A Structured and High-Level Definition of Java and of its Provably Correct and Secure Implementation on the Java Virtual Machine

(The ASM Java/JVM Project)

Egon Börger

Dipartimento di Informatica, Universita di Pisa
http://www.di.unipi.it/~boerger
Reference:

Java and the Java Virtual Machine -
Definition, Verification, and Validation

R. Stärk, J. Schmid, E. Börger

Springer-Verlag, 2001

see http://www.inf.ethz.ch/~jbook/
Goal: Real-life Industrial Case Study Book

Illustrate through a relevant & complex example how to enhance practical syst design & analysis using ASMs for rigorous high-level modeling linked seamlessly to executable code in a verifiable and validatable way

- developing succinct ground models with precise, unambiguous, yet understandable meaning to provide the possibility for implementation independent system analysis and validation

- refining & structuring models into a system (hierarchy) of (sub)models, modularizing orthogonal design decisions (“for change”), justifying them as correct
  • linking the ground model to the implementation
  • documenting the entire design for reuse and maintenance
Method: Separate & Combine Different Concerns using ASMs

• Separating orthogonal design decisions
  – to keep design space open (specify for change, avoiding premature design decisions)
  – to structure design space (rigorous interfaces for system (de)composition)

• Separating design from analysis
  – separating validation (by simulation) from verification (by proofs)
  – separating verification levels (degrees of proof detail)
    • reasoning for human inspection (design justification)
    • rule based reasoning systems
      – interactive systems
      – automatic tools: model checkers, automatic theorem provers

• Crossing system levels by most general abstraction and refinement notions offered by ASMs, tunable to the given problem
The Problem

Java/JVM claimed by SUN to be a safe and secure, platform independent programming env for Internet: correctness problem for compiler, loader (name space support), verifier, access right checker (security manager), interpreter.

Usr. Usr.class Internet

Java Compiler Usr.class

JVM

Verifier Loader

Preparator Interpreter

Run Time Machine

insecure

Input Output

Sys.class

© Egon Boerger: The ASM Java/JVM Project
Specific Goal of the ASM Java/JVM Project

Abstract (platform independent), rigorous but transparent, modular definition providing basis for mathematical and experimental analysis

– Reflecting SUN’s design decisions (faithful ground model)
– Offering correct high-level understanding (to be practically useful for programmers)
– Providing rigorous, implementation independent basis for
  • Analysis and Documentation (for designers) through
    – Mathematical verification
    – Experimental validation
    – Comparison of different implementations
  • Implementation (compiln, loading, bytecode verification, security schemes)
Main Result

A Structured and High-Level Definition of Java
and of its Provably Correct and Secure Implementation
on the Java Virtual Machine

Theorem. Under explicitly stated conditions, any
well-formed and well-typed Java program:

- upon correct compilation
- passes the verifier
- is executed on the JVM
- executes
  - without violating any run-time checks
  - correctly wrt Java source pgm semantics
Language driven decomposition of Java, JVM, compilation

Split into horizontal language components (conservative extensions)
The language driven decomposition of execJava and its submachines

execJava =

\( \text{execJava}_I \) imperative control constructs

\( \text{execJava}_C \) static class features (modules)

\( \text{execJava}_O \) oo features

\( \text{execJava}_E \) exception handling

\( \text{execJava}_T \) concurrent threads

execJava\(_I\) =

\( \text{execJavaExp}_I \) expression evaluation

\( \text{execJavaStmt}_I \) statement execution

NB. Grouping similar instructions into one parameterized abstract instr
Pgm exec as walk thru annotated abstract syntax tree

STATE defined by pos : Pos restbody: Pos → Phrase ∪ Val ∪ Abr

MACROS: context (pos) = if restbody/pos ∈ Exp ∪ Bstm or pos = first then restbody/pos else restbody/up(pos)

Replacing a phrase (in the current pos) by its result:
yield (result) = restbody:= restbody[result/pos]

Passing the result of a phrase (in the current pos) to its parent phrase:
yieldUp (result) = restbody:= restbody[result/up(pos)]
pos := up(pos)

Being positioned on a direct subphrase of a structure f (...t...):
s=f (... t ...) stands for s = f (...t...) & pos = & restbody(pos)=t

Phrase: exps & block stms Val: values Abr: reasons for abruption
execJavaExp \_ = case context(pos) of
  lit \rightarrow yield(JLS(lit))
  loc \rightarrow yield(locals(loc))
  uop ^\alpha exp \rightarrow pos := \alpha
  uop \downarrow val \rightarrow yieldUp(JLS(uop, val))
  ^\alpha exp_1 bop ^\beta exp_2 \rightarrow pos := \alpha
  \downarrow val bop ^\beta exp \rightarrow pos := \beta
  ^\alpha val_1 bop \downarrow val_2 \rightarrow if \neg(bop \in \text{divMod} \land isZero(val_2)) then
    yieldUp(JLS(bop, val_1, val_2))
loc = ^\alpha exp \rightarrow pos := \alpha
loc = \downarrow val \rightarrow locals := locals \oplus \{(loc, val)\}
    yieldUp(val)
  ^\alpha exp_0 ? ^\beta exp_1 : ^\gamma exp_2 \rightarrow pos := \alpha
  \downarrow val ? ^\beta exp_1 : ^\gamma exp_2 \rightarrow if val then pos := \beta else pos := \gamma
  ^\alpha True ? \downarrow val : ^\gamma exp \rightarrow yieldUp(val)
  ^\alpha False ? ^\beta exp : \downarrow val \rightarrow yieldUp(val)
execJavaStmI = case context(pos) of
  ; → yield(Norm)
  α exp; → pos := α
  ▶ val; → yieldUp(Norm)
  break lab; → yield(Break(lab))
  continue lab; → yield(Continue(lab))
  lab : α stm → pos := α
  lab : ▶ Norm → yieldUp(Norm)
  lab : ▶ Break(lab_b) → if lab = lab_b then yieldUp(Norm)
                   else yieldUp(Break(lab_b))
  lab : ▶ Continue(lab_c) → if lab = lab_c then yield(body/up(pos))
                             else yieldUp(Continue(lab_c))
  phrase(▶ abr) → if pos \neq firstPos \& propagatesAbr(restbody/up(pos)) then
                     yieldUp(abr)

{} → yield(Norm)
{α1 stm1 \ldots αn stmn} → pos := α1
{α1 Norm \ldots ▶ Norm} → yieldUp(Norm)
{α1 Norm \ldots ▶ Norm αi+1 stmi+1 \ldots αn stmn} → pos := αi+1

if (α exp) β stm1 else γ stm2 → pos := α
if (▶ val) β stm1 else γ stm2 → if val then pos := β else pos := γ
if (α True) ▶ Norm else γ stm → yieldUp(Norm)
if (α False) β stm else ▶ Norm → yieldUp(Norm)

while (α exp) β stm → pos := α
while (▶ val) β stm → if val then pos := β else yieldUp(Norm)
while (α True) ▶ Norm → yieldUp(body/up(pos))

Type x; → yield(Norm)
The execJava\textsubscript{C/O} extensions

\begin{align*}
\text{execJava}_C &= \\
\text{execJavaExp}_C &\quad \text{extending expression evaluation} \\
\text{execJavaStmt}_C &\quad \text{extending statement execution}
\end{align*}

Adding

\begin{itemize}
  \item class fields (global variables)
  \item class method invocation/return (procedures)
  \item class initializers (module initializers)
\end{itemize}

\begin{align*}
\text{execJava}_O &= \\
\text{execJavaExp}_O &\quad \text{adding instance fields/methods}
\end{align*}
Fields treated similarly to local vars (with local replaced by global), but: one has to initialize each class at its first active use, i.e. when for the first time accessing (or assigning to) some of its fields or calling some of its methods (after left-to-right arg evaluation) (or upon creation in Java

\[
\text{execJavaExp}_C = \text{case context}(\text{pos}) \text{ of}
\]
\[
c.f \rightarrow \text{if } \text{initialized}(c) \text{ then yield(globals}(c/f)) \text{ else initialize}(c)
\]
\[
c.f = \alpha \text{ exp} \rightarrow \text{pos} := \alpha
\]
\[
c.f = \triangleright \text{ val} \rightarrow \text{if } \text{initialized}(c) \text{ then}
\]
\[
\text{globals}(c/f) := \text{val}
\]
\[
\text{yieldUp}(\text{val})
\]
\[
\text{else initialize}(c)
\]
\[
c.m^\alpha(\text{exp}s) \rightarrow \text{pos} := \alpha
\]
\[
c.m^{\triangleright}(\text{vals}) \rightarrow \text{if } \text{initialized}(c) \text{ then invoke}(\text{up}(\text{pos}), c/m, \text{vals})
\]
\[
\text{else initialize}(c)
\]
\[
() \rightarrow \text{yield}([])
\]
\[
(\alpha_1 \text{ exp}_1, \ldots, \alpha_n \text{ exp}_n) \rightarrow \text{pos} := \alpha_1
\]
\[
(\alpha_1 \text{ val}_1, \ldots, \triangleright \text{ val}_n) \rightarrow \text{yieldUp}([\text{val}_1, \ldots, \text{val}_n])
\]
\[
(\alpha_1 \text{ val}_1, \ldots, \triangleright \text{ val}_i, \alpha_{i+1} \text{ exp}_{i+1} \ldots \alpha_n \text{ exp}_n) \rightarrow \text{pos} := \alpha_{i+1}
\]

© Egon Boerger: The ASM Java/JVM Project 14
Execution of initialization code for a class is started only when the superclass is already initialized, and also at the top of the class hierarchy. A class becomes initialized upon exiting from its initialization method.

\[
\text{execJavaStm}_C = \text{case } \text{context}(\text{pos}) \text{ of }
\]
\[\text{static } \alpha \text{stm} & \rightarrow \text{let } c = \text{classNm}(\text{meth}) \]
\[\text{if } c = \text{Object} \lor \text{initialized}(\text{super}(c)) \text{ then } \text{pos} := \alpha \]
\[\text{else } \text{initialize}(\text{super}(c)) \]
\[\text{static } \uparrow \text{Return} \rightarrow \text{exitMethod}(\text{Norm}) \]
\[\text{classState}(\text{classNm}(\text{meth})) := \text{Initialized} \]
\[\text{return } \alpha \text{exp}; \rightarrow \text{pos} := \alpha \]
\[\text{return } \uparrow \text{val}; \rightarrow \text{yieldUp}(\text{Return}(\text{val})) \]
\[\text{return}; \rightarrow \text{yield}(\text{Return}) \]
\[\text{lab} : \uparrow \text{Return} \rightarrow \text{yieldUp}(\text{Return}) \]
\[\text{lab} : \uparrow \text{Return}(\text{val}) \rightarrow \text{yieldUp}(\text{Return}(\text{val})) \]
\[\text{Return} \rightarrow \text{if } \text{pos} = \text{firstPos} \land \neg \text{null}(\text{frames}) \text{ then } \]
\[\text{exitMethod}(\text{Norm}) \]
\[\text{Return}(\text{val}) \rightarrow \text{if } \text{pos} = \text{firstPos} \land \neg \text{null}(\text{frames}) \text{ then } \]
\[\text{exitMethod}(\text{val}) \]
\[\uparrow \text{Norm}; \rightarrow \text{yieldUp}(\text{Norm}) \]
instance field values of objects stored (using setField) in & retrieved (using getField) from Heap, under the ref of the object; default values assigned upon creation. The class of new parametrized class instances is initialized before parameter evaln. this stored as local var; bound by inst meth call & by return from a constructor (to the newly created object, see the extension of exitMethod)
The execJava\textsubscript{E/T} extensions

\begin{align*}
\text{execJava}_{E} &= \\
\text{execJavaExp}_{E} &\quad \text{for evaluation of run-time exceptions} \\
\text{execJavaStm}_{E} &\quad \text{for execution of exception statements} \\
\text{execJava}_{T} &= \\
\text{execJavaStm}_{T} &\quad \text{for synchronization statements} \\
\end{align*}

(as part of execJavaThread)
Ahrs in try stms: caught excs lead to catch code exec, othr ahrs propagate

catch code yields up Norm or an abr

For finally stms: ahrs suspend upon entering finally stm

exiting propagates up the suspended abr (resumed) or a new abr

Uncaught excs propagate up the method call stack; in static class initializers they make the class unusable
Examples of run-time exceptions

\[
\text{execJavaExp}_E = \text{case \hspace{1em} context(pos) of} \\
\quad \alpha \text{val}_1 \hspace{1em} bop \hspace{1em} \triangleright \hspace{1em} \text{val}_2 \\
\hspace{1em} \rightarrow \text{if} \hspace{1em} bop \in \text{divMod} \land \text{isZero(val}_2\hspace{1em}) \hspace{1em} \text{then} \\
\hspace{4em} \text{fail(ArithmeticException)} \\
\quad \triangleright \hspace{1em} \text{ref} . \hspace{1em} c / f \\
\hspace{1em} \rightarrow \text{if} \hspace{1em} \text{ref} = \text{null} \hspace{1em} \text{then} \hspace{1em} \text{fail(NullPointerException)} \\
\quad \alpha \text{ref} . \hspace{1em} c / f = \hspace{1em} \triangleright \hspace{1em} \text{val} \\
\hspace{1em} \rightarrow \text{if} \hspace{1em} \text{ref} = \text{null} \hspace{1em} \text{then} \hspace{1em} \text{fail(NullPointerException)} \\
\quad \alpha \text{ref} . \hspace{1em} c / m \hspace{1em} \triangleright \hspace{1em} \text{(vals)} \\
\hspace{1em} \rightarrow \text{if} \hspace{1em} \text{ref} = \text{null} \hspace{1em} \text{then} \hspace{1em} \text{fail(NullPointerException)} \\
\quad (c) \hspace{1em} \triangleright \hspace{1em} \text{ref} \\
\hspace{1em} \rightarrow \text{if} \hspace{1em} \text{ref} \neq \text{null} \land \text{classOf(ref)} \not\subset_h c \hspace{1em} \text{then} \\
\hspace{4em} \text{fail(ClassCastException)} \\
\text{where \hspace{1em} fail(exc) = yield (throw new exc();)}
\]

When classes become unusable, their initialization is impossible, so that \text{initialize(c)} is extended by the following:

\[
\text{if classState(c) = Unusable then fail (NoClassDefFoundError)}
\]
Theorem: Java is type safe

i.e. when a legal well-typed Java pgm is executed:
- run-time vals of static/instance fields/array elems are compatible with their declared types
- references to objects are in the heap (no dangling pointers)
- run-time positions satisfy compile-time constraints (reachable, definitely assigned vars are well-defined local vars with vals of compile-time type,…)
- positions of normally completing stms are compile-time normal
- evaluated exprs/returned vals have compile-time compatible type
- abruptions (jump,return,exc) have compile-time compatible type
- stacks do not overflow nor underflow, …

Proof: induction on Java ASM runs, based upon a rigorous definition of the rules for definite assignment
Extending execJava, to become component of ExecJavaThread

\(\text{execJavaStm}_T = \text{case context}(\text{pos}) \text{ of}
\)

synchronized \((^\alpha \text{exp})^\beta \text{stm} \rightarrow \text{pos} := \alpha\)

synchronized \((\uparrow \text{ref})^\beta \text{stm} \rightarrow
\)

\text{if} \ \text{ref} = \text{null} \ \text{then} \ \text{fail}(\text{NullPointerException})
\text{else}
\text{if} \ \text{ref} \in \text{sync}(\text{thread}) \ \text{then}
\text{sync}(\text{thread}) := [\text{ref}] \cdot \text{sync}(\text{thread})
\text{locks}(\text{ref}) := \text{locks}(\text{ref}) + 1
\text{pos} := \beta
\text{else}
\text{exec}(\text{thread}) := \text{Synchronizing}
\text{syncObj}(\text{thread}) := \text{ref}
\text{cont}(\text{thread}) := (\text{frames}, (\text{meth}, \text{restbody}, \beta, \text{locals}))
\text{synchronized} \((^\alpha \text{ref}) \uparrow \text{Norm} \rightarrow \text{releaseLock}(\text{Norm})\)
\text{synchronized} \((^\alpha \text{ref}) \uparrow \text{abr} \rightarrow \text{releaseLock}(\text{abr})\)

\text{static} \uparrow \text{abr} \rightarrow \text{notifyThreadsWaitingForInitialization}
\text{abr} \rightarrow \text{if} \ \text{pos} = \text{firstPos} \land \text{null}(\text{frames}) \ \text{then} \ \text{killThread}
Abstract scheduling of Multiple Threads:
inserting execJava into ExecJavaThread

Thread scheduling separated from thread execution

ExecJavaThread ≡

\[
\text{choose } q \text{ in } \text{dom}(\text{exec}), \text{runnable}(q)
\]

\[
\begin{align*}
\text{if } q &= \text{thread and exec}(q) = \text{Active} \\
\text{then } \text{execJava} \\
\text{else} \\
\text{if } \text{exec}(q) = \text{Active} \text{ then} \\
& \quad \text{cont(thread)} := (\text{frames}, (\text{methd}, \text{restbody}, \text{pos}, \text{locals})) \\
& \quad \text{thread} := q \\
& \quad \text{run}(q)
\end{align*}
\]
Diagram notation for Control State ASMs

- **Diagram Description:**
  - If $ctl = i$ then
    - If $cond_1$ then $rule_1$
    - $ctl := j_1$
    - ... if $cond_n$ then $rule_n$
    - $ctl := j_n$

- **Labeling:**
  - Labeling of the arrows by "control" states
  - Often suppressed

- **UML:**
  - Combined branching/action nodes
**Defining execJavaThread as control state ASM**

Choose \( t \) in \( \text{ExecRunnableThread} \)

- **suspend thread**
- **resume \( t \)**

no \( t \) is curr Active thread

yes \( \text{execJava} \)

---

**Thread scheduling separated from thread execution**

\[
\begin{align*}
\text{t in } \text{ExecRunnableThread} &= ( t \in \text{dom(exec)} \&\& \text{runnable}(t) ) \\
\text{t is curr Active thread} &= ( t = \text{thread} \&\& \text{exec}(t) = \text{Active}) \\
\text{suspend thread} &= \text{if } \text{exec(thread)} = \text{Active} \\
&\quad \text{then } \text{cont(thread)} := (\text{frames, currframe}) \\
\text{resume}(t) &= \text{thread} := t \\
&\quad \text{run}(t)
\end{align*}
\]
Theorem: Correctness of Thread Synchronization in Java

- Runtime threads are valid threads (of type THREAD).
- If the execution state of a thread is Not Started, then the thread is not synchronized on any object and is not in the wait set of any object.
- If the state of a thread is synchronizing, then the thread is not already synchronized on the object it is competing for.
- If a thread is synchronized on an object, then the object is a valid reference in the heap.
- If a thread is waiting for an object, then it is synchronized on and is in the wait set of the object (without holding the lock of the object).
- If a thread has been notified on an object, then it is no longer in the wait set of the object. It is still synchronized on the object, but it does not hold the lock of the object.
Theorem: Correctness of Thread Synchronization in Java (Cont’d)

• A thread cannot be in the wait set of two different objects.
• If a thread has terminated normally or abruptly, then it does not hold the lock of any object.
• If a thread holds the lock of an object, then the lock counter of the object is exactly the number of occurrences of the object in the list of synchronized objects of the thread.
• It is not possible that at the same time, two different threads hold the lock of the same object.
• If the lock counter of an object is greater than zero, then there exists a thread which holds the lock of the object.
• ...

PROOF. Induction on Java ASM runs.
Security Driven JVM Decomposition

- **trustfulVM**: defines the execution functionality incrementally from language layered submachines `execVM`, `switchVM`
- **defensiveVM**: defines the constraints to be checked, in terms of `trustfulVM` execution, from the language layered submachine `check`; calls `trustfulVM` for execution
- **diligentVM**: checks the constraints at link-time, using a language layered submachine `verifyVM`; calls `trustfulVM` for execution
- **verifyVM** built up from language layered submachines `check`, `propagateVM`, `succ`
- **dynamicVM**: dynamic loading and linking of classes
Stepwise refinement of trustfulVM

execVM and switchVM incrementally extended (language driven)

\[
\text{trustfulVM}_I = \text{execVM}_I \subseteq \text{execVM}_C \subseteq \text{execVM}_O \subseteq \text{execVM}_E \\
\text{execVM}_N \subseteq \text{execVM}_D
\]
defining instructionwise changes of current frame

\[
\text{switchVM}_C \subseteq \text{switchVM}_E \subseteq \text{switchVM}_D
\]
defining changes of frame stack
reflecting meth call/return, class initialization, capturing exceptions, class load/linking
Stating rigorously and proving the Correctness of compiling from Java to JVM

• With respect to the ASM models for Java and JVM, and wrt the definition of compile from Java to JVM code, including the exception table, the execution of P in Java and the execution of compile(P) in Trustful VM are equivalent (in a sense made precise), for arbitrary pgms P.

• PROOF. By induction on the runs of the Java/JVM ASMs, using the type safety theorem.

• NB. This includes the correctness of exception handling

Deriving the Bytecode Verifier Conditions from Type Checking Runtime Constraints

- **Defensive VM**: Checks at run-time, before every execution step, the “structural constraints” which describe the verifier functionality (restrictions on run-time data: argument types, valid Ret addresses, resource bounds,… ) guaranteeing “safe” execution.
- **Static constraints** (well-formedness) checked at link-time.
- **Theorem**: If Defensive VM executes P successfully, then so does Trustful VM, with the same semantical effect.
Stepwise refinement of defensiveVM

check incrementally extended, language driven as for trustfulVM

i.e. check₁ extended by check₂
    extended by check₃
    extended by check₄
    extended by check₅
    extended by check₆
    extended by check₇

report failure
Bytecode Type Assignments

- Link-time verifiable type assignments (conditions) extracted from checking function of the Defensive VM
  
  Main problem: return addresses of Jsr(s), reached using Ret(x)

- **Soundness Theorem**: If P satisfies the type assignment conditions, then Defensive VM executes P without violating any run-time check.
  
  Proof by induction on runs of the Defensive VM

- **Completeness Theorem**: Bytecode generated by compile from a legal Java program does have type assignments.
  
  Inductive proof introduces certifying compiler assigning to each byte code instr also a type frame, which then can be shown to constitute a type assignment for the compiled code
Stepwise refinement of diligentVM\textsubscript{I,C,O,E}

switchVM\textsubscript{C} in trustfulVM is refined to also link classes before their initialization, where the linking submachine triggers verifyVM.

verifyVM decomposed into lang layered check, succ, propagate.
Stepwise refinement of verifyVM

choose pc for verification

check(pc)

no

yes

report failure

propagateVM(succ, pc)

record pc as verified

propagateVM the checked type frame from pc to all possible successor frames, simulating execVM on types frames

propagateVM and succ incrementally extended

succ_I ⊆ succ_C ⊆ succ_O ⊆ succ_E

propagate_I ⊆ propagate_E
Proving Bytecode Verifier Complete and Correct

- **Bytecode Verifier Soundness Theorem**: For any program $P$, the bytecode verifier either rejects $P$ or during the verification satisfies the type assignment conditions for $P$.

- **Bytecode Verifier Completeness Theorem**: If $P$ has a type assignment, then the Bytecode Verifier does not reject $P$ and computes a most specific type assignment.
Validating Java, JVM, compile

- **AsmGofer**: ASM programming system, extending TkGofer to execute ASMs (with Haskell definable external fcts)
- Provides **step-by-step execution**, with GUIs to support **debugging** of Java/JVM programs.
- Allows for the executable ASM models of Java/JVM:
  - to execute the Java source code $P$ (**no counterpart in SUN env**)
  - to compile Java pgms $P$ to bytecode $\text{compile}(P)$ (in textual representation, using JASMIN to convert to binary class format)
  - to execute the bytecode programs $\text{compile}(P)$
  E.g. our Bytecode Verifier rejects Saraswat’s program
- **Developed by Joachim Schmid**, available at [www.tydo.de/AsmGofer](http://www.tydo.de/AsmGofer)
Java and the Java Virtual Machine. Definition, Verification, and Validation

R. Stärk, J. Schmid, E. Börger

see http://www.inf.ethz.ch/~jbook/

For ASMGofer see www.tydo.de/AsmGofer/

My home page http://www.di.unipi.it/~boerger