Egon Börger (Pisa)

The Abstract State Machines Method

for Modular Design and Analysis of Programming Languages

A Survey

Università di Pisa, Dipartimento di Informatica, I-56127 Pisa, Italy
boerger@di.unipi.it

Evolution of the ASM method

- 1984-1995/2000: *Foundational concern*: sharpen Church-Turing thesis by "an alternative computation model which explicitly recognizes finiteness of computers" (Gurevich 1984)
  - finding an appropriate definition of ASM
- Fall 1989-1992: *Recognition of practical potential* of ASM concept for building and analyzing reliable ground models and their provably correct ASM refinements to executable code ('bridge the gap')
  - experiments with ASM models to relate in a verifiable way semantics of programming languages to their implementation
- Fall 1992-1995: *Scalability test* (test of ASM thesis) thru variety of case studies (architectures, hw, VMs, protocols, controller sw)
  - influenced the final definition of ASMs (Lipari Guide 1995)
- Since Fall 1995: *Integration of ASM method* into industrial software development environments
  - tool support by exec/debug engines, model checkers, thm provers

Notion of ASM: generalizing Finite State Machines

```
$FSM =$ if $Defined(in)$ then  // do in parallel!
    $ctl\_state := \delta(ctl\_state, in)$ // static function $\delta$
    $out := \lambda(ctl\_state, in)$ // static function $\lambda$
```

FSMs come with five characteristic restrictions:

- *only 3 locations*, furthermore 0-ary (variables without parameters):
  - in: *monitored* (only read by FSM, but written by environment)
  - ctl_state: *controlled* (read and written by FSM)
  - out: *output* (only written by FSM, but read by environment)
- *no* shared *locations* (mono-agent view: strict separation of in-/output)
- *only 2 simultaneous updates*
- *only 3 special data types*: finite sets of
  - input/output symbols (letters of an alphabet)
  - control states (labels/integers) representing bounded memory
- *only 2 background functions* (furthermore static) δ, λ

Notion of ASM: extend FSM states to abstract states

ASMs withdraw those restrictions, permitting in a machine

- to read and update in each step simultaneously (synch. parallelism)
  - *arbitrarily many* locations (instead of 2)
  - *parameterized* locations (‘array variables’)
  - *shared* locations (read/written by multiple agents)
- *arbitrary* data structures
  - location *values of arbitrary type*
  - *arbitrary background functions* (possibly dynamic and > 2)
  - *arbitrary* conditions as *rule guards* (not only input definedness)

This leads to the definition: ASM = finite set of rules

$$\textbf{if } Cond \textbf{ then } Updates$$

- $Updates$ is a set of simultaneous assignments $f(t_1, \ldots, t_n) := t$
- t_i, t arbitrary exps, $Cond$ arbitrary Boolean-valued exp

Notion of ground models

Accurate blueprints —'golden models' in semiconductor industry—of to-be-implemented piece of real world (here: pgg lg) which

- define 'the conceptual construct/the essence' of the software system (Brooks) prior to coding, *abstractly and rigorously*
  - at application-problem-determined (here: programming) level of detailing (*minimality*)
  - formulated in application domain (here: language user) terms (*precision*, informal accuracy)
  - authoritatively for the further development activities: design contract/process/evaluation and maintenance (*simplicity*)
- ground the design in reality by justifying the definition as
  - *correct*: model elems reliably convey original intentions (the manual)
  - *complete*: every semantically relevant feature is present, no gap in understanding of 'how to use' resp. 'what to build'
  - *consistent*: conflicting objectives in requirements identified/resolved

Ground model justification must solve three problems

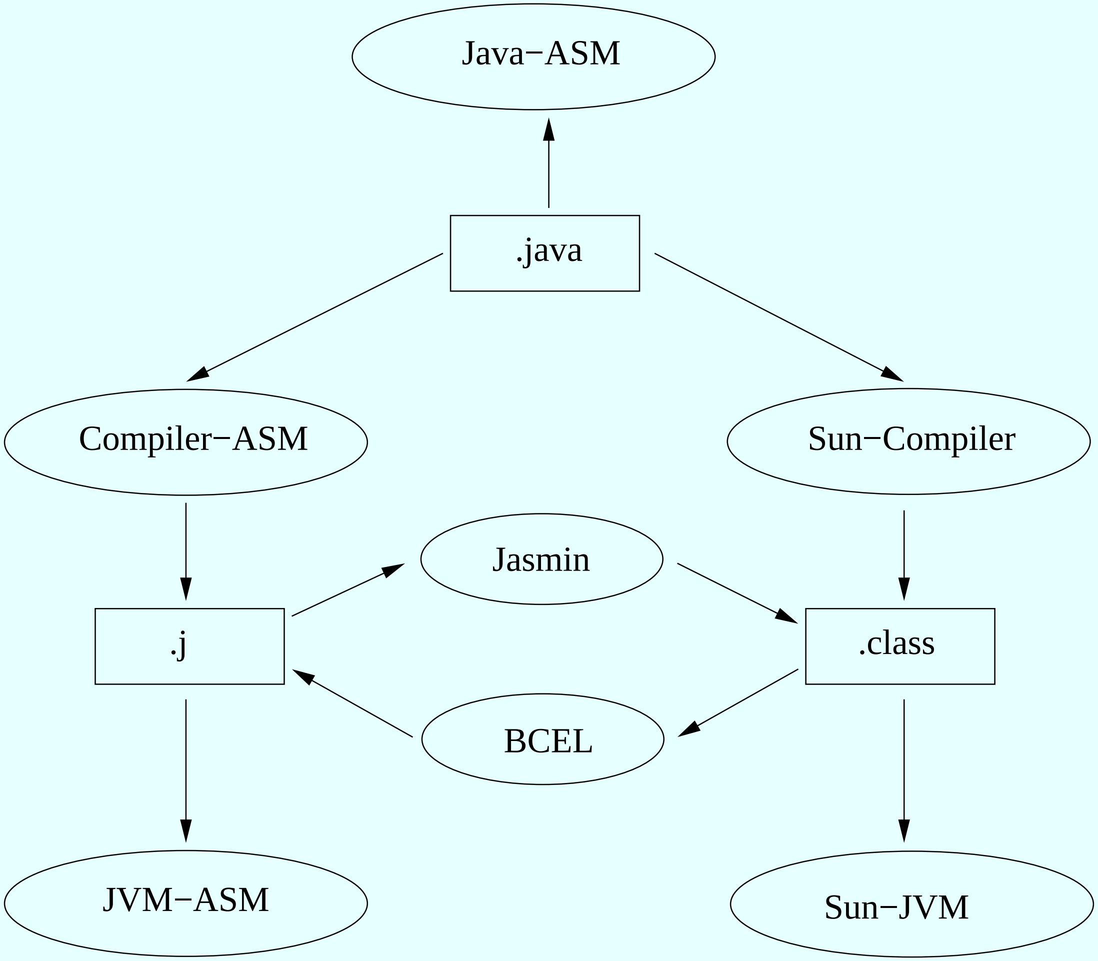
- Communication (language) problem: mediate between
  - sw designers, domain experts and customers for common understanding prior to coding of ‘precisely what to build’
  - problem domain and world of models, requiring
    - capability to calibrate degree of model precision to the problem
    - most general data type and interface concept
- Verification method problem: no infinite regress (Aristotle)
  - no math. transition from informal to precise descriptions, BUT
  - inspection can provide *evidence of direct correspondence* bw ground model and reality the model has to capture (completeness, correctness, empirical interpretation of extra-logical terms)
  - domain-specific reasoning can check consistency issues
- Validation problem: need for *repeatable experiments* to validate (falsify) model behaviour (runtime verification and analysis, testing)

Exls of ground model ASMs for programming languages

- ground model ASMs defining industrial *standards* of
  - ISO for Prolog: Börger/Rosenzweig: 1991-95
  - IEEE for VHDL93: Müller/Glässer/Börger:1994-95
  - ITU-T for SDL-2000: Glässer/Prinz et al. 1998-2003
  - ECMA for C#: Börger/Fruja/Gervasi/Stärk: TCS 336 (2006)
  - OASIS for BPEL: Farahbod et al. ASM'04 and IJBPMI 1 (2006)
  - OMG for BPMN (1.0/2.0): Börger/Thalheim/Sörensen 2007-11
- ground model ASMs as *basis for verifiably correct refinements* of language semantics to its implementation
  - Java/JVM (including bytecode verifier, see *JBook*) & C#/.NET CLR
  - Occam-to-Transputer: Börger/Durdanovic/Rosenzweig: 1994-96

  including machine verification of Prolog-to-WAM compilation scheme using KIV(Schellhorn/Ahrendt 1997-98) and of compiler front/back-ends using PVS (Goos/Langmaack/von Henke 1996-2000)

ExI: Mixing execution engines for model validation



Notion of ASM refinement: freedom to define:

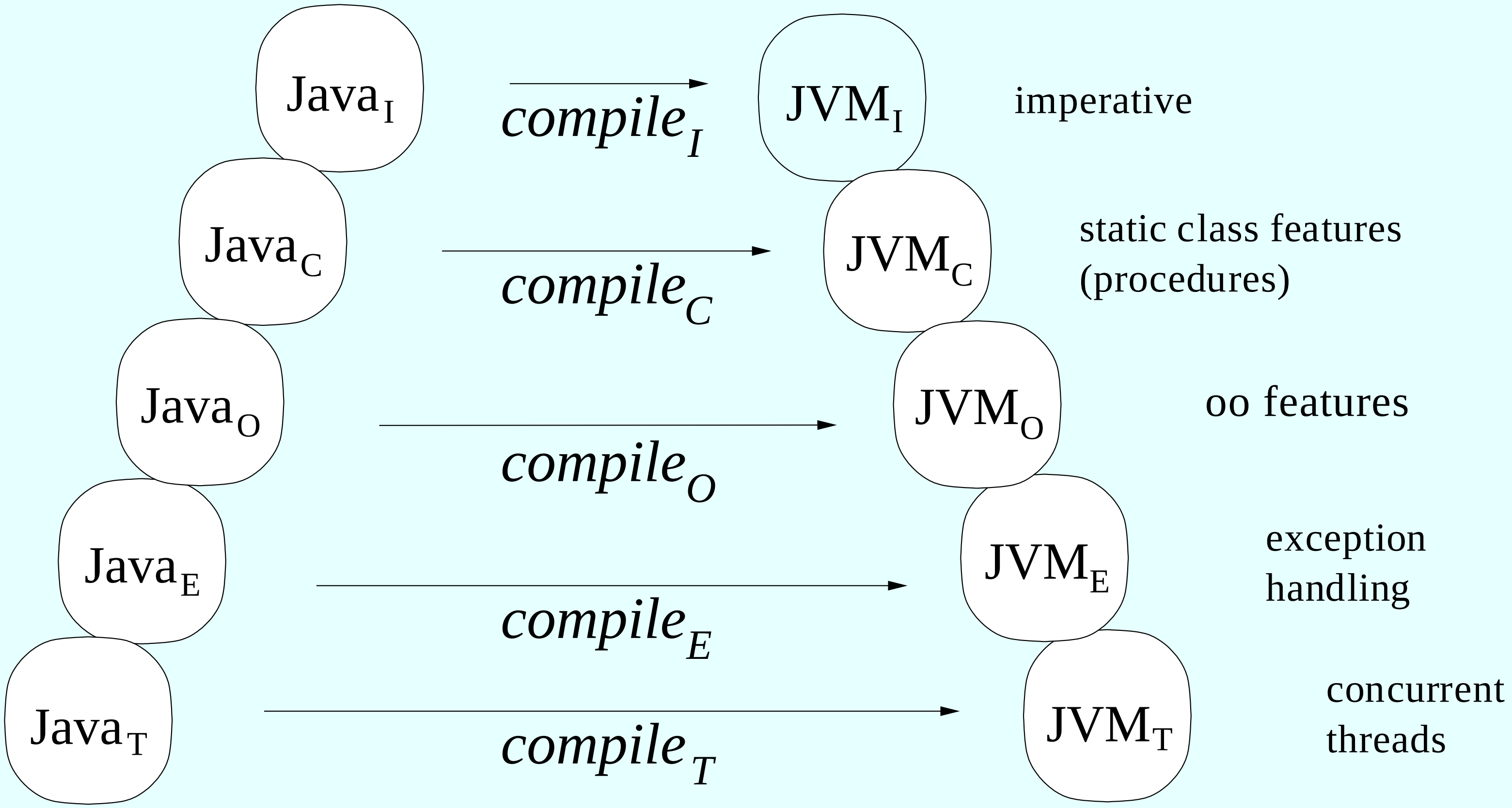
- *abstract/refined state*
- states of interest and correspondence bw pairs (S, S^*) of abstract/refined states of interest
- abstract/refined computation segments of m/n single abstract/refined steps τ_i/σ_j leading from/to corresponding states of interest
- *locations of interest* and *corresponding* abstract/refined locs of interest
- equivalence of values in corresponding locations of interest

$$\begin{tikzcd}
\text{State } S \arrow[r, "\overbrace{\tau_1 \cdots \tau_m}^{m \text{ steps of } M}"] \arrow[d, leftrightarrow, "\equiv"'] & S' \arrow[d, leftrightarrow, "\equiv"] \\
\text{State } S^* \arrow[r, "\underbrace{\sigma_1 \cdots \sigma_n}_{n \text{ steps of } M^*}"'] & S^{*\prime}
\end{tikzcd}$$

Main usages of ASM refinements

- construct hierarchical levels for
  - horizontal piecemeal extensions and adaptations (*design for change*)
    - e.g. of ISO Prolog model by constraints (Prolog III), polymorphism (Protos-L), narrowing (Babel), o-orientation, parallelism (Parlog, Concurrent Prolog, Pandora), abstract execution strategy (Gödel)
  - (provably correct) vertical stepwise detailing of models (*design for reuse*) to their implementation, e.g. model chains leading from
    - Prolog to WAM (13 levels), Occam to Transputer (15 levels), Java to JVM (5 horizontal, 4 vertical levels), C# to CLR
- reuse justifications (proofs) for system properties, e.g.
  - reusing Prolog-to-WAM compiler correctness proof for IBM's CLP(R)-to-CLAM, Protos-L-to-PAM
  - verification for software product lines (Batory/Börger)
- capture orthogonalities by modular (maintainable) components
  - e.g. Java/JVM components (interpreters, compiler, verifier, ...)

ExI: Language-Oriented Horizontal Refinements of Java/JVM



$$\mathrm{Java}_{\mathrm{I}} \xrightarrow[compile_I]{} \mathrm{JVM}_{\mathrm{I}}$$ imperative

$$\mathrm{Java}_{\mathrm{C}} \xrightarrow[compile_C]{} \mathrm{JVM}_{\mathrm{C}}$$ static class features (procedures)

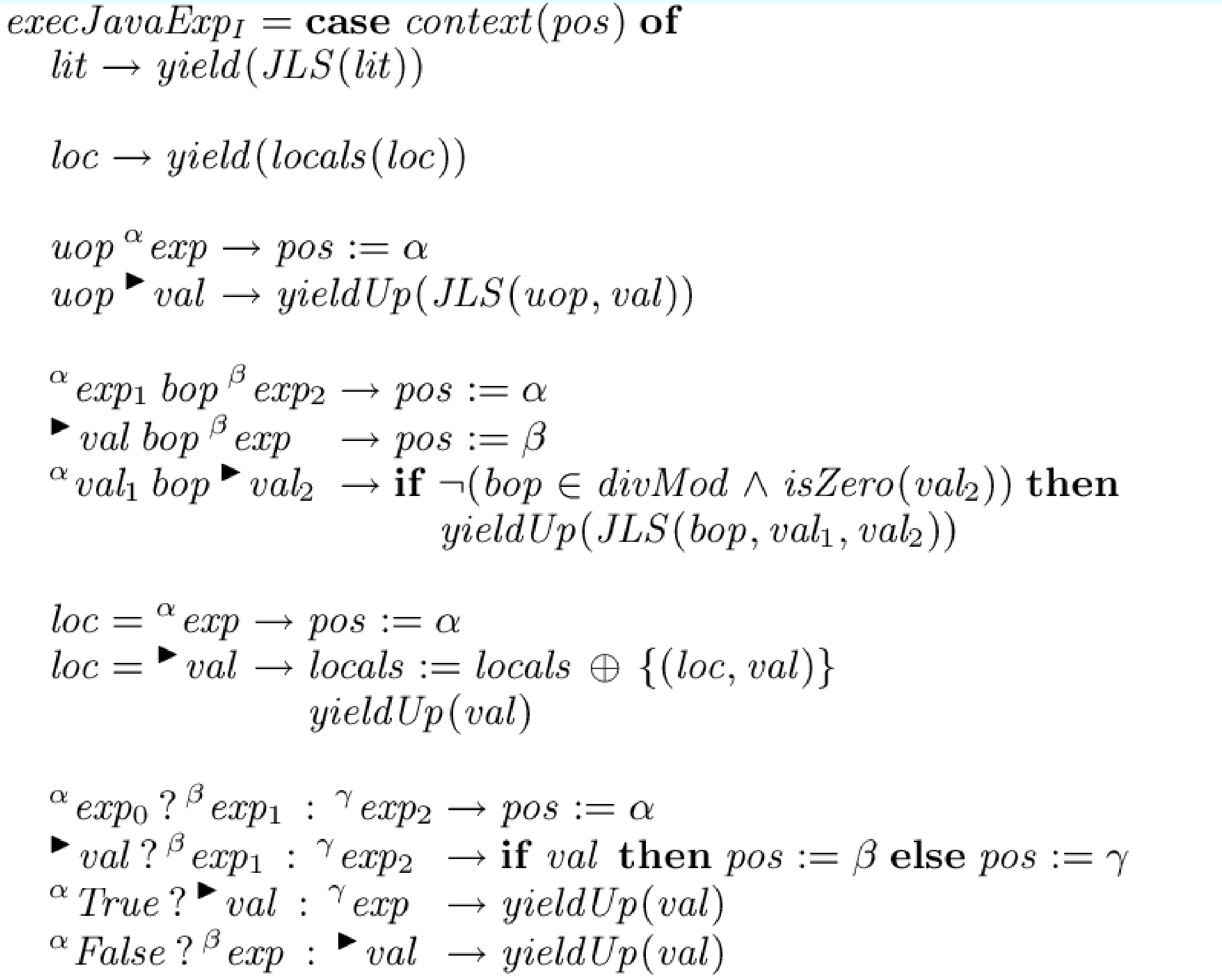
$$\mathrm{Java}_{\mathrm{O}} \xrightarrow[compile_O]{} \mathrm{JVM}_{\mathrm{O}}$$ oo features

$$\mathrm{Java}_{\mathrm{E}} \xrightarrow[compile_E]{} \mathrm{JVM}_{\mathrm{E}}$$ exception handling

$$\mathrm{Java}_{\mathrm{T}} \xrightarrow[compile_T]{} \mathrm{JVM}_{\mathrm{T}}$$ concurrent threads

Layers are conservative extensions of each other and thus support componentwise design and analysis (validation & verification). Combination with an appropriate parameterization provides an *orthogonal treatment of language constructs* ("instructionwise").

ExI: Java$_I$ Expression Evaluation Component



```
$execJavaExp_I =$ case $context(pos)$ of
    $lit \rightarrow yield(JLS(lit))$

    $loc \rightarrow yield(locals(loc))$

    $uop\, {}^{\alpha}exp \rightarrow pos := \alpha$
    $uop\, {}^{\blacktriangleright}val \rightarrow yieldUp(JLS(uop, val))$

    ${}^{\alpha}exp_1\ bop\ {}^{\beta}exp_2 \rightarrow pos := \alpha$
    ${}^{\blacktriangleright}val\ bop\ {}^{\beta}exp \rightarrow pos := \beta$
    ${}^{\alpha}val_1\ bop\ {}^{\blacktriangleright}val_2 \rightarrow$ if $\neg(bop \in divMod \wedge isZero(val_2))$ then
                                    $yieldUp(JLS(bop, val_1, val_2))$

    $loc = {}^{\alpha}exp \rightarrow pos := \alpha$
    $loc = {}^{\blacktriangleright}val \rightarrow locals := locals \oplus \{(loc, val)\}$
                      $yieldUp(val)$

    ${}^{\alpha}exp_0\ ?\ {}^{\beta}exp_1 : {}^{\gamma}exp_2 \rightarrow pos := \alpha$
    ${}^{\blacktriangleright}val\ ?\ {}^{\beta}exp_1 : {}^{\gamma}exp_2 \rightarrow$ if $val$ then $pos := \beta$ else $pos := \gamma$
    ${}^{\alpha}True\ ?\ {}^{\blacktriangleright}val : {}^{\gamma}exp \rightarrow yieldUp(val)$
    ${}^{\alpha}False\ ?\ {}^{\beta}exp : {}^{\blacktriangleright}val \rightarrow yieldUp(val)$
```

NB. One rule group per grammar clause (feature-based approach)

ExI: Java$_I$ Statement Execution Component

```
$execJavaStm_I = \textbf{case}\ context(pos)\ \textbf{of}$
    $; \rightarrow yield(Norm)$
    $^{\alpha}exp; \rightarrow pos := \alpha$
    $^{\blacktriangleright}val; \rightarrow yieldUp(Norm)$

    $\texttt{break}\ lab; \rightarrow yield(Break(lab))$
    $\texttt{continue}\ lab; \rightarrow yield(Continue(lab))$
    $lab : {}^{\alpha}stm \rightarrow pos := \alpha$
    $lab : {}^{\blacktriangleright}Norm \rightarrow yieldUp(Norm)$
    $lab : {}^{\blacktriangleright}Break(lab_b) \rightarrow \textbf{if}\ lab = lab_b\ \textbf{then}\ yieldUp(Norm)$
                                $\textbf{else}\ yieldUp(Break(lab_b))$
    $lab : {}^{\blacktriangleright}Continue(lab_c) \rightarrow \textbf{if}\ lab = lab_c\ \textbf{then}\ yield(body/pos)$
                                $\textbf{else}\ yieldUp(Continue(lab_c))$
    $phrase({}^{\blacktriangleright}abr) \rightarrow \textbf{if}\ pos \neq firstPos \wedge propagatesAbr(restbody/up(pos))\ \textbf{then}$
                          $yieldUp(abr)$

    $\{\} \rightarrow yield(Norm)$
    $\{^{\alpha_1}stm_1 \ldots {}^{\alpha_n}stm_n\} \rightarrow pos := \alpha_1$
    $\{^{\alpha_1}Norm \ldots {}^{\blacktriangleright}Norm\} \rightarrow yieldUp(Norm)$
    $\{^{\alpha_1}Norm \ldots {}^{\blacktriangleright}Norm^{\alpha_{i+1}}stm_{i+1} \ldots {}^{\alpha_n}stm_n\} \rightarrow pos := \alpha_{i+1}$

    $\texttt{if}\ ({}^{\alpha}exp)\ {}^{\beta}stm_1\ \texttt{else}\ {}^{\gamma}stm_2 \rightarrow pos := \alpha$
    $\texttt{if}\ ({}^{\blacktriangleright}val)\ {}^{\beta}stm_1\ \texttt{else}\ {}^{\gamma}stm_2 \rightarrow \textbf{if}\ val\ \textbf{then}\ pos := \beta\ \textbf{else}\ pos := \gamma$
    $\texttt{if}\ ({}^{\alpha}True)\ {}^{\blacktriangleright}Norm\ \texttt{else}\ {}^{\gamma}stm \rightarrow yieldUp(Norm)$
    $\texttt{if}\ ({}^{\alpha}False)\ {}^{\beta}stm\ \texttt{else}\ {}^{\blacktriangleright}Norm \rightarrow yieldUp(Norm)$

    $\texttt{while}\ ({}^{\alpha}exp)\ {}^{\beta}stm \rightarrow pos := \alpha$
    $\texttt{while}\ ({}^{\blacktriangleright}val)\ {}^{\beta}stm \rightarrow \textbf{if}\ val\ \textbf{then}\ pos := \beta\ \textbf{else}\ yieldUp(Norm)$
    $\texttt{while}\ ({}^{\alpha}True)\ {}^{\blacktriangleright}Norm \rightarrow yieldUp(body/up(pos))$

    $Type\ x; \rightarrow yield(Norm)$
```

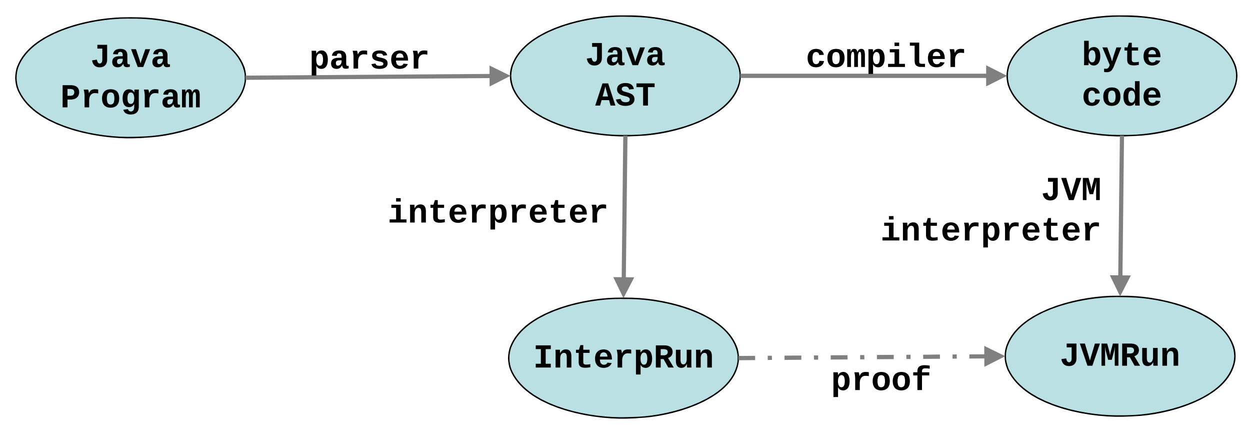
NB. Some rules trigger execution of exp evaluation rules

ExI: Vertical Refinements of Java/JVM Components

Components involved in compiler correctness verification (we omit standard grammar components):

$$JavaInterpreter, JvmInterpreter, JavaToJvmCompiler, Theorem$$

NB. $Theorem$ conveniently split into $Statement/Proof$ components



Similarly for other components (class loader, bytecode verifier, preparator) and their properties

Compatibility of horizontal with vertical Jbook refinements

Vertical Components are

- definable at each horizontal level (modular design principle)
- verifiable at each horizontal level (compositional proof technique)

Exl: *Tuple representation of components* for imperative expressions:

$$(Java_{Exp_I}, Jvm_{Exp_I}, JavaToJvm_{Exp_I}, Thm_{Exp_I})$$

Refinement of tuples, e.g. by components for imperative statements, *is componentwise composition* $\circ$ *of horizontal refinements*:

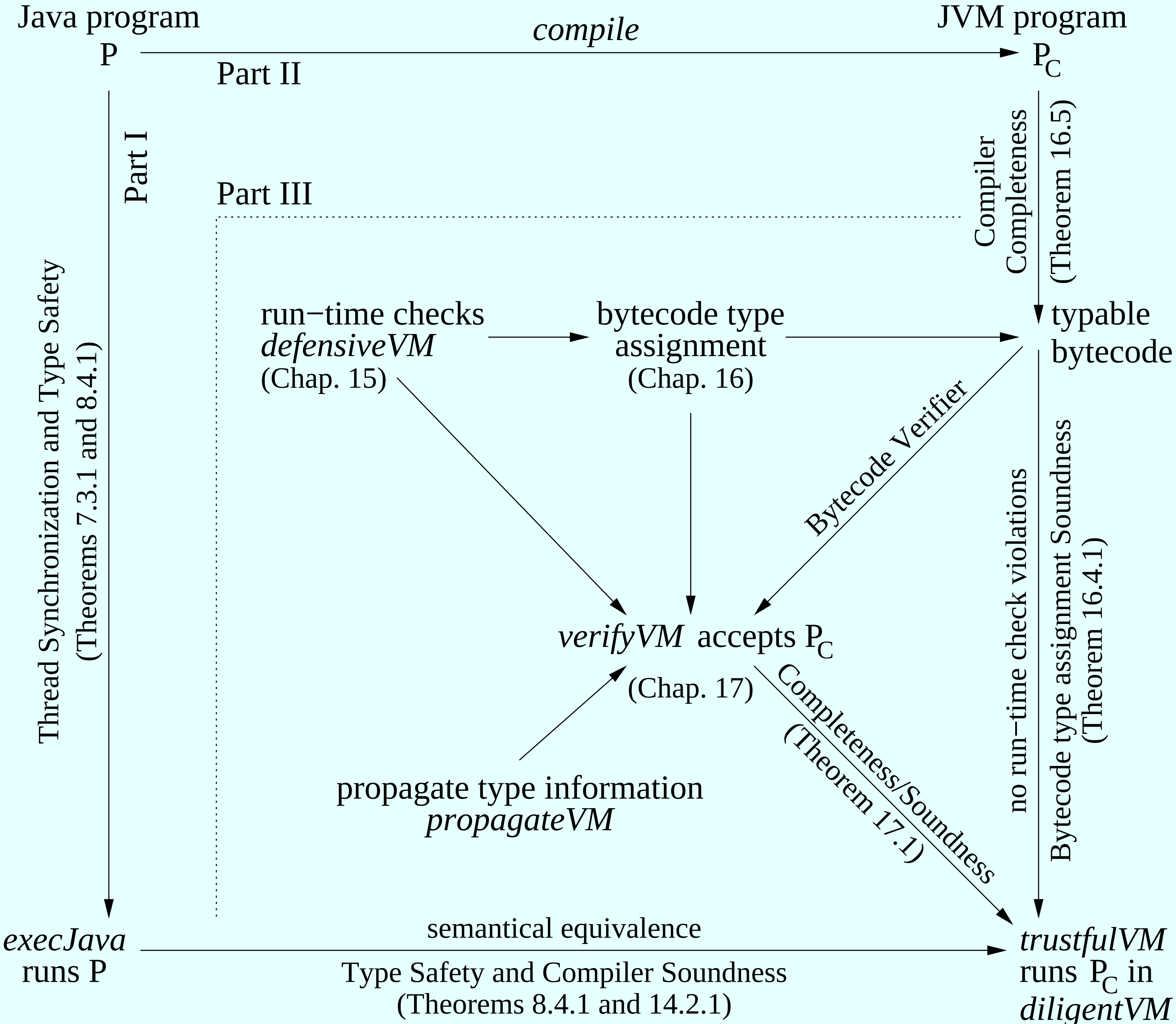
$$(Java_{Stm_I}, Jvm_{Stm_I}, JavaToJvm_{Stm_I}, ThmS_{Stm_I}, ThmP_{Stm_I})$$
$$\circ(Java_{Exp_I}, Jvm_{Exp_I}, JavaToJvm_{Exp_I}, ThmS_{Exp_I}, ThmP_{Exp_I})$$
$$=$$
$$(Java_{Stm_I} \circ Java_{Exp_I}, Jvm_{Stm_I} \circ Jvm_{Exp_I},$$
$$JavaToJvm_{Stm_I} \circ JavaToJvm_{Exp_I},$$
$$ThmS_{Stm_I} \circ ThmS_{Exp_I}, ThmP_{Stm_I} \circ ThmP_{Exp_I})$$

Integrating verification into feature-based development

- $Java_{Exp_I}$ has 6 interpreter rule groups, 1 per grammar clause
  - $Java_{Stm_I}$ *adds* nine *interpreter rule groups* for stm clauses
- $JavaToJvm_{Exp_I}$ has 6 recursive equations (plus 11 for non-strict (Boolean) exps exploited by the bytecode verifier)
  - $JavaToJvm_{Stm_I}$ *adds* eight *recursive equations* for stm clauses
- $ThmS_{Exp_I}$ has 5 invariants: about val equiv (of local variables/JVM registers) & equiv positions and computed intermediate vals at begin/end of exp eval (2 for strict, 2 for non-strict exps)
  - $ThmS_{Stm_I}$ *adds* 3 *invariants* about begin resp. (normal or abrupted) end of stm exec
- $ThmP_{Exp_I}$ verification has 13 (feature-determined) cases
  - $ThmP_{Stm_I}$ *adds verification of* 22 new *cases* concerning stm exec

NB. $ThmP_{Stm_I}$ uses $ThmP_{Exp_I}$ when invoking induction hypo for exps

NB. Some refinements add to resp. change given rules/invariants/proofs



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

Mechanical Verification Technology Transfer Challenge

Starting from the structured and high-level ASM definition of Java and of its implementation on the Java Virtual Machine

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

- upon compliant compilation
- passes the verifier (Compiler completeness)
- is executed on the JVM
  - without violating any run-time checks (Bytecode Verifier correctness)
  - correctly wrt Java source pgm semantics (Compiler correctness)

in a way that can be applied by language developers supporting stepwise model/theorem refinements, e.g. reuse for language extensions/variations

- NB. Fruja (2005-08) reused Java/JVM models and proofs for proving properties about .NET CLR exception handling and .NET CIL type safety (MSR Cambridge ROTOR project)

Modeling parallel systems programming

Synchronous parallelism is part of ASM semantics (**forall** construct)

- *APE architecture* reengineering project (Börger/DelCastillo 94-95):
  - programmer's view ground model ASM (with Rosenzweig/Glavan)
  - stepwise refinement (along APE100 compilation chain introducing pipelining and VLIW parallelism) to VLSI-implemented microprocessor zCPU
- *Verification of RISC pipelining* techniques:
  - Proven-to-be-correct stepwise refinement of sequential ground model to pipelined DLX architecture (Börger/Mazzanti 1996-97)
  - Applied to ARM2 microprocessor (Huggins/VanCampenhout 1998)
  - Extended in Teich's arch/compiler co-generation project (2000-01)
    - modeling application specific instruction set processors (read: register transfer descriptions) by ASM refinement hierarchies leading to XASM-executable (Anlauff 2000-01) models

Modeling lgs for programming distributed systems

Lipari Guide (1995) definition of distributed (asynchronous) ASMs replaced preceding ad hoc definitions to model concurrency with ASMs

- Variations of ASMs tailored for Occam (Gurevich/Moss 1990), Chemical Abstract Machine and π-calculus (Glavan/Rosenzweig 1993)

Two early examples of using Lipari Guide (asynchronous) ASMs:

- Ground model for *PVM at C-interface level* (Glässer/Börger 94-95)
  - PVM: env for programming heterogeneous distributed processes
- Ground model ASM interpreting *concurrent non-deterministic Occam* programs and its proven-to-be-correct stepwise refinement to a processor that runs high-/low-priority queues of Occam processes (Börger/Durdanovic/Rosenzweig 1994)
- Hierarchy of further *proven-to-be-correct refinement steps leading to Transputer code* (Börger/Durdanovic 1996)
  - following Inmos' Occam-to-Transputer compilation scheme

Exls of interpreter ASMs for domain specific languages

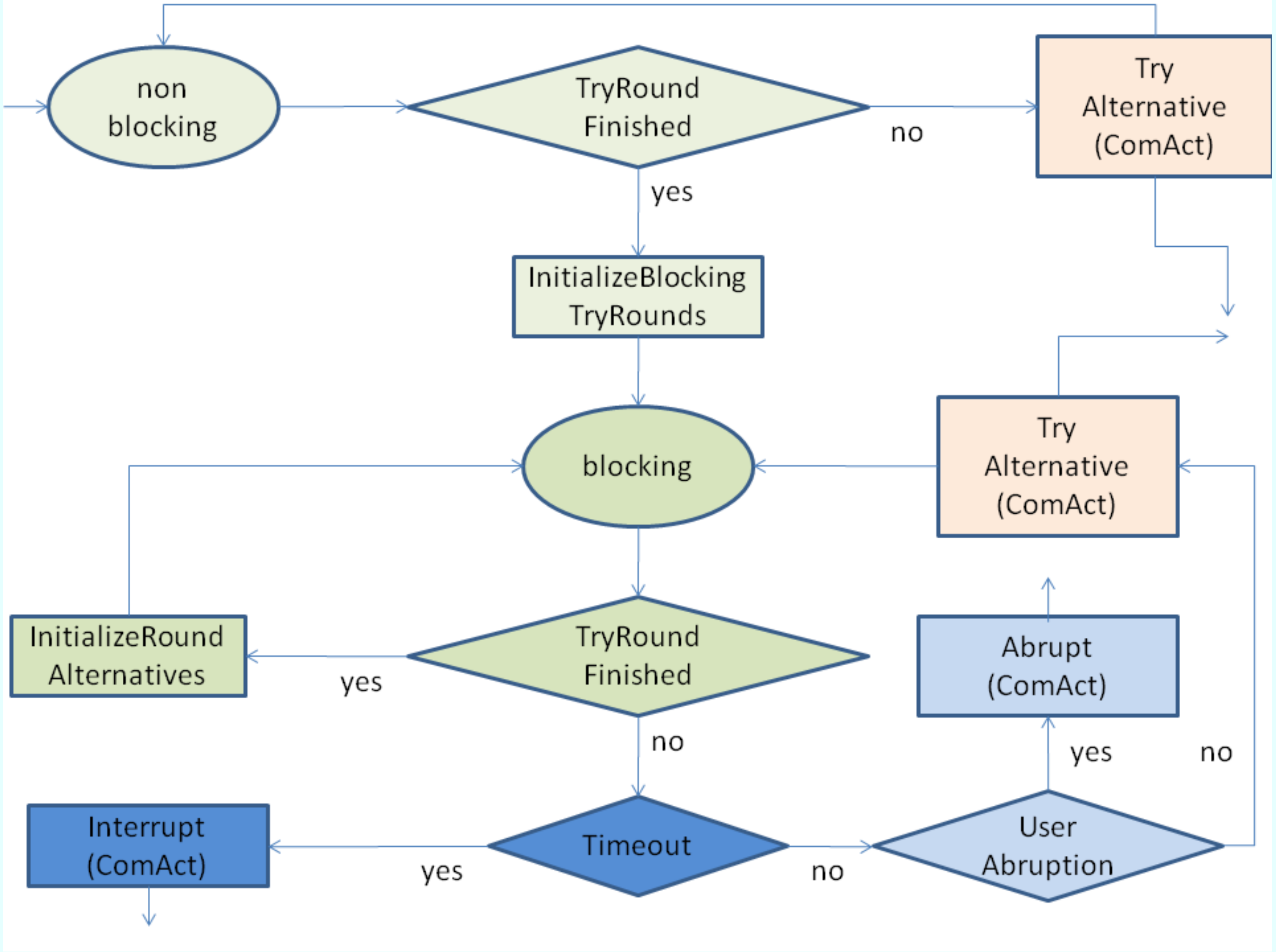
- HERA lg to program *schedulers for business processes*, obtained by a refinement of the Prolog ground model (Sauer 1993)
- lg to program *control for event-driven database applcs* (Behrend 1995)
- IEEE standard of *hardware design language VHDL93* (Börger/Glässer/W. Müller 1994-95). The model has been reused for
  - *pictorial extension* PHDL of VHDL'93 (W. Müller 1996)
  - extension to *analog VHDL and Verilog* (at Toshiba 1997-1999)
  - adaptation to *SystemC and SpecC* (W. Müller et al. 2001-03)
- *driver specification* lg at UBS (Kutter/Schweizer/Thiele 1998)
- ITU-T standard of *SDL2000* to design distributed real-time (in particular industrial telecommunication) systems (Glässer et al.)
  - ground model ASM refined to an AsmL-executable model (Prinz)

Exls of interpreter ASMs for BPM/web service languages

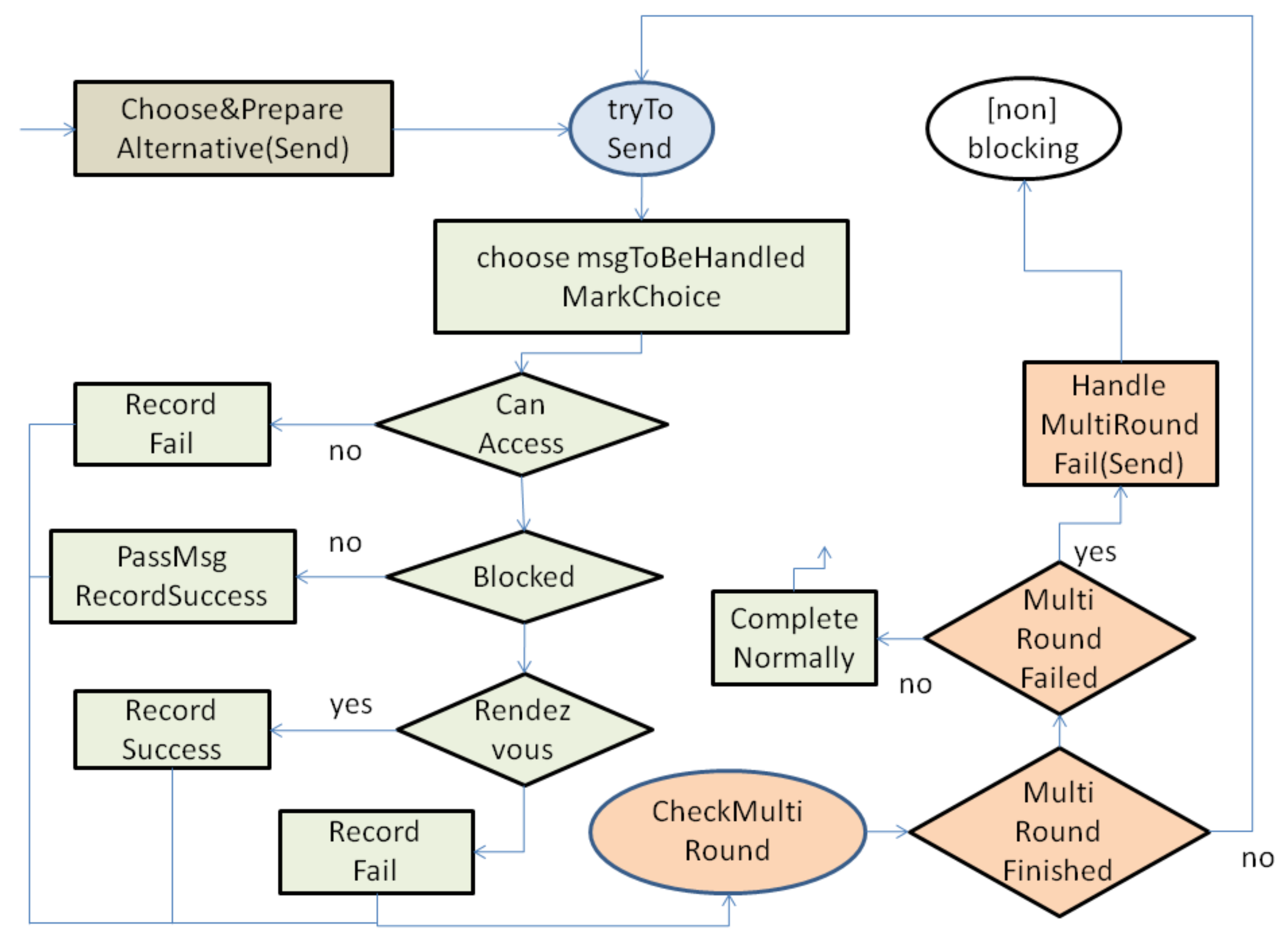
- *UML Activity Diagrams* version 1.3 (Börger/Cavarra/Riccobene 2000)
  - extension to ground model for richer version 2.0 (Sarstedt 2006)
    - implemented and integrated into a software development env where activity diagrams are executed (and visualized) directly (ibid.)
  - integration of *other behavioral UML 2.0 diagrams* by refining ASM models defined by different authors (Kohlmeyer/Guttmann 2009)
    - resulting in a rather practical, rigorous, ground model driven development approach for business process design
- (basic features of) OASIS executable lg *BPEL* to program BPs using web services as actions—also used as BPMN compilation target (Farahbod/Glässer/Vajihollahi 2004-06)
- graphical lg *BPMN 2.0*: proposing a rational reconstruction of OMG standard (Börger/Sörensen/Thalheim 2008-10)
- *S-BPM*: feature-based stepwise refined interpreter ASM (Börger 2011)
  - CoreAsm executable version under development at U Linz (Lerchner)

S-BPM communication component PERFORM($ComAct$)

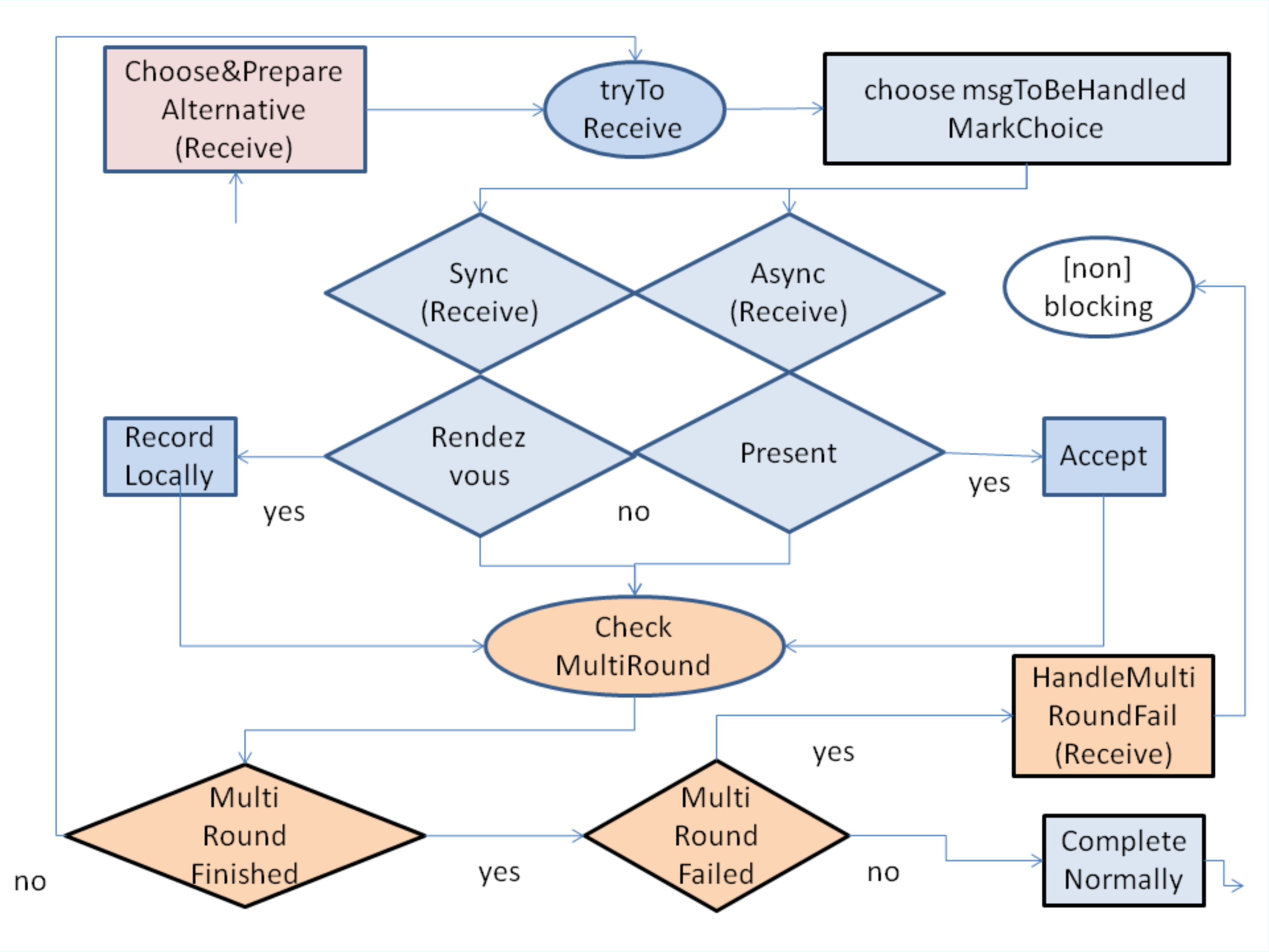
In S-BPM diagrams each node (before PROCEEDing) PERFORMs until completion either an InternalAction or an Alternative(Send/Receive)



S-BPM $\textsc{TryAlternative}(\mathit{Send})$ refinement

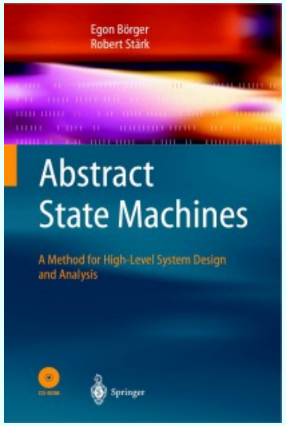


S-BPM TryAlternative(*Receive*) refinement

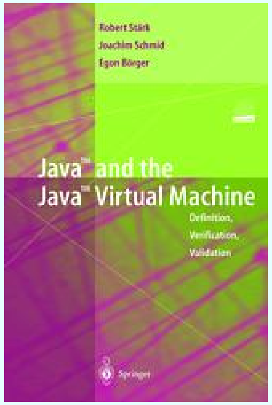


References

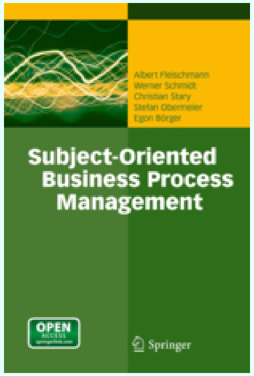
- E. Börger and R. F. Stärk: Abstract State Machines. Springer 2003
- R. Stärk, J. Schmid, E. Börger: Java and the Java Virtual Machine. *Definition, Verification, Validation*. Springer 2001
- A. Fleischmann et al.: Subject-Oriented Business Process Management. Open Access Book www.springer.com/978-3-642-32391-1



Egon Börger
Robert Stärk
Abstract State Machines
A Method for High-Level System Design and Analysis
Springer



Robert Stärk
Joachim Schmid
Egon Börger
Java™ and the Java™ Virtual Machine
Definition, Verification, Validation
Springer



Albert Fleischmann
Werner Schmidt
Christian Stary
Stefan Obermeier
Egon Börger
Subject-Oriented Business Process Management
OPEN ACCESS
Springer

- D. Batory and E. Börger: Modularizing Theorems for Software Product Lines: *The Jbook Case Study*. J.UCS 2008
- E. Börger: The Abstract State Machines Method for Modular Design and Analysis of Programming Languages (submitted)