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

Copyright © Egon Börger, Dipartimento di Informatica, Università di Pisa, Pisa, Italy.
Notion of ASM: generalizing Finite State Machines

\[
FSM = \begin{cases}
\text{if } \text{Defined}(\text{in}) \text{ then} & // \text{ do in parallel!} \\
\text{ctl}_\text{state} := \delta(\text{ctl}_\text{state}, \text{in}) & // \text{ static function } \delta \\
\text{out} := \lambda(\text{ctl}_\text{state}, \text{in}) & // \text{ static function } \lambda
\end{cases}
\]

FSMs come with five characteristic restrictions:

- **only 3 locations**, furthermore 0-ary (variables without parameters):
  - \text{in}: monitored (only read by FSM, but written by environment)
  - \text{ctl}_\text{state}: controlled (read and written by FSM)
  - \text{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) \(\delta, \lambda\)
Notion of ASM: extend FSM states to abstract states

ASM withdraws 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: \( \text{ASM} = \) finite set of rules

\[
\text{if } Cond \text{ then Updates}
\]

- \( \text{Updates} \) 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, \textit{abstractly and rigorously}
- at application-problem-determined (here: programming) level of detailing (\textit{minimality})
- formulated in application domain (here: language user) terms (\textit{precision}, informal accuracy)
- authoritatively for the further development activities: design contract/process/evaluation and maintenance (\textit{simplicity})

- ground the design in reality by justifying the definition as
  - \textit{correct}: model elems reliably convey original intentions (the manual)
  - \textit{complete}: every semantically relevant feature is present, no gap in understanding of ‘how to use’ resp. ‘what to build’
  - \textit{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
  - 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
    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)
Exl: Mixing execution engines for model validation

Java–ASM

Compiler–ASM

.j

Jasmin

BCEL

.class

Sun–JVM

Sun–Compiler

.j.java

JVM–ASM

Copyright © Egon Börger, Dipartimento di Informatica, Università di Pisa, Pisa, Italy.
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** \(\tau_i/\sigma_j\) **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**
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, … )
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").
execJavaExp₁ = case context(pos) of
    lit → yield(JLS(lit))
    loc → yield(locals(loc))

    uop α exp → pos := α
    uop ▶ val → yieldUp(JLS(uop, val))

    α exp₁ bop β exp₂ → pos := α
    ▶ val bop β exp → pos := β
    α val₁ bop ▶ val₂ → if -(bop ∈ divMod ∧ isZero(val₂)) then
                        yieldUp(JLS(bop, val₁, val₂))

loc = α exp → pos := α
loc = ▶ val → locals := locals ⊕ {(loc, val)}
yieldUp(val)

α exp₀ ? β exp₁ : γ exp₂ → pos := α
▶ val ? β exp₁ : γ exp₂ → if val then pos := β else pos := γ
α True ? ▶ val : γ exp → yieldUp(val)
α False ? β exp : ▶ val → yieldUp(val)

NB. One rule group per grammar clause (feature-based approach)
Exl: Java_I Statement Execution Component

```
execJavaStm_1 = case context(pos) of
  ; → yield(Norm)
  α exp; → pos := α
  ▷ val; → yieldUp(Norm)

  break lab;   → yield(Break(lab))
  continue lab; → yield(Continue(lab))
  lab : ▷ stmt → 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 yieldUp(body/pos)
                           else yieldUp(Continue(lab,c))

phrase(▷ abr) → if pos ̸= firstPos ∧ propagatesAbr(restbody/up(pos)) then
               yieldUp(abr)

{} → yield(Norm)
{α_1 stmt_1 ... α_n stmt_n} → pos := α_1
{α Norm ... ▷ Norm} → yieldUp(Norm)
{α Norm ... ▷ Norm α_i+1 stmt_i+1 ... α_n stmt_n} → pos := α_{i+1}

if (▷ exp) β stmt_1 else ▷ stmt_2 → pos := α
if (▷ val) β stmt_3 else ▷ stmt_2 → if val then pos := β else pos := γ
if (▷ True) ▷ Norm else ▷ stmt → yieldUp(Norm)
if (▷ False) β stmt else ▷ Norm → yieldUp(Norm)

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

Type x; → yield(Norm)
```

NB. Some rules trigger execution of exp evaluation rules

Copyright © Egon Börger, Dipartimento di Informatica, Università di Pisa, Pisa, Italy.
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:

\[
(\text{Java}_{\text{Exp}I}, \text{Jvm}_{\text{Exp}I}, \text{JavaToJvm}_{\text{Exp}I}, \text{Thm}_{\text{Exp}I})
\]

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

\[
(\text{Java}_{\text{Stm}I}, \text{Jvm}_{\text{Stm}I}, \text{JavaToJvm}_{\text{Stm}I}, \text{Thm}_{\text{Stm}I}, \text{ThmP}_{\text{Stm}I}) \\
\circ(\text{Java}_{\text{Exp}I}, \text{Jvm}_{\text{Exp}I}, \text{JavaToJvm}_{\text{Exp}I}, \text{Thm}_{\text{Exp}I}, \text{ThmP}_{\text{Exp}I}) \\
= \\
(\text{Java}_{\text{Stm}I} \circ \text{Java}_{\text{Exp}I}, \text{Jvm}_{\text{Stm}I} \circ \text{Jvm}_{\text{Exp}I}, \\
\text{JavaToJvm}_{\text{Stm}I} \circ \text{JavaToJvm}_{\text{Exp}I}, \text{Thm}_{\text{Stm}I} \circ \text{Thm}_{\text{Exp}I}, \text{ThmP}_{\text{Stm}I} \circ \text{ThmP}_{\text{Exp}I})
\]
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
- $JavaToFvm_{Exp_I}$ has 6 recursive equations (plus 11 for non-strict (Boolean) exps exploited by the bytecode verifier)
  - $JavaToFvm_{Stm_I}$ adds nine 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 abruptly ended) 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 \( P \) \( \xrightarrow{\text{compile}} \) JVM program \( P_C \)

Part II
(Theorems 7.3.1 and 8.4.1)
Thread Synchronization and Type Safety

Type Safety and Compiler Soundness
(Theorems 8.4.1 and 14.2.1)

\( \text{execJava} \) runs \( P \)

semantical equivalence

\( \text{execJava} \) runs \( P \)

\( \text{propagateVM} \) propagates type information

\( \text{defensiveVM} \) (Chap. 15)
run-time checks

\( \text{bytecode type assignment} \) (Chap. 16)

\( \text{verifyVM} \) accepts \( P_C \)
(Chap. 17)

(Chap. 16)

Bytecode Verifier

Compiler Completeness/Soundness
(Chap. 17)

Bytecode type assignment Soundness
(Chap. 16.4.1)

no run-time check violations

\( \text{trustfulVM} \) runs \( P_C \) in
\( \text{diligentVM} \)

Copyright © Egon Börger, Dipartimento di Informatica, Università di Pisa, Pisa, Italy.
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)
Synchronous parallelism is part of ASM semantics (\texttt{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 $\pi$-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 Ig to program *schedulers for business processes*, obtained by a refinement of the Prolog ground model (Sauer 1993)
- Ig 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* Ig 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)
In S-BPM diagrams each node (before \textbf{PROCEEDing}) \textbf{PERFORMs} until completion either an \texttt{InternalAction} or an \texttt{Alternative(Send/Receive)}.
S-BPM TryAlternative(Receive) refinement
References

- E. Börger: The Abstract State Machines Method for Modular Design and Analysis of Programming Languages (submitted)