prolog standardization document*, number 58 in ISO/IEC Documents, pages 92–114. NPL Middlesex, 1990.
10. E. Börger and I. Durdanovic. Correctness of Compiling Occam to Transputer Code. *Computer Journal*, 39(1):52–92, 1996.
11. E. Börger, E. Riccobene, and J. Schmid. Capturing Requirements by Abstract State Machines: The Light Control Case Study. *J. Universal Computer Science*, 6.7:597–620, 2000. Special Requirement Engineering Issue.
12. E. Börger and D. Rosenzweig. The WAM–Definition and Compiler Correctness. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, pages 20–90. Elsevier Science B.V./North–Holland, 1995.
13. 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.
14. E. Börger, J. Schmid, and P. Päppinghaus. Report on a Practical Application of ASMs in Software Design. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines. Theory and Applications*, number 1912 in Lecture Notes in Computer Science, pages 361–366. Springer-Verlag, 2000.
15. U. Glässer, Y. Gurevich, and M. Veanes. Universal Plug and Play Machine Models. Technical Report MSR-TR-2001-59, Microsoft Research, 2001.

16. W. Goerigk and H. Langmaack. Compiler Implementation Verification and Trojan Horses. In D. Bainov, editor, *Proc. 9th International Colloquium on Numerical Analysis and Computer Science with Applications*, Plovdiv, Bulgaria, 2001. Extended Draft Version available at `http://www.informatik.uni-kiel.de/~wg/-Berich te/Plovdiv.ps.gz`.
17. W. Goerigk and H. Langmaack. Will Informatics be able to Justify the Construction of Large Computer Based Systems? Technical Report 2015, CS Department University of Kiel, 2001.
18. G. Goos and W. Zimmermann. Verification of compilers. In *Correct System Design*, number 1710 in Lecture Notes in Computer Science, pages 201–230. Springer-Verlag, 1999.
19. G. Goos and W. Zimmermann. Verifying compilers and ASMs. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Correct System Design*, number 1912 in Lecture Notes in Computer Science, pages 177–202. Springer-Verlag, 2000.
20. J. Gosling, B. Joy, and G. Steele. *The Java(tm) Language Specification*. Addison Wesley, 1996.
21. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java(tm) Language Specification*. Addison Wesley, second edition, 2000.
22. Y. Gurevich. Reconsidering Turing's Thesis: Toward More Realistic Semantics of Programs. Technical Report CRL-TR-36-84, University of Michigan, Computing Research Lab, 1984.
23. Y. Gurevich. A New Thesis. *Notices of the American Mathematical Society*, page 317, 1985. abstract 85T-68-203, received May 13.
24. Y. Gurevich. Logic and the Challenge of Computer Science. In E. Börger, editor, *Current Trends in Theoretical Computer Science*, pages 1–57. Computer Science Press, 1988.
25. Y. Gurevich. Evolving algebras: An attempt to discover semantics. *Bulletin of the EATCS*, 43:264–284, 1991.
26. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
27. J. Huggins. Abstract State Machines. ASM Web pages, maintained at `http://www.eecs.umich.edu/gasm/`.
28. ISO. *ISO/IEC 13211-1 Information Technology-Programming Languages-Prolog-Part 1: General Core*. ISO, 1995.
29. ITU. ITU-T Recommendation Z-100: Languages for Telecommunications Applications – Specification and Description Language SDL. Annex F: SDL formal semantics definition, 2000.
30. A. Kappel. Implementation of Dynamic Algebras with an Application to Prolog. Master's thesis, CS Dept., University of Dortmund, Germany, November 1990. An extended abstract "Executable Specifications based on Dynamic Algebras" appeared in A. Voronkov (ed.): Logic Programming and Automated Reasoning, volume 698 of LNAI, Springer, 1993, pages 229-240.
31. T. Lindholm and F. Yellin. *The Java(tm) Virtual Machine Specification*. Addison Wesley, second edition, 1999.
32. M. Russo. Java et ses aspect concurrents: semantiques formelle, visualisation et proprietes, 2001. PhD thesis University of Nice-Sophia-Antipolis.
33. J. Schmid. Executing ASM specifications with AsmGofer, 1999. Web pages at http://www.tydo.de/AsmGofer.
34. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine. Definition, Verification, Validation*. Springer-Verlag, 2001.