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 succint ground models with precise, unambiguous, yet understandable meaning

to provide the possibility for implementation independent system analysis and validation

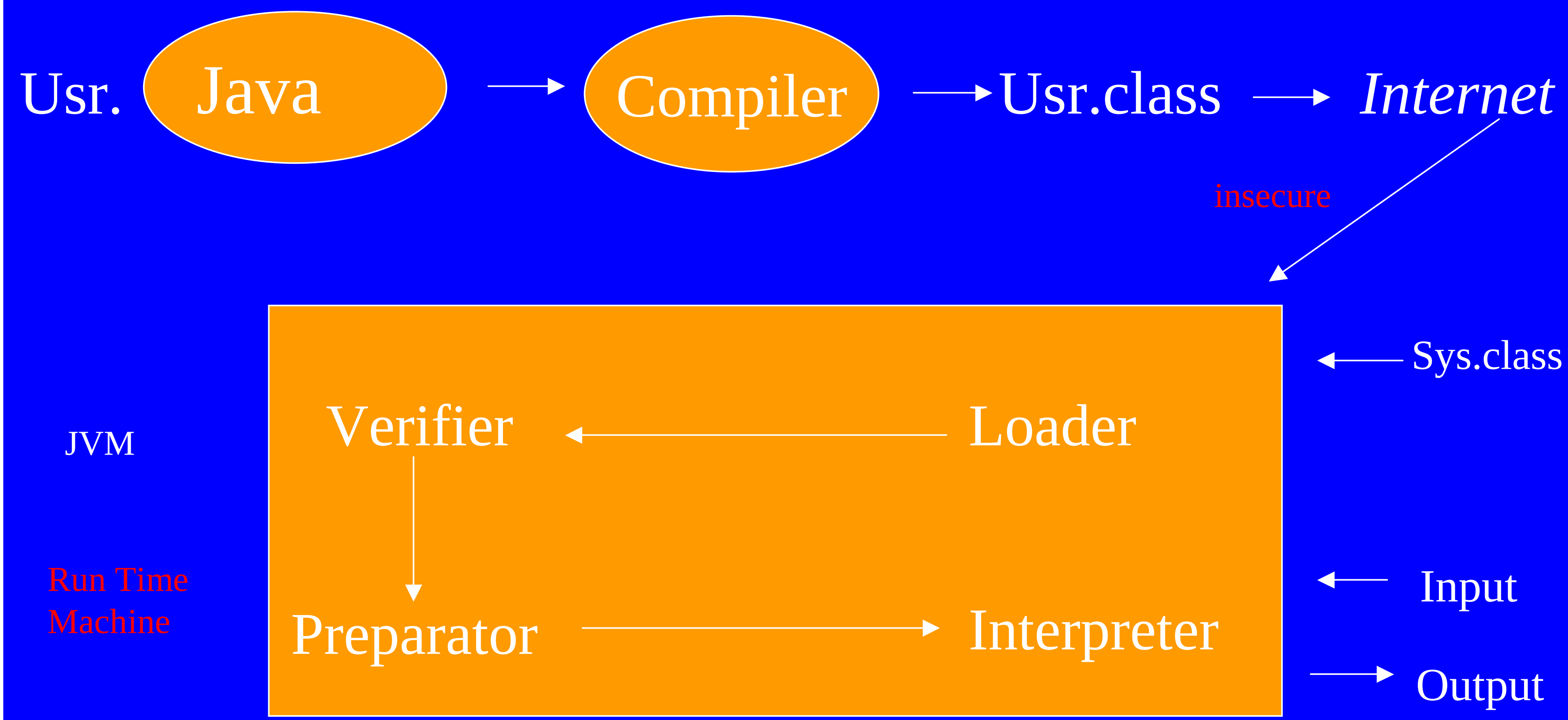
- refining & structuring models into a system (hierarchy) of (sub)models, modularizing orthogonal design decisions (“for change”), justifying them as correct
  - linking the ground model to the implementation
  - documenting the entire design for reuse and maintenance

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.



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)

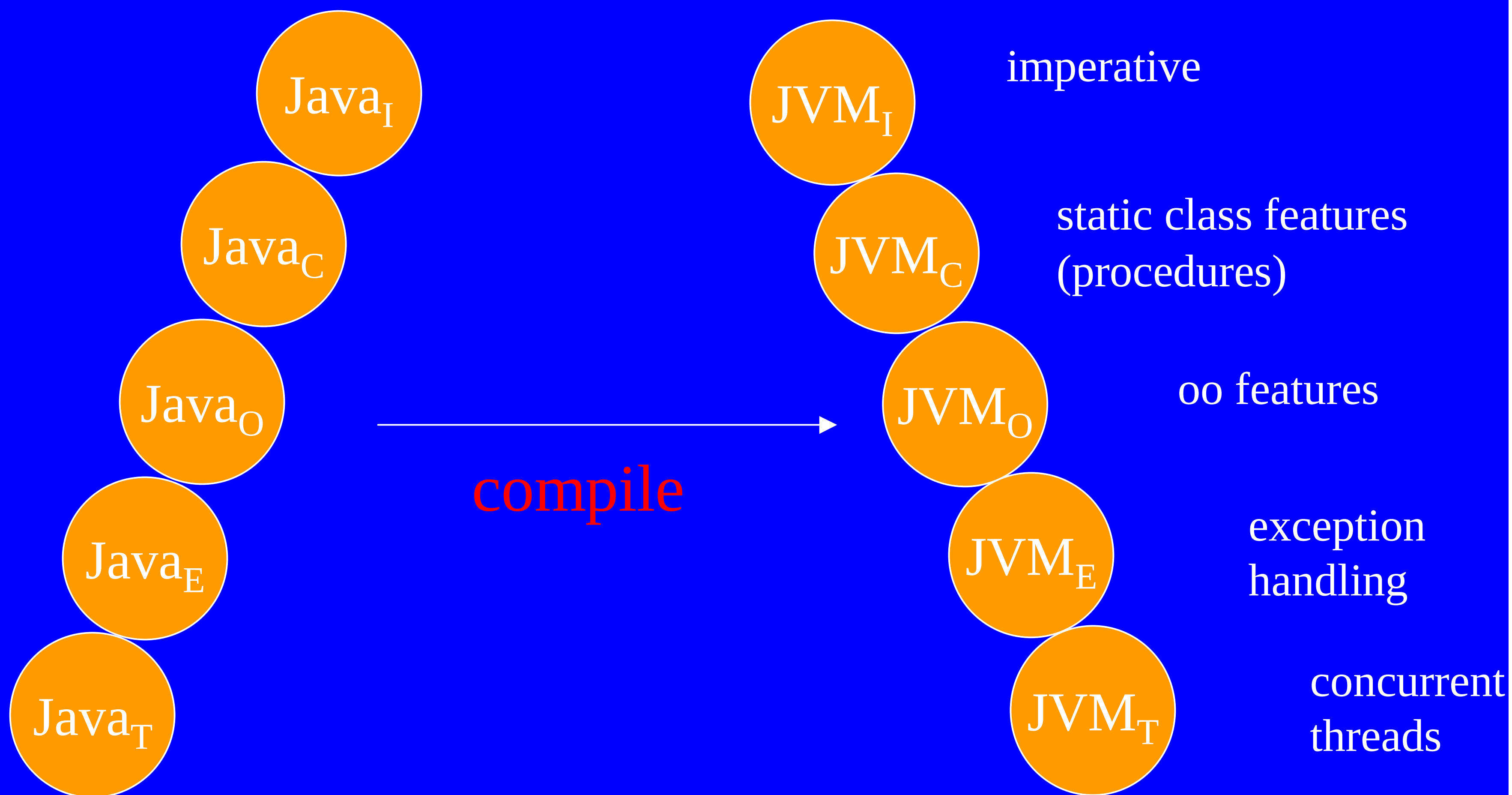
Main Result

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 =

$\mathbf{execJava}_I$	**imperative control constructs**
$\mathbf{execJava}_C$	**static class features (modules)**
$\mathbf{execJava}_O$	**oo features**
$\mathbf{execJava}_E$	**exception handling**
$\mathbf{execJava}_T$	**concurrent threads**

$\mathbf{execJava}_I$ =

$\mathbf{execJavaExp}_I$	**expression evaluation**
$\mathbf{execJavaStm}_I$	**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 ...) **s**tands for s = f (...t...) & pos = ▸ & restbody(pos)=t

Phrase: exps & block stms **Val**: values **Abr**: reasons for abruption

$$
\begin{array}{l}
\mathit{execJavaExp}_I = \textbf{case } \mathit{context}(\mathit{pos}) \textbf{ of} \\
\quad \mathit{lit} \longrightarrow \mathit{yield}(\mathit{JLS}(\mathit{lit})) \\
\\
\quad \mathit{loc} \longrightarrow \mathit{yield}(\mathit{locals}(\mathit{loc})) \\
\\
\quad \mathit{uop}\ {}^{\alpha}\mathit{exp} \longrightarrow \mathit{pos} := \alpha \\
\quad \mathit{uop}\ {}^{\blacktriangleright}\mathit{val} \longrightarrow \mathit{yieldUp}(\mathit{JLS}(\mathit{uop}, \mathit{val})) \\
\\
\quad {}^{\alpha}\mathit{exp}_1\ \mathit{bop}\ {}^{\beta}\mathit{exp}_2 \longrightarrow \mathit{pos} := \alpha \\
\quad {}^{\blacktriangleright}\mathit{val}\ \mathit{bop}\ {}^{\beta}\mathit{exp} \quad \longrightarrow \mathit{pos} := \beta \\
\quad {}^{\alpha}\mathit{val}_1\ \mathit{bop}\ {}^{\blacktriangleright}\mathit{val}_2 \ \longrightarrow \textbf{if } \neg(\mathit{bop} \in \mathit{divMod} \wedge \mathit{isZero}(\mathit{val}_2)) \textbf{ then} \\
\qquad\qquad\qquad\qquad\qquad \mathit{yieldUp}(\mathit{JLS}(\mathit{bop}, \mathit{val}_1, \mathit{val}_2)) \\
\\
\quad \mathit{loc} = {}^{\alpha}\mathit{exp} \longrightarrow \mathit{pos} := \alpha \\
\quad \mathit{loc} = {}^{\blacktriangleright}\mathit{val} \longrightarrow \mathit{locals} := \mathit{locals} \oplus \{(\mathit{loc}, \mathit{val})\} \\
\qquad\qquad\qquad\quad\ \mathit{yieldUp}(\mathit{val}) \\
\\
\quad {}^{\alpha}\mathit{exp}_0\ ?\ {}^{\beta}\mathit{exp}_1\ :\ {}^{\gamma}\mathit{exp}_2 \longrightarrow \mathit{pos} := \alpha \\
\quad {}^{\blacktriangleright}\mathit{val}\ ?\ {}^{\beta}\mathit{exp}_1\ :\ {}^{\gamma}\mathit{exp}_2 \ \longrightarrow \textbf{if } \mathit{val} \textbf{ then } \mathit{pos} := \beta \textbf{ else } \mathit{pos} := \gamma \\
\quad {}^{\alpha}\mathit{True}\ ?\ {}^{\blacktriangleright}\mathit{val}\ :\ {}^{\gamma}\mathit{exp} \ \longrightarrow \mathit{yieldUp}(\mathit{val}) \\
\quad {}^{\alpha}\mathit{False}\ ?\ {}^{\beta}\mathit{exp}\ :\ {}^{\blacktriangleright}\mathit{val} \ \longrightarrow \mathit{yieldUp}(\mathit{val})
\end{array}
$$

$$
\begin{array}{l}
execJavaStm_I = \textbf{case } context(pos) \textbf{ of} \\
\quad ; \qquad \to yield(Norm) \\
\quad {}^{\alpha}exp; \to pos := \alpha \\
\quad {}^{\blacktriangleright}val; \to yieldUp(Norm) \\
\\
\quad \texttt{break } lab; \qquad \to yield(Break(lab)) \\
\quad \texttt{continue } lab; \qquad \to yield(Continue(lab)) \\
\quad lab : {}^{\alpha}stm \qquad \to pos := \alpha \\
\quad lab : {}^{\blacktriangleright}Norm \qquad \to yieldUp(Norm) \\
\quad lab : {}^{\blacktriangleright}Break(lab_b) \qquad \to \textbf{if } lab = lab_b \textbf{ then } yieldUp(Norm) \\
\qquad\qquad\qquad\qquad \textbf{else } yieldUp(Break(lab_b)) \\
\quad lab : {}^{\blacktriangleright}Continue(lab_c) \to \textbf{if } lab = lab_c \textbf{ then } yield(body/pos) \\
\qquad\qquad\qquad\qquad \textbf{else } yieldUp(Continue(lab_c)) \\
\quad phrase({}^{\blacktriangleright}abr) \to \textbf{if } pos \neq firstPos \wedge propagatesAbr(restbody/up(pos)) \textbf{ then} \\
\qquad\qquad\qquad yieldUp(abr) \\
\\
\quad \{\} \qquad \to yield(Norm) \\
\quad \{{}^{\alpha_1}stm_1 \ldots {}^{\alpha_n}stm_n\} \qquad \to pos := \alpha_1 \\
\quad \{{}^{\alpha_1}Norm \ldots {}^{\blacktriangleright}Norm\} \qquad \to yieldUp(Norm) \\
\quad \{{}^{\alpha_1}Norm \ldots {}^{\blacktriangleright}Norm\, {}^{\alpha_{i+1}}stm_{i+1} \ldots {}^{\alpha_n}stm_n\} \to pos := \alpha_{i+1} \\
\\
\quad \texttt{if } ({}^{\alpha}exp)\, {}^{\beta}stm_1 \texttt{ else } {}^{\gamma}stm_2 \qquad \to pos := \alpha \\
\quad \texttt{if } ({}^{\blacktriangleright}val)\, {}^{\beta}stm_1 \texttt{ else } {}^{\gamma}stm_2 \qquad \to \textbf{if } val \textbf{ then } pos := \beta \textbf{ else } pos := \gamma \\
\quad \texttt{if } ({}^{\alpha}True)\, {}^{\blacktriangleright}Norm \texttt{ else } {}^{\gamma}stm \to yieldUp(Norm) \\
\quad \texttt{if } ({}^{\alpha}False)\, {}^{\beta}stm \texttt{ else } {}^{\blacktriangleright}Norm \to yieldUp(Norm) \\
\\
\quad \texttt{while } ({}^{\alpha}exp)\, {}^{\beta}stm \qquad \to pos := \alpha \\
\quad \texttt{while } ({}^{\blacktriangleright}val)\, {}^{\beta}stm \qquad \to \textbf{if } val \textbf{ then } pos := \beta \textbf{ else } yieldUp(Norm) \\
\quad \texttt{while } ({}^{\alpha}True)\, {}^{\blacktriangleright}Norm \to yieldUp(body/up(pos)) \\
\\
\quad Type\ x; \to yield(Norm)
\end{array}
$$

propagatesAbr iff phrase is no: labeled stm

Later refined : no static initializer try stm finally stm synchronized stm

The $\text{execJava}_{C/O}$ extensions

execJava_C =

execJavaExp_C extending expression evaluation

execJavaStm_C extending statement execution

Adding

- **class fields (global variables)**
- **class method invocation/return (procedures)**
- **class initializers (module initializers)**

execJava_O =

execJavaExp_O adding instance fields/methods

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 $\text{Java}_{\mathbf{O}}$)

$$
\begin{array}{l}
execJavaExp_C = \textbf{case } context(pos) \textbf{ of} \\
\quad \begin{array}{lll}
c.f & \to & \textbf{if } initialized(c) \textbf{ then } yield(globals(c/f)) \textbf{ else } initialize(c) \\
c.f = {}^{\alpha} exp & \to & pos := \alpha \\
c.f = {}^{\blacktriangleright} val & \to & \textbf{if } initialized(c) \textbf{ then} \\
 & & \quad globals(c/f) := val \\
 & & \quad yieldUp(val) \\
 & & \textbf{else } initialize(c)
\end{array}
\end{array}
$$

$$
\begin{array}{lll}
c.m^{\alpha}(exps) & \to & pos := \alpha \\
c.m^{\blacktriangleright}(vals) & \to & \textbf{if } initialized(c) \textbf{ then } invoke(up(pos), c/m, vals) \\
 & & \textbf{else } initialize(c)
\end{array}
$$

$$
\begin{array}{ll}
() & \to yield([\,]) \\
({}^{\alpha_1} exp_1, \ldots, {}^{\alpha_n} exp_n) & \to pos := \alpha_1 \\
({}^{\alpha_1} val_1, \ldots, {}^{\blacktriangleright} val_n) & \to yieldUp([val_1, \ldots, val_n]) \\
({}^{\alpha_1} val_1, \ldots, {}^{\blacktriangleright} val_i, {}^{\alpha_{i+1}} exp_{i+1} \ldots {}^{\alpha_n} exp_n) & \to pos := \alpha_{i+1}
\end{array}
$$

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.

$$
\begin{array}{l}
execJavaStm_C = \textbf{case } context(pos) \textbf{ of} \\
\quad \texttt{static}\ {}^{\alpha}stm \rightarrow \textbf{let } c = classNm(meth) \\
\qquad\qquad\qquad \textbf{if } c = \texttt{Object} \vee initialized(super(c)) \textbf{ then } pos := \alpha \\
\qquad\qquad\qquad \textbf{else } initialize(super(c)) \\
\quad \texttt{static}\ {}^{\blacktriangleright}Return \rightarrow exitMethod(Norm) \\
\qquad\qquad\qquad classState(classNm(meth)) := Initialized \\
\\
\quad \texttt{return}\ {}^{\alpha}exp; \rightarrow pos := \alpha \\
\quad \texttt{return}\ {}^{\blacktriangleright}val; \rightarrow yieldUp(Return(val)) \\
\quad \texttt{return}; \rightarrow yield(Return) \\
\quad lab : {}^{\blacktriangleright}Return \rightarrow yieldUp(Return) \\
\quad lab : {}^{\blacktriangleright}Return(val) \rightarrow yieldUp(Return(val)) \\
\quad Return \rightarrow \textbf{if } pos = firstPos \wedge \neg null(frames) \textbf{ then} \\
\qquad\qquad\qquad exitMethod(Norm) \\
\quad Return(val) \rightarrow \textbf{if } pos = firstPos \wedge \neg null(frames) \textbf{ then} \\
\qquad\qquad\qquad exitMethod(val) \\
\\
\quad {}^{\blacktriangleright}Norm; \rightarrow yieldUp(Norm)
\end{array}
$$

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)

$execJavaExp_O = \textbf{case}\ context(pos)\ \textbf{of}$

$\quad \texttt{this} \rightarrow yield(locals(\texttt{"this"}))$

$\quad \texttt{new}\ c/m^{\alpha}(exps) \rightarrow \textbf{if}\ initialized(c)\ \textbf{then}\ pos := \alpha\ \textbf{else}\ initialize(c)$

$\quad \texttt{new}\ c/m^{\blacktriangleright}(vals) \rightarrow \textbf{create}\ ref$

$\qquad heap(ref) := Object(c, \{(f, defaultVal(type(f))) \mid f \in instanceFields(c)\})$

$\qquad invoke(up(pos), c/m, [ref] \cdot vals)$

$\quad {}^{\alpha}exp.c/f \rightarrow pos := \alpha$

$\quad {}^{\blacktriangleright}ref.c/f \rightarrow \textbf{if}\ ref \neq null\ \textbf{then}\ yieldUp(getField(ref, c/f))$

$\quad {}^{\alpha}exp_1.c/f = {}^{\beta}exp_2 \rightarrow pos := \alpha$

$\quad {}^{\blacktriangleright}ref.c/f = {}^{\beta}exp \rightarrow pos := \beta$

$\quad {}^{\alpha}ref.c/f = {}^{\blacktriangleright}val \rightarrow \textbf{if}\ ref \neq null\ \textbf{then}$

$\qquad setField(ref, c/f, val)$

$\qquad yieldUp(val)$

$\quad {}^{\alpha}exp\ \texttt{instanceof}\ c \rightarrow pos := \alpha$

$\quad {}^{\blacktriangleright}ref\ \texttt{instanceof}\ c \rightarrow yieldUp(ref \neq null \wedge classOf(ref) \preceq_h c)$

$\quad (c)^{\alpha}exp \rightarrow pos := \alpha$

$\quad (c)^{\blacktriangleright}ref \rightarrow \textbf{if}\ ref = null \vee classOf(ref) \preceq_h c\ \textbf{then}\ yieldUp(ref)$

$\quad {}^{\alpha}exp.c/m^{\beta}(exps) \rightarrow pos := \alpha$

$\quad {}^{\blacktriangleright}ref.c/m^{\beta}(exps) \rightarrow pos := \beta$

$\quad {}^{\alpha}ref.c/m^{\blacktriangleright}(vals) \rightarrow \textbf{if}\ ref \neq null\ \textbf{then}$

$\qquad \textbf{let}\ c' = \textbf{case}\ callKind(up(pos))\ \textbf{of}$

$\qquad\qquad Virtual \rightarrow lookup(classOf(ref), c/m)$

$\qquad\qquad Super \rightarrow lookup(super(classNm(meth)), c/m)$

$\qquad\qquad Special \rightarrow c$

$\qquad invoke(up(pos), c'/m, [ref] \cdot vals)$

The execJava$_{E/T}$ extensions

execJava$_{E}$ =

execJavaExp$_{E}$ **for evaluation of run-time exceptions**

execJavaStm$_{E}$ **for execution of exception statements**

execJava$_{T}$ =

execJavaStm$_{T}$ **for synchronization statements**

(as part of execJavaThread)

Abrs in try stms: caught excs lead to catch code exec, othr abrs propagate

catch code yields up Norm or an abr

For finally stms: abrs 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

$$
\begin{array}{l}
execJavaStm_E = \textbf{case } context(pos) \textbf{ of}\\
\quad \texttt{throw}\ {}^{\alpha}exp; \rightarrow pos := \alpha\\
\quad \texttt{throw}\ {}^{\blacktriangleright}ref; \rightarrow\\
\qquad \textbf{if } ref = null \textbf{ then } fail(\texttt{NullPointerException}) \textbf{ else } yieldUp(Exc(ref))\\
\\
\quad \texttt{try}\ {}^{\alpha}stm\ \texttt{catch} \ldots \rightarrow pos := \alpha\\
\quad \texttt{try}\ {}^{\blacktriangleright}Norm\ \texttt{catch} \ldots \rightarrow yieldUp(Norm)\\
\quad \texttt{try}\ {}^{\blacktriangleright}Exc(ref)\ \texttt{catch}\ (c_1\ x_1)\ {}^{\beta_1}stm_1 \ldots \texttt{catch}\ (c_n\ x_n)\ {}^{\beta_n}stm_n \rightarrow\\
\qquad \textbf{if } \exists 1 \le j \le n : classOf(ref) \preceq_{\mathrm{h}} c_j \textbf{ then}\\
\qquad\quad \textbf{let } j = \min\{i \mid classOf(ref) \preceq_{\mathrm{h}} c_i\}\\
\qquad\quad pos := \beta_j\\
\qquad\quad locals := locals \oplus \{(x_j, ref)\}\\
\qquad \textbf{else } yieldUp(Exc(ref))\\
\quad \texttt{try}\ {}^{\blacktriangleright}abr\ \texttt{catch}\ (c_1\ x_1)\ {}^{\beta_1}stm_1 \ldots \texttt{catch}\ (c_n\ x_n)\ {}^{\beta_n}stm_n \rightarrow yieldUp(abr)\\
\quad \texttt{try}\ {}^{\alpha}Exc(ref) \ldots \texttt{catch}\ (c_i\ x_i)\ {}^{\blacktriangleright}Norm \ldots \rightarrow yieldUp(Norm)\\
\quad \texttt{try}\ {}^{\alpha}Exc(ref) \ldots \texttt{catch}\ (c_i\ x_i)\ {}^{\blacktriangleright}abr \ldots \rightarrow yieldUp(abr)\\
\\
\quad {}^{\alpha}stm_1\ \texttt{finally}\ {}^{\beta}stm_2 \rightarrow pos := \alpha\\
\quad {}^{\blacktriangleright}Norm\ \texttt{finally}\ {}^{\beta}stm \rightarrow pos := \beta\\
\quad {}^{\blacktriangleright}abr\ \texttt{finally}\ {}^{\beta}stm \rightarrow pos := \beta\\
\quad {}^{\alpha}s\ \texttt{finally}\ {}^{\blacktriangleright}Norm \rightarrow yieldUp(s)\\
\quad {}^{\alpha}s\ \texttt{finally}\ {}^{\blacktriangleright}abr \rightarrow yieldUp(abr)\\
\\
\quad Exc(ref) \rightarrow \textbf{if } pos = firstPos \wedge \neg null(frames) \textbf{ then}\\
\qquad\qquad exitMethod(Exc(ref))\\
\\
\quad lab : {}^{\blacktriangleright}Exc(ref) \rightarrow yieldUp(Exc(ref))\\
\quad \texttt{static}\ {}^{\blacktriangleright}Exc(ref) \rightarrow\\
\qquad classState(classNm(meth)) := Unusable\\
\qquad \textbf{if } classOf(ref) \preceq_{\mathrm{h}} \texttt{Error} \textbf{ then } exitMethod(Exc(ref))\\
\qquad\quad \textbf{else } fail(\texttt{ExceptionInInitializerErr})
\end{array}
$$

Examples of run-time exceptions

$$
\begin{array}{l}
\mathit{execJavaExp}_E = \mathbf{case}\ \mathit{context}(\mathit{pos})\ \mathbf{of} \\
\quad {}^{\alpha}\mathit{val}_1\ \mathit{bop}\ {}^{\blacktriangleright}\mathit{val}_2 \quad \rightarrow \mathbf{if}\ \mathit{bop} \in \mathit{divMod} \wedge \mathit{isZero}(\mathit{val}_2)\ \mathbf{then} \\
\qquad\qquad\qquad\qquad\qquad \mathit{fail}(\mathtt{ArithmeticException}) \\
\quad {}^{\blacktriangleright}\mathit{ref}.c/f \qquad\qquad\ \rightarrow \mathbf{if}\ \mathit{ref} = \mathit{null}\ \mathbf{then}\ \mathit{fail}(\mathtt{NullPointerException} \\
\quad {}^{\alpha}\mathit{ref}.c/f = {}^{\blacktriangleright}\mathit{val} \ \rightarrow \mathbf{if}\ \mathit{ref} = \mathit{null}\ \mathbf{then}\ \mathit{fail}(\mathtt{NullPointerException} \\
\quad {}^{\alpha}\mathit{ref}.c/m^{\blacktriangleright}(\mathit{vals}) \rightarrow \mathbf{if}\ \mathit{ref} = \mathit{null}\ \mathbf{then}\ \mathit{fail}(\mathtt{NullPointerException} \\
\quad (c)^{\blacktriangleright}\mathit{ref} \qquad\qquad\ \rightarrow \mathbf{if}\ \mathit{ref} \neq \mathit{null} \wedge \mathit{classOf}(\mathit{ref}) \not\preceq_{\mathrm{h}} c\ \mathbf{then} \\
\qquad\qquad\qquad\qquad\qquad \mathit{fail}(\mathtt{ClassCastException})
\end{array}
$$

where **fail (exc)** = yield (throw new exc() ;)

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

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**

$$
\begin{array}{l}
execJavaStm_T = \textbf{case } context(pos) \textbf{ of} \\
\quad \texttt{synchronized } ({}^{\alpha} exp)\ {}^{\beta} stm \rightarrow pos := \alpha \\
\quad \texttt{synchronized } ({}^{\blacktriangleright} ref)\ {}^{\beta} stm \rightarrow \\
\quad\quad \textbf{if } ref = null \textbf{ then } fail(\texttt{NullPointerException}) \\
\quad\quad \textbf{else} \\
\quad\quad\quad \textbf{if } ref \in sync(thread) \textbf{ then} \\
\quad\quad\quad\quad sync(thread) := [ref] \cdot sync(thread) \\
\quad\quad\quad\quad locks(ref) \quad := locks(ref) + 1 \\
\quad\quad\quad\quad pos \quad := \beta \\
\quad\quad\quad \textbf{else} \\
\quad\quad\quad\quad exec(thread) \quad := Synchronizing \\
\quad\quad\quad\quad syncObj(thread) := ref \\
\quad\quad\quad\quad cont(thread) \quad := (frames, (meth, restbody, \beta, locals)) \\
\quad \texttt{synchronized } ({}^{\alpha} ref)\ {}^{\blacktriangleright} Norm \rightarrow releaseLock(Norm) \\
\quad \texttt{synchronized } ({}^{\alpha} ref)\ {}^{\blacktriangleright} abr \quad \rightarrow releaseLock(abr) \\
\\
\quad \texttt{static } {}^{\blacktriangleright} abr \rightarrow notifyThreadsWaitingForInitialization \\
\quad abr \rightarrow \textbf{if } pos = firstPos \wedge null(frames) \textbf{ then } killThread
\end{array}
$$

Abstract scheduling of Multiple Threads: inserting execJava into ExecJavaThread

Thread scheduling separated from thread execution

ExecJavaThread ≡

choose q **in** *dom(exec), runnable(q)*

if *q=thread and exec(q)=Active*

then execJava

else

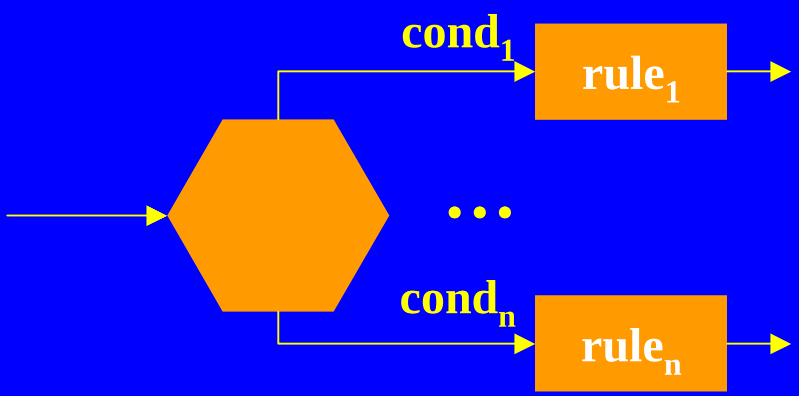
if *exec(q)=Active* **then**

cont(thread) := (frames,(methd,restbody,pos,locals))

thread := q

run(q)

Diagram notation for Control State ASMs



UML: combined branching/action nodes

meaning

if ctl = i then

if $cond_1$ then $rule_1$

$ctl:=j_1$

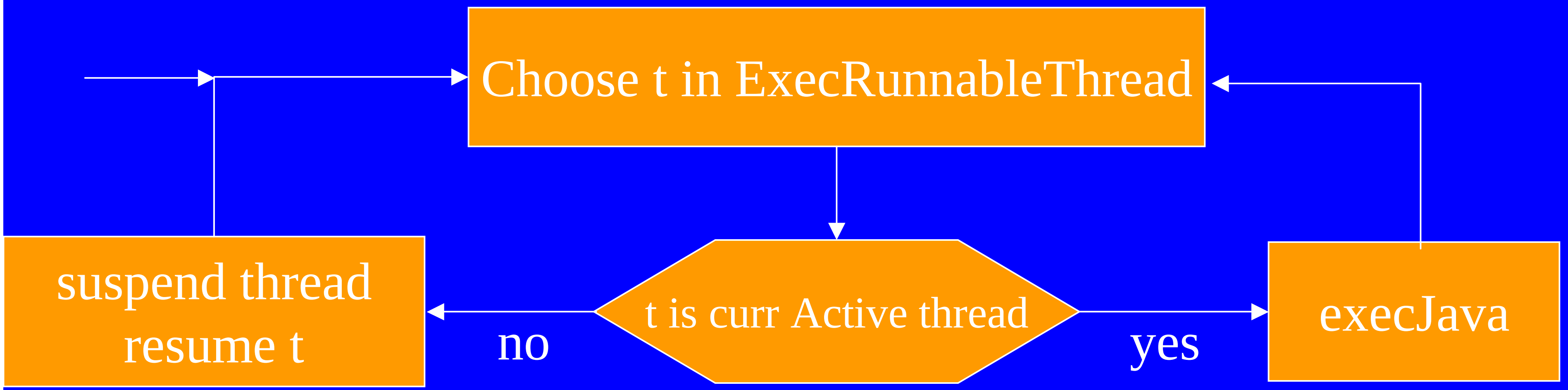
….

if $cond_n$ then $rule_n$

$ctl:=j_n$

labeling of the arrows by "control" states often suppressed

Defining execJavaThread as control state ASM



Thread scheduling separated from thread execution

t in ExecRunnableThread = (t in dom(exec) & runnable(t))

t is curr Active thread = (t = thread & exec(t) = Active)

suspend thread = if exec(thread) = Active
then cont(thread) := (frames, currframe)

resume(t) = thread := t
run(t)

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.

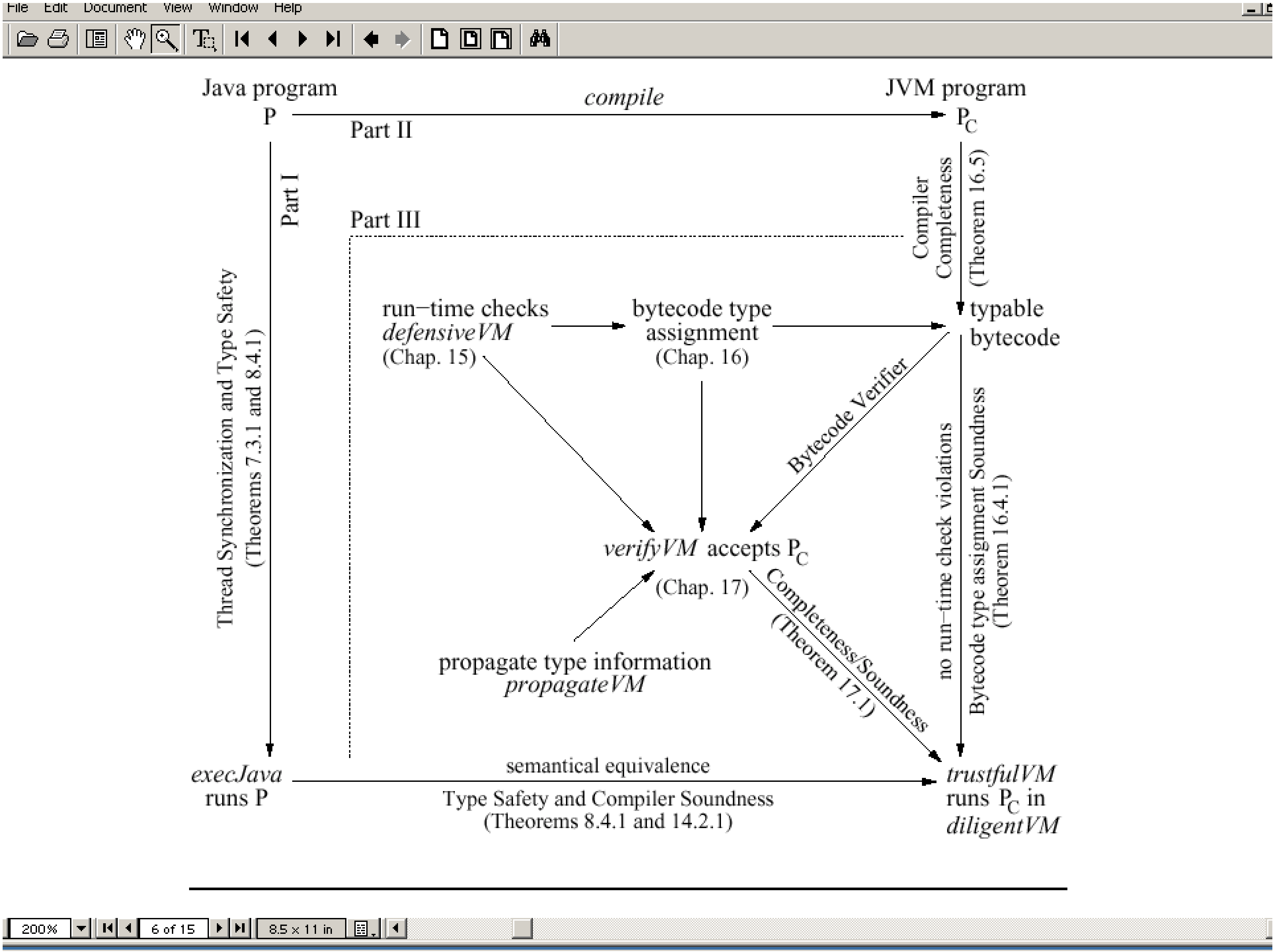
Theorem: Correctness of Thread Synchronization in Java (Cont'd)

- 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.
- …

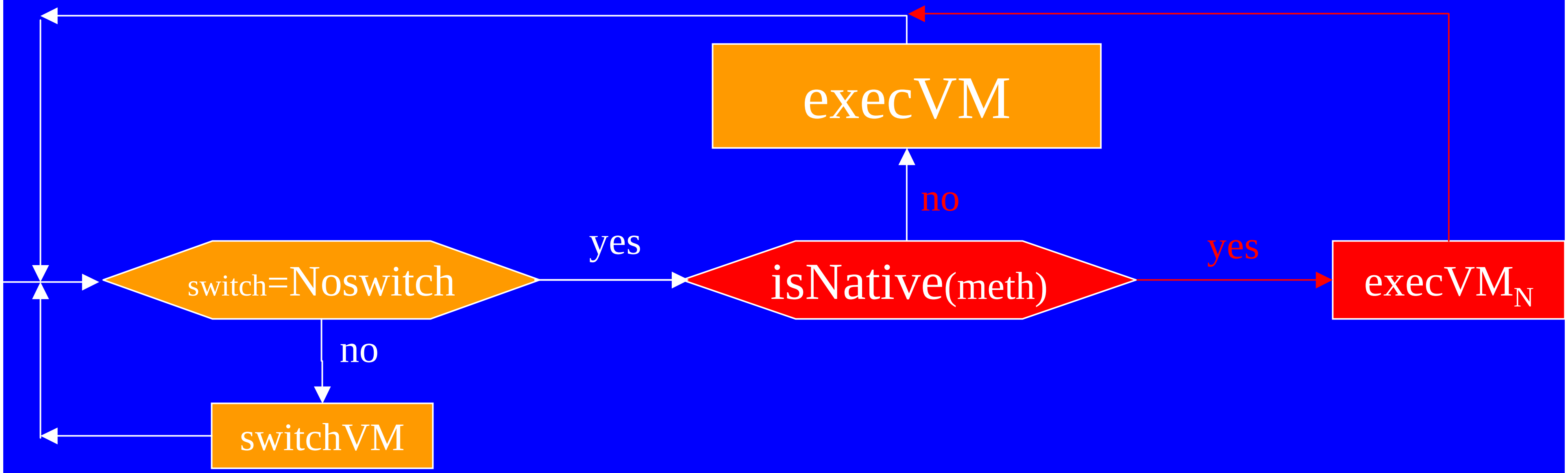
PROOF. Induction on Java ASM runs.

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


File Edit Document View Window Help
Java program
P
compile
JVM program
P_C
Part II
Part I
Part III
Compiler Completeness
(Theorem 16.5)
Thread Synchronization and Type Safety
(Theorems 7.3.1 and 8.4.1)
run−time checks
defensiveVM
(Chap. 15)
bytecode type assignment
(Chap. 16)
typable bytecode
Bytecode Verifier
no run−time check violations
Bytecode type assignment Soundness
(Theorem 16.4.1)
verifyVM accepts P_C
(Chap. 17)
Completeness/Soundness
(Theorem 17.1)
propagate type information
propagateVM
execJava
runs P
semantical equivalence
Type Safety and Compiler Soundness
(Theorems 8.4.1 and 14.2.1)
trustfulVM
runs P_C in
diligentVM
200%
6 of 15
8.5 x 11 in

Stepwise refinement of trustfulVM



execVM and switchVM incrementally extended (language driven)

$trustfulVM_I$ = $execVM_I \subseteq execVM_C \subseteq execVM_O \subseteq execVM_E$
$execVM_N \subseteq execVM_D$ defining instructionwise changes of current frame

$switchVM_C \subseteq switchVM_E \subseteq switchVM_D$ 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

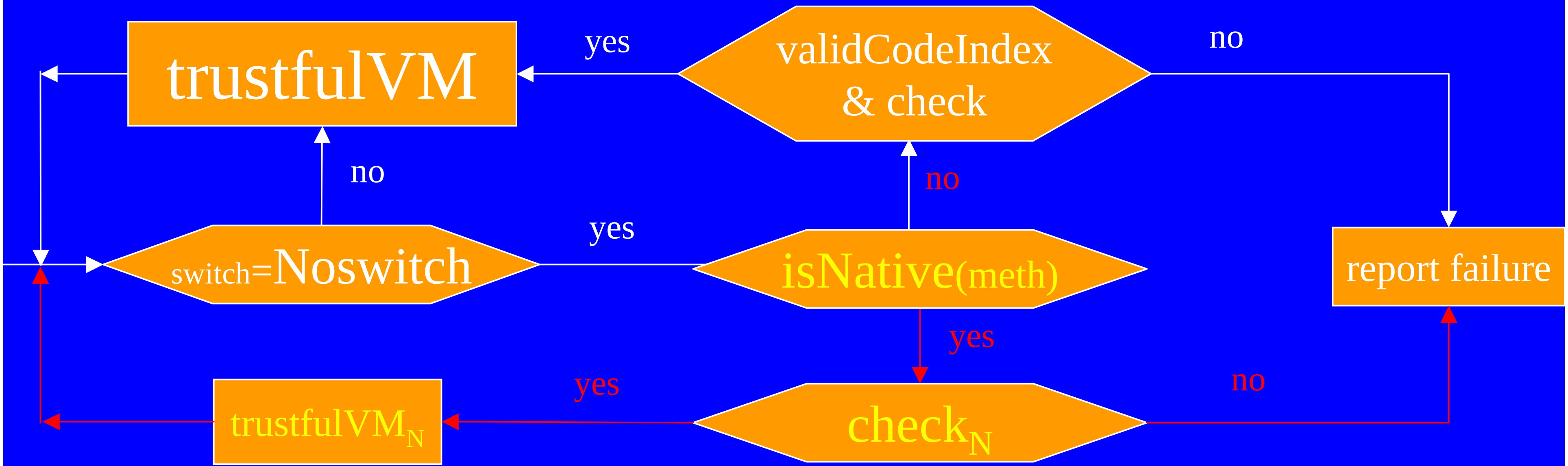
- With respect to the ASM models for Java and JVM, and wrt the definition of compile from Java to JVM code, including the exception table, the execution of P in Java and the execution of compile(P) in Trustful VM are equivalent (in a sense made precise), for arbitrary pgms P.
- PROOF. By induction on the runs of the Java/JVM ASMs, using the type safety theorem.
- NB. This inlcudes the correctness of exception handling

see Börger E., Schulte W., A Practical Method for Specification and Analysis of Exception Handling -- A Java/JVM Case Study. IEEE Transactions of Software Engineering, Vol.26, No.10, October 2000 (Special Issue on Exception Handling, eds. D.Perry, A.Romanovsky, A.Tripathi).

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. check$_I$ extended by check$_C$
extended by check$_O$
extended by check$_E$
extended by check$_N$
extended by check$_D$

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)

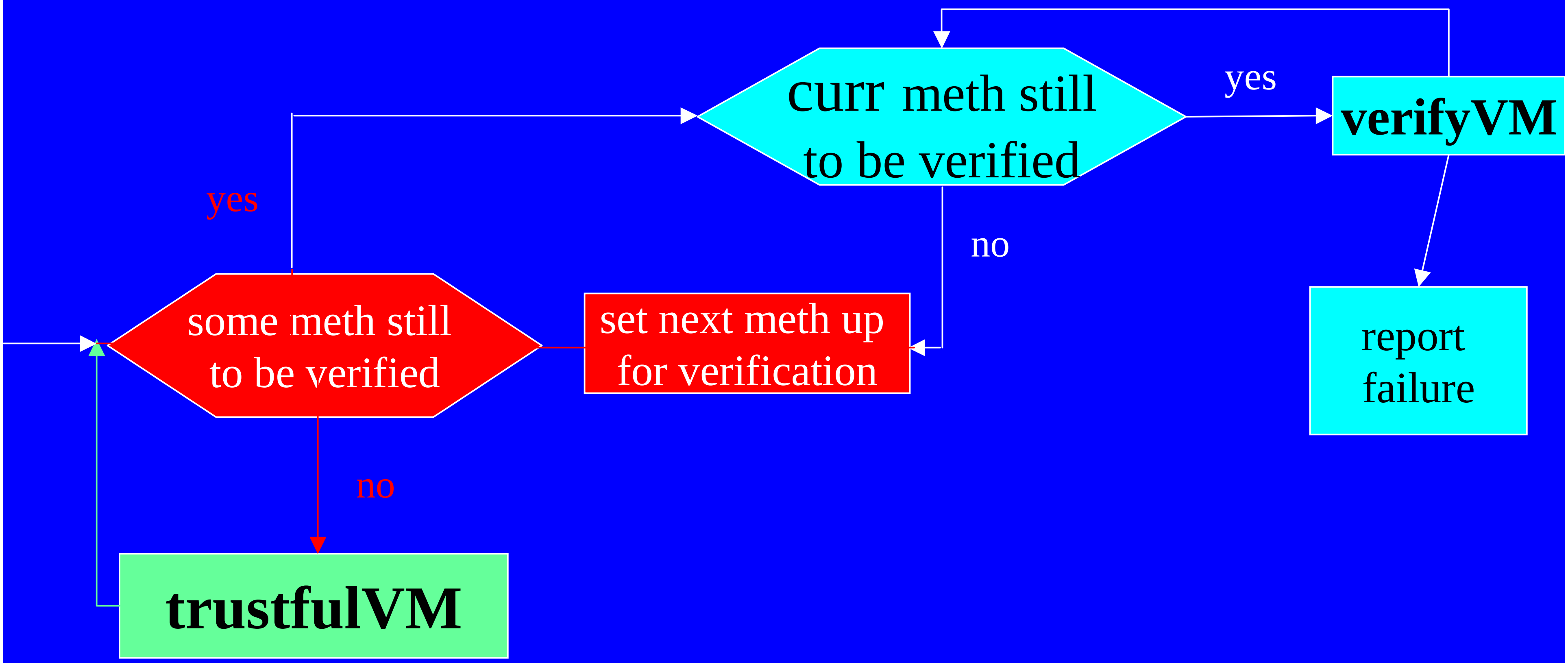
- Soundness Theorem: If P satisfies the type assignment conditions, then Defensive VM executes P without violating any run-time check.

  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

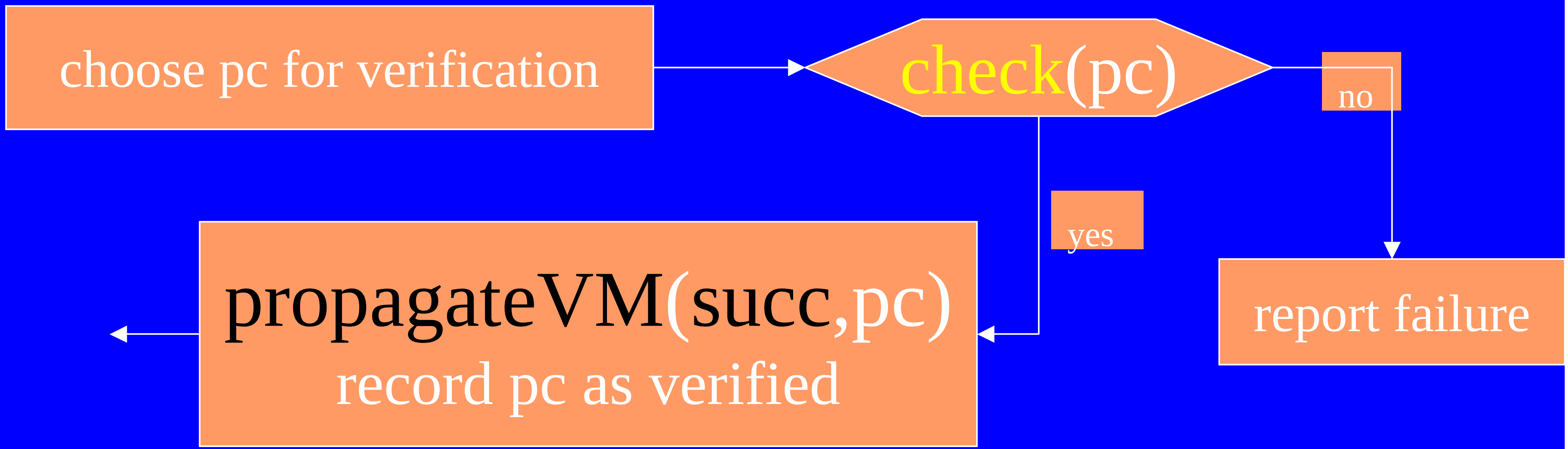
Stepwise refinement of diligentVM$_{I,C,O,E}$



switchVM$_C$ in trustfulVM is refined to also link classes before their initialization, where the linking submachine triggers verifyVM

verifyVM decomped into lang layered check, succ, propagate

Stepwise refinement of verifyVM



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

propagateVM and succ incrementally extended

$succ_I \subseteq succ_C \subseteq succ_O \subseteq succ_E$

$propagate_I \subseteq propagate_E$

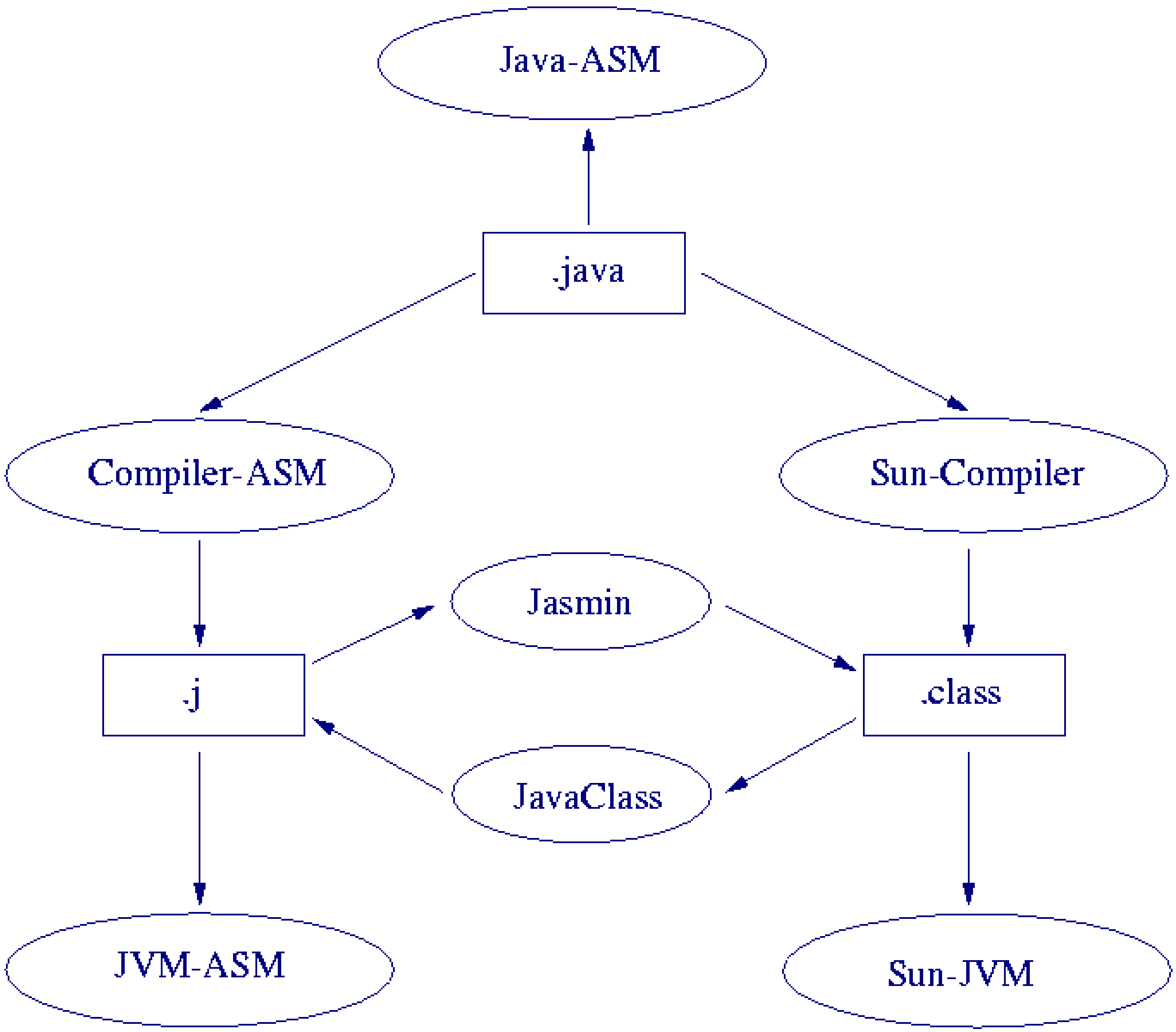
Proving Bytecode Verifier Complete and Correct

- 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 compile(P) (in textual representation, using JASMIN to convert to binary class format)
  - to execute the bytecode programs compile(P)

  E.g. our Bytecode Verifier rejects Saraswat’s program
- Developed by Joachim Schmid, available at `www.tydo.de/AsmGofer`


Java-ASM
.java
Compiler-ASM
Sun-Compiler
Jasmin
.j
.class
JavaClass
JVM-ASM
Sun-JVM

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/

For ASMGofer see **`www.tydo.de/AsmGofer/`**

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