Egon Börger (Pisa) & Alexander Raschke (Ulm)

The Abstract State Machines Method

for Modeling and Analysis of Software-Based Systems

An introductory survey of main concepts and characteristic results[1]

Università di Pisa, Dipartimento di Informatica, boerger@di.unipi.it
Università Ulm, Abteilung Informatik, alexander.raschke@uni-ulm.de

[1] Figures are © 2001,2003 Springer-Verlag Berlin Heidelberg, reused with permission

What the ASM method is about

PROBLEM: still frequently experienced *mismatch* between

- human understanding and formulation of real-world problems
  - by domain experts and system designers
- and deployment of problem solutions by code-executing machines
  - on changing platforms

To bridge the gap bw those two ends of system development

- the ASM method provides a practicable, mathematically well-founded systems engineering framework for
  - the construction of reliable computer-based systems
  - their reliable use
  - their cost-effective change management

  which are objectively and *effectively controllable* (certifiable)

Wide-spectrum method vs special-purpose technique

Consequently the ASM method

- is NOT a special-purpose technique
  - like static analysis, bytecode verification, model checking, theorem proving, run-time verification, etc., which draw their success from being tailored to particular types of problems at specific (usually technically detailed if not code) levels of abstraction
- in particular is NOT a ‘formal’ method
- but is a wide-spectrum method
  - assisting system engineers in every aspect and at any level of abstraction of an effectively controllable construction of reliable computer-based systems

Nevertheless the ASM method

- allows one to integrate special-purpose sw engg techniques
- can be tailored to application domain languages

The gap: how to match requirements and code?

- Requirement docus: *descriptions of real-world problems/activities*
  - written by domain experts *for system designers* who typically are not knowledgeable in the application domain
  - in natural lg, interspersed with diagrams, tables, formulae, etc.
  - frequently suffer from incompleteness (implicit assumptions), lack of precision (ambiguity) or inconsistency
- Compilable programs: *software representations of solutions*
  - written *for mechanical elaboration* by machines coming with technically detailed precision, completeness, consistency

——————————THE PROBLEM:——————————

- How can (informal) requs & (formal) code (written to satisfy the requs) be linked to certifiably guarantee that the code does what the requs describe and not something else?
- How can the link bw requirements and their implementation be reliably preserved during maintenance (*design for change*)?

What the ASM method offers to ‘bridge the gap’

- a precise general language with a validation/verification framework
  - practicing domain experts & system designers can use in daily work

  to formulate, justify, document *prior to implementing*

  accurate models of real-world problems

  to solve the ground model problem
- a rigorous general design and verification method
  - practicing system designers and implementers can incorporate into their development environment

  to successively/incrementally detail, controllably correct and traceably, hierarchies of model abstractions

  refining ground models to running system behavior models

  to solve the verified software problem

(called also certifiable implementation or refinement problem)

Characteristics of the ASM Method

Supports, within a single *precise yet simple conceptual framework*, and uniformly integrates the following activities/techniques:

- the major **software life cycle activities**, linking in a controllable way the two ends of the development of complex software systems:
  - **requirements capture** by constructing rigorous ground models
  - **architectural and component design** bridging the gap between specification and code by *piecemeal, systematically documented detailing* via stepwise refinement of models to code
  - **documentation** for *inspection*, *reuse*, *maintenance* (change management) providing, via intermediate models and their analysis, explicit descriptions of *software structure* and major *design decisions*
- the principal **modeling and analysis techniques**
  - dynamic (*operational*) and static (*declarative*) descriptions
  - validation (simulation) and verification (proof) methods *at any desired level of detail*

The three fundamental constituents of the ASM method

- 1. ASMs: an FSM-extension to let communicating agents compute concurrently over ‘most general’ states (Tarski structures)
- 2. ASM ground models: accurate description of requirements
  - at *application-domain-determined* abstraction level
  - expressed using rigorous natural lg ‘templates’ (ASMs)
  - providing *authoritative* reference for system lifecycle activities
  - *evaluatable* via analysis— testing/inspection(reasoning)/review process—to certify consistency, correctness, completeness properties

  supporting precise, documented & inspectable high-level design
- 3. ASM refinements: linking series of detailed design/coding decisions in an *organic & effectively maintainable chain of rigorous, coherent system models* leading to code
  - refinement links must guarantee that ground model system properties are preserved via series of design decisions —and document this for maintenance (reuse and change management)

How ASMs describe behavior (of ground/design models)

ASMs describe when and how to change a state yielding the next state

- state given *at whatever level of abstraction* (requs/design/implemtn)
  – sets of whatever objects with predicates/operations defined on them
- state change *by local actions* which update some state components directly, at that level of abstraction, without extraneous encoding

ASMs use only the *fundamental* **if** ... **then** –– *template* of commands (also of reasoning) in natural and scientific language (called *rules*):

if $GivenSituation$ **then** PERFORMACTION -- Draw Conclusion

- $GivenSituation$ describes any trigger/event/stateProperty that guards the execution of the action // resp. implies the $Conclusion$
- PERFORMACTION consists of finitely many data changes $f := exp$ which update (the value of) object f to (the value of) exp
  – Objects may be parameterized as $f(e)$ or $f(e_1, \ldots, e_n)$ with arbitray expressions e, e_i

ad 1. Notion of Finite State Machine

$FSM =$ -- this is an interpreter scheme

if $Defined(in)$ **then** -- *do in parallel!*

$ctl\_state := \delta(ctl\_state, in)$ -- static function δ

$out := \lambda(ctl\_state, in)$ -- static function λ

FSMs come with four characteristic restrictions:

- *only three locations*, all 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)
- *only three special data types*: finite sets of
  - input/output symbols (letters of an alphabet)
  - control states (labels/integers) representing bounded memory
- only two auxiliary (furthermore static) functions δ, λ
- strict separation of input (read) and output (write) locations

ASMs generalize FSM states, permitting:

- to read and update in each step simultaneously (synch. parallelism)
  - *arbitrarily many – possibly parameterized – locations*
    - memory locations (l,val(params)) of array variables l(params)
  - location *values of arbitrary type*
- to have *arbitrary* conditions as *rule guard* (not only input definedness)
- to have (not 2 but) *arbitrarily many simultaneously executed updates*
- to provide *env* with whatever needed auxiliary functions

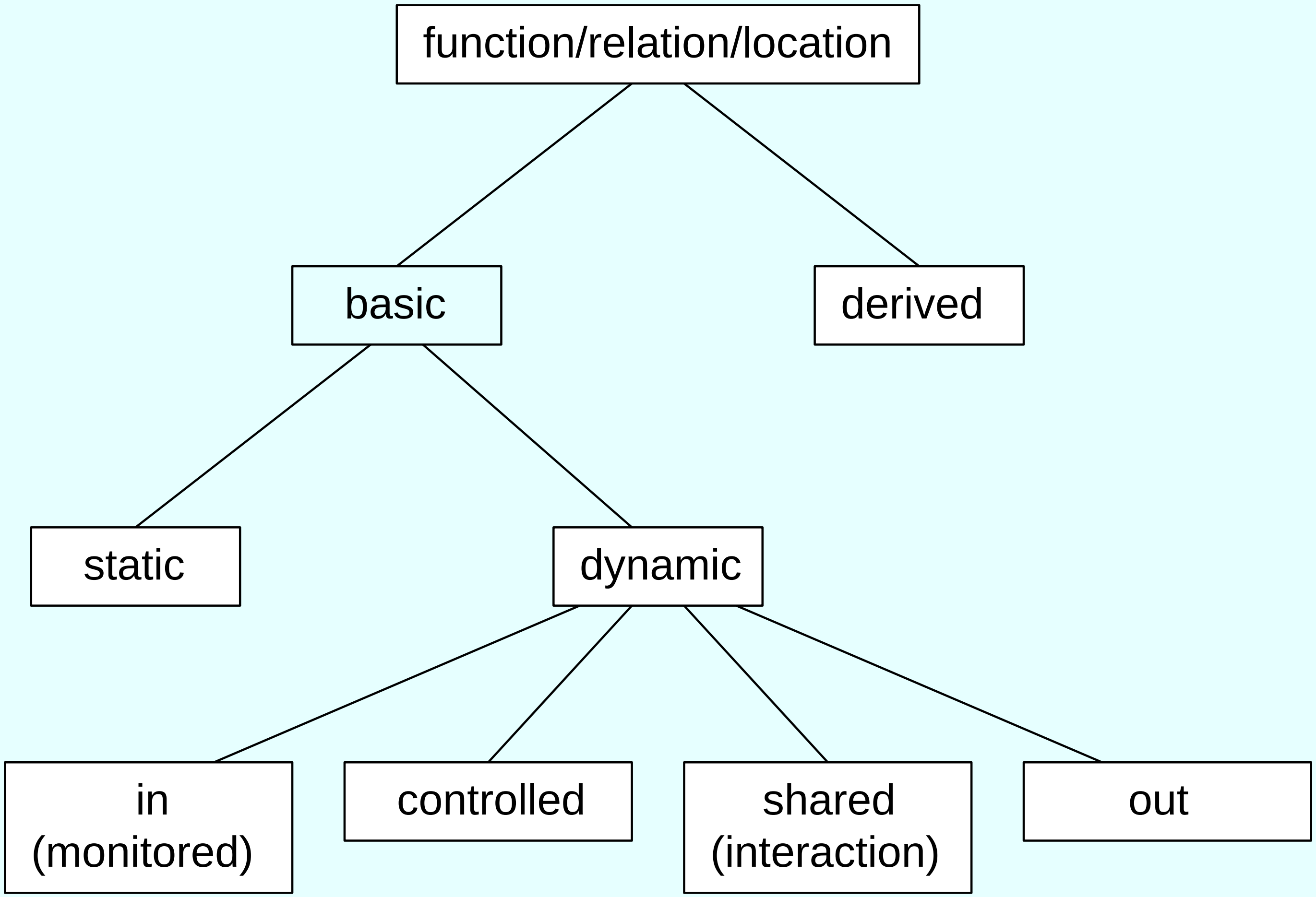
This leads to the definition: ASM = finite set of instructions ('rules')

if $Cond$ **then** $Updates$

- $Updates$: set of (simultaneous) assignments $f(t_1, \ldots, t_n) := t$
- $Cond, t_1, \ldots, t_n, t$: arbitrary exps ('formulae/terms')

Top-level, visualizable FSM $ctl\_state/phase/mode$ structure yields hierarchical system decomposition means into components

Generalized classification of locations/functions



supporting the separation of concerns: information hiding, data abstraction, modularization and stepwise refinement

Structuring abstract state memory into function tables

For each array variable l and each occuring length m of parameters of l:

group the subset of its *locations* $(l, (a_1, \ldots, a_m))$ of length m

- i.e. location name and values arg of parameter sequences of length n

This yields a *table* representation of those memory locations:

l	(l, arg_1)	$\ldots$	(l, arg_m)

Associate a value $l(arg)$ to each table entry (l, arg)

This yields a function table, i.e. a table which defines an n-ary function l, where $l(arg)$ represents for the given argument the uniquely determined value which is currently contained in location (l, arg):

l	$l(arg_1)$	$\ldots$	$l(arg_m)$

For simple variables (case $n = 0$) we write $\boxed{l}$

Example: Table representation of FSMs (states & pgm)

$$\textsc{Fsm}(in, out, \delta, \lambda) = \begin{cases} ctl\_state & := \delta(ctl\_state, in) \\ out & := \lambda(ctl\_state, in) \end{cases}$$

ctl_state
in
out

ctl_state/in	a_1	$\ldots$	a_m
1	$\delta(1, a_1)$	$\ldots$	$\delta(1, a_m)$
$\ldots$		$\ldots$	
n	$\delta(n, a_1)$	$\ldots$	$\delta(n, a_m)$

Similarly for the λ function in instructions (i, a, b, j).

This representations permits to *exploit sophisticated table manipulation and documentation* techniques (Parnas).

ASM states are Tarski structures

Via the structuring of ASM memory locations one can view an

- ASM state as a set of function tables
- ASM step as changing some values in some of these tables

ASM = FSM operating over function tables

In logic a function table for a function with name l is called an interpretation of function symbol l. Treating predicates by their characteristic functions yields:

Tarski structure = a set of tables
ASM = FSM operating over Tarski structures

NB. Structures of only functions are also called algebras.

The encompassing character of ASM states

ASM locations are not flat:

- their values can be structured complex objects of any type: records, documents, files, folders, images, sounds, movies, Web pages,...
- permit uniform combination of control, communication, data, resources

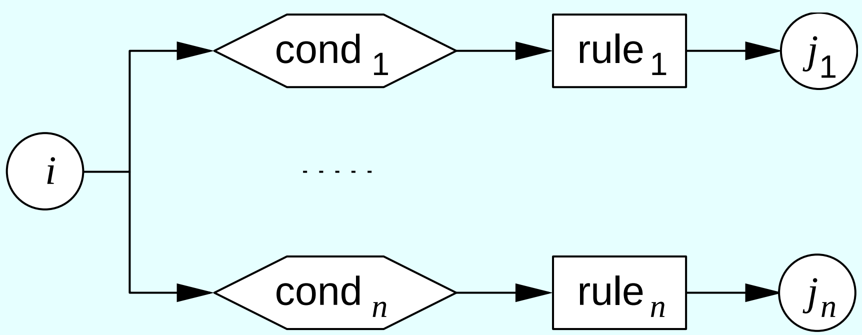
In fact Tarski structures represent a most general notion of structure

- the structures of mathematics are Tarski structures
- the models of abstract data types are Tarski structures
- classes (class instances) of oo pgg lgs are Tarski structures
- states of Virtual Machines are Tarski structures
- . . .

Important subclass of ASMs: Control State ASMs

Control State ASM = ASM all of whose rules have the form

$$\textbf{if } ctl\_state = i \textbf{ and } cond \textbf{ then } \begin{array}{l} rule \\ ctl\_state := j \end{array}$$



control-states $i, j, \ldots$ represent an overall system status (mode, phase), which allows the designer to

- *structure* the set of *states* into subsets
  - *visualize* this overall structure
- *refine* control-state transitions by control-state submachines (modules)
  - sequentializing (overall parallel) control where needed

Control State ASMs rigorously capture UML activity dgms

- UML event driven activity diagram scheme:
  - *If a certain event (situation) takes place, perform an action and proceed*
- control-state ASMs provide a general, mathematically rigorous, abstract meaning of:
  - *situation*: configuration of whatever items/data (abstract state) rigorously expressed by rule guarding $cond$itions
  - *action*: change of the configuration of some items (state transition/update) rigorously expressed by ASM transition $rule$s
  - *proceed*: rigorously expressed by ctl_state update
- NB. Each (synchronous) UML activity diagram can be built from alternating branching and action nodes of the control-state ASM diagram form (for each of the synchronized agents)

See the ASM-based framework built at U of Ulm for rigorous UML diagrams (Saarstedt, Guttmann, Raschke et al.)

Notation for non-determinism and parallelism

- *selection functions* (describing non-determinism) supported by dedicated notation for $rule(select\ \{x : \varphi(x)\})$:

  choose x **with** φ **in** $rule$

  to execute $rule$ for one element x, which is arbitrarily chosen among those satisfying the selection criterion φ
- symmetric notation to enhance synchronous parallelism:

  forall x **with** φ **do** $rule$

  to execute $rule$ simultaneously for every element x satisfying φ

Allow for standard notations, e.g. **let** $x = t$ **in** M.

The parallel ASM execution model

- easens specification of macro steps (by modularization and refinement)
- avoids unnecessary sequentialization of independent actions
- easens parallel/distributed implementations

Role of abstraction, parallelism, operational character

- abstraction enables to
  - represent *whatever objects* of discourse DIRECTLY, as is
    - focussing on their application-specific properties and operations
    - controlling encoding of data structures by dedicated refinements
  - *compose systems out of components* with precise interfaces
  - *reuse models* to capture requirements changes
- parallelism makes independence of actions explicit
  - abstracting from behaviorally irrelevant sequentialization
  - easens specification of macro steps (modularization/refinement)
  - supports distributed implementations & performance optimization by multi-threading
- operational character provides executability of models
  - both conceptual (for analysis) and mechanical (for experiments, testing and monitoring system behavior)

From seq in/out-focus to communication & concurrency

Replace sequential runs

- of stepwise sync in/output interaction of a single FSM with its env

by concurrent runs of multiple $a \in Agent$ with pgm ASM_a

- components are *event-triggered* & *discrete* (env can be continuous)
- $Agent$ and program assignment ASM_a are *dynamic*
  - e.g. add/delete agents and/or modify their programs to model dynamic networks with changing nodes or node functionality
- components interact asynchronously using communication
  - at a priori unpredictable moments (there is no global clock)
  - for a priori unpredictable reasons (interrupts, service requests, etc.)
  - state comprises in/out-mailbox actions $\textsc{Send}, \textsc{Receive}$
  - actions of *communication medium* separated from internal actions

  or synchronously via shared functions. See Boerger/Schewe in `https://link.springer.com/journal/236/53/5`

ExI for dynamic agent creation/deletion in ASMs

When $Agent$ and program association to agents are dynamic, ASM rules may add/delete agents and/or modify their programs

- ExI from a Web Apps Infrastructure model:

$\textsc{StartBrowsingContext}(r) =$ -- r a service request ID

let $a, b =$ **new** $Agent$ **in**

$program(a) := \textsc{Renderer}(r)$

-- yields user interface of DOM

$program(b) := \textsc{EventLoop}(r)$ -- executes browser events

$DOM(r) := initialDOM$

$agents(r) := \{a, b\}$

NB. agents a, b are deleted when the browsing context is stopped

ad 2. What are ground models?

Accurate blueprints of the to-be-implemented piece of real world —called 'golden models' in the semiconductor industry—which

- define 'the conceptual construct/the essence' of the software system (Brooks) prior to coding, *abstractly and rigorously*
  - at an application-problem-determined level of detailing (*minimality*)
  - formulated in application domain 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 elements reliably convey original intentions
  - *complete*: every semantically relevant feature is present (env,arch, domain knowledge), no gap in understanding of 'what to build'
  - *consistent*: conflicting objectives in requirements resolved

NB. Poor requirements are number one cause of project failures!

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
    - general data and operation framework and general interface concept (to represent system environments)
- Evidence problem: no infinite regress
  - 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)

Variety of real-life ASM ground models (1)

- **industrial standards**: ground models for the standards of
  - AODV routing protocol (2018)
  - OMG for BPMN 2.0: Kossak et al.(Springer book 2014)
  - OASIS for BPEL: Farahbod et al. ASM'04 and IJBPMI 1 (2006)
  - ECMA for C#: Börger, Fruja, Gervasi, Stärk: TCS 336 (2006)
  - ITU-T for SDL-2000: Glässer, Prinz et al. 1998–2003
  - IEEE for VHDL93: Müller, Glässer, Börger:1994–1995
  - ISO for Prolog: Börger, Rosenzweig: 1991–1995
- **design, reengineering, testing of industrial systems**:
  - railway & mobile telephony network component sw (at Siemens)
  - fire detection system sw (in German coal mines)
  - implementation of behavioral interface specifications on the .NET platform and conformence test of COM components (at Microsoft)
  - business systems interacting with intelligent devices (at SAP)

Variety of ASM ground models and their refinements (2)

- **programming lgs**: semantics/implementation of major pgg lgs, e.g.
  - Prolog (Quintus), SystemC, Java/JVM including bytecode verifier (SUN), C# (Microsoft)
  - domain-specific languages (UBS Switzerland)

  with KIV/PVS verification of compilers/compiler back-ends (DFG)
- architectural design: verification of pipelining schemes & VHDL-based hw design (Siemens), architecture/compiler co-exploration
- protocols: for authentication, cryptography, cache-coherence, routing-layers for mobile ad hoc networks, group-membership, etc.
- modeling e-commerce, workflows, business processes, web services, web apps infrastructure (at SAP and Metasonic)
- memory systems (Java, Cassandra)

6 fundamental questions for building ground models

The ASM method suggests to ask the following 6 questions when building a ground model as ‘models of the system’s intended behaviour’

- called *golden model* in the International Technology Roadmap for Semiconductors (2005)

1. Who are the system agents and what are their relations? What is the relation between the *system* and its *environment*?

2. What are the system states?

- What are the domains of objects and what are the functions, predicates and relations defined on them? (object-oriented approach to system design)
- What are the *static* and the *dynamic* parts (including input/output) of states?

6 fundamental questions for building ground models (Cont'd)

3. How and by which transitions do system states evolve?

- Under which conditions (*guards*) do the state transitions (*actions*) of single agents happen and what is their effect on the state?
- What is supposed to happen if those conditions are not satisfied? Which forms of *erroneous use* are to be foreseen and which *exception handling* mechanisms should be installed to catch them? What are the desired *robustness* features?
- How are the transitions of different agents related? How are the *internal* actions of agents related to *external* actions of the environment?

6 fundamental questions for building ground models (Cont'd)

4. What is the initialization of the system and who provides it? Are there termination conditions and, if yes, how are they determined? What is the relation between initialization/termination and input/output?

5. Is the system description complete and consistent?

6. What are the system assumptions and what are the desired system properties?

- At the level of transitions this question can be formulated and dealt with in terms of pre-/postconditions (assume/guarantee scheme)

ad 3. ASM Refinement for Management of Design Decisions

Refinement is a general methodological principle:

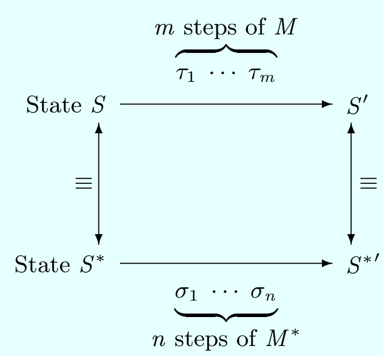
- piecemeal de-/composition of a system into/from constituent parts which are treated separately and combined to manage complexity
- goes together with the inverse process of abstraction

ASM refinements exploit the availability in ASMs of arbitrary structures to directly reflect states/operations to

- support divide-and-conquer techniques for system design *and* analysis
  – without privileging one to the detriment of the other
- allow the designer to tailor refinement/abstraction pairs which
  – faithfully *reflect a design decision* or reengineering idea
  – provide means to *justify an implementation* as 'correct'
    - linking—thru various levels of abstraction—the system architect's view (blueprint) to the developer's view (implementation)
  – support design *communication*, design *reuse* and system *maintenance* thru accurate, precise, indexed and searchable docu

ASM refinements offer freedom to choose notions of:

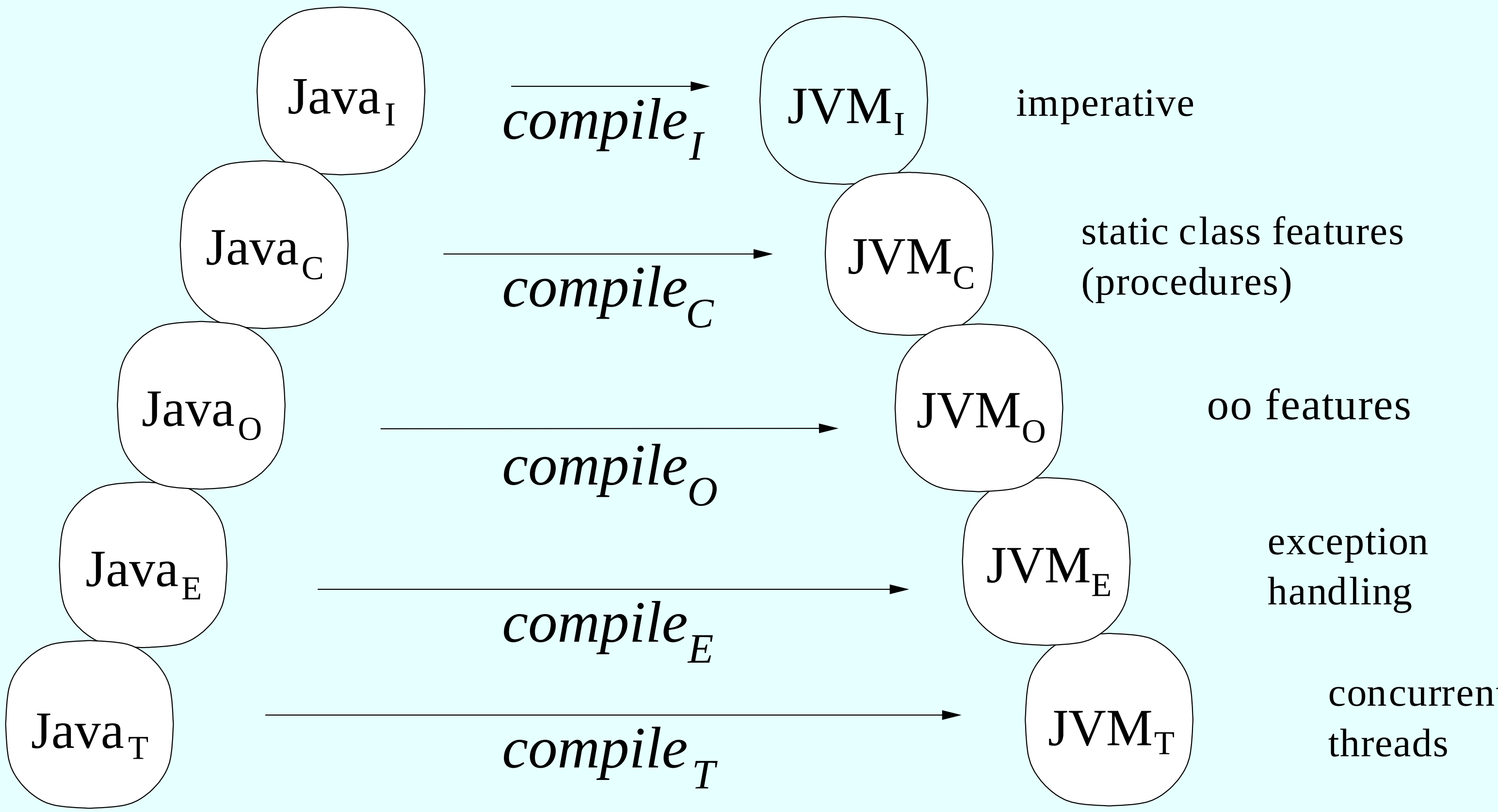
- *abstract/refined state*
- states of interest and correspondence bw pairs (S, S^*) of abstract/refined states of interest
- abstract/refined computation segments of m/n single abstract/refined steps τ_i/σ_j leading from/to corresponding states of interest
- *locations of interest* and *corresponding*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



Main usages of ASM refinements

- capture orthogonalities by modular (maintainable) components
  - separate orthogonal design decisions, relate different system aspects
- construct hierarchical levels (cost-effective system maintenance) 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 (Müller), parallelism (Parlog, Concurrent Prolog), 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/run experiments) for system properties
  - e.g. reusing Prolog-to-WAM compiler correctness proof for IBM's CLP(R)-to-CLAM, Protos-L-to-PAM, Java to C#, etc.

Java2JVM exl for horizontal/vertical ASM refinements



- horizontal (incremental) refinement of components (conservative extensions) supports componentwise design, validation, verification
- vertical refinement from spec to implementation: Java $\rightarrow_{compile}$ JVM via horizontally refined $compile$ with appropriate parameterization

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 correct compilation
- passes the verifier
- is executed on the JVM
  - without violating any run-time checks
  - correctly wrt Java source pgm semantics

in a way that can be applied by language developers, e.g. reused for language extensions: C#, ...

- integrating verification into feature-based devpmt of sw product lines

see Batory/Börger in J.UCS 14.12 (2008) http://www.jucs.org/jucs_14_12/modularizing_theorems_for_software

Machine verified ASMs suggest feasability

Examples:

- *Prolog2WAM* compiler correctness theorem (Börger/Rosenzweig 1995) verified in KIV (Schellhorn/Ahrendt J.UCS 3.4 (1997) http://www.jucs.org/jucs_3_4/reasoning_about_abstract_state)
- First full mechanical verification of *Mondex electronic purse* using KIV (Schellhorn et al. LNCS 4085 etc.)
  - extended to include treatment of security issues and coding of the protocol (in Java)
- AsmTP verification of *thread handling properties for C#* interpreter model (Stärk in TCS 343, 2005)

Simple refinement exl: refining FSM to 2-Way FSM

The $input$ location is extended to a tape where the input reader can

- Move its current head Position // by instructions (i, a, b, m, j)
- read the current input location in(head).

$$\textsc{TwoWayFsm}(in, out, \delta, \lambda, Move, head) =$$
$$\quad \textsc{Fsm}(in(head), out, \delta, \lambda) \text{ // parameterize locn } in \text{ by } head$$
$$\quad head := head + Move(ctl\_state, in(head)) \text{ //add new rule}$$

Correspondingly we extend the table representation of FSMs:

ctl_state
head
in(head)
out

ctl_state/in	a_1	...	a_m
1	$Move(1, a_1)$	...	$Move(1, a_m)$
...		...	
n	$Move(n, a_1)$	...	$Move(n, a_m)$

ExI: refining 2-Way FSM to (Interactive) Turing Machine

$\textsc{TwoWayFsm}(in, out, \delta, \lambda, Move, head) =$

$\quad \textsc{Fsm}(in(head), out, \delta, \lambda)$

$\quad head := head + Move(ctl\_state, in(head))$

Merge read and write locations on a unique tape: in=out

$\mathrm{TM}(tape, \delta, \lambda, Move, head) =$

$\quad \textsc{TwoWayFsm}(tape, tape(head), \delta, \lambda, Move, head)$

Dynamic state reduced to $\ldots\ i\ a \ldots$

tape	ctl_state	head

$\textsc{InteractiveTM}(input, \ldots) =$

-- data refine $\delta, \lambda Move$ by monitored param $input$

$\quad \mathrm{TM}(tape, \delta_{input}, \lambda_{input}, Move_{input}, head)$

$\quad \textsc{Output}(input, ctl\_state, tape(head))$ -- add external output rule

ASM characteristics: a coherent divide-and-conquer approach

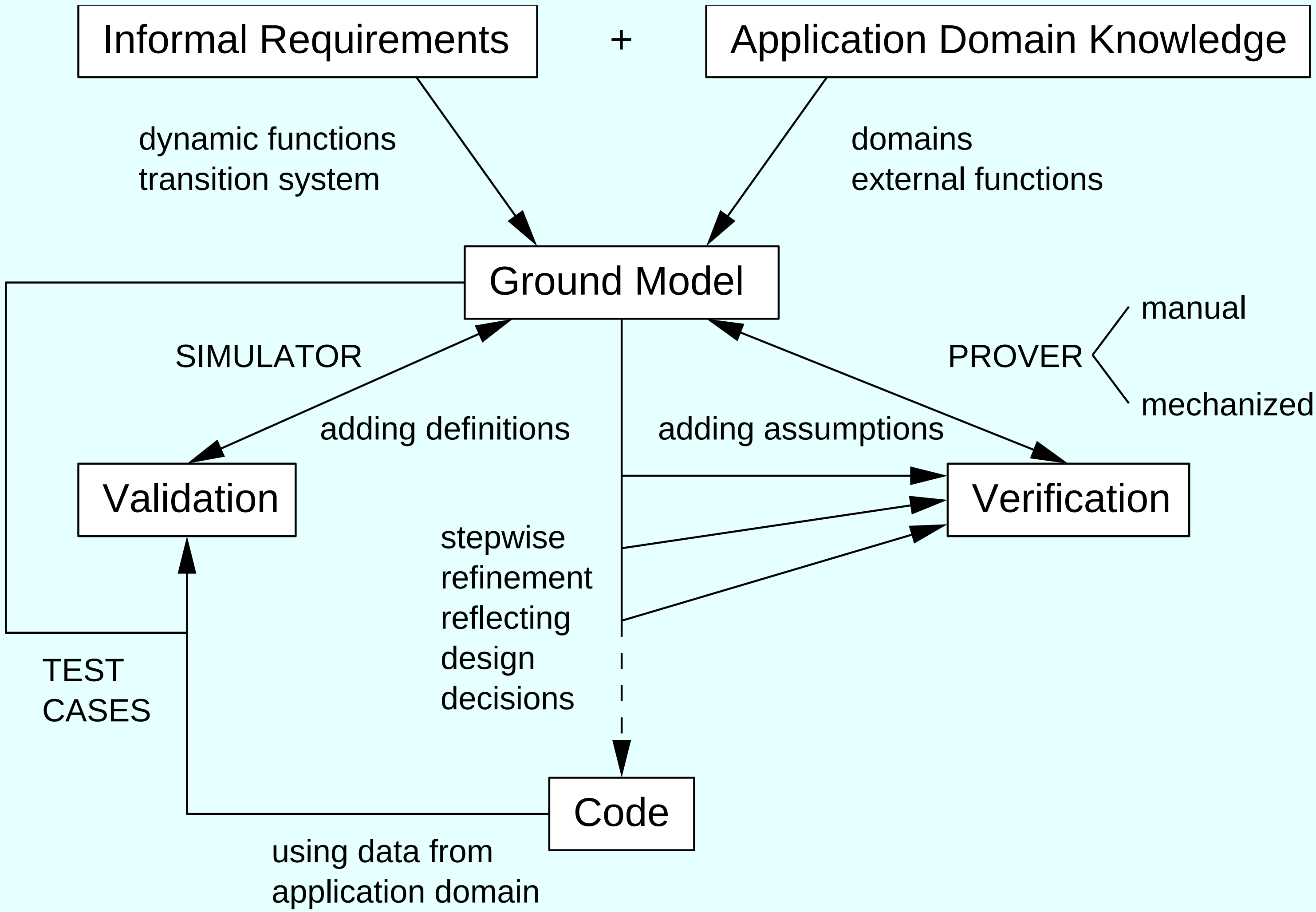
- capture orthogonalities by modular (maintainable) components
  - *separate multiple concerns, concepts and techniques*
  - choose for each task appropriate engineering methods
    - at the level of abstraction and precision where the task occurs
- construct hierarchical levels for
  - horizontal piecemeal extensions and adaptations (*design for change*)
  - vertical stepwise detailing of models (*design for reuse*) to their implementation in a provably correct way
- combine design and analysis in a consistent way, integrating
  - *mathematical verification* by a variety of reasoning techniques
  - *experimental validation* of system behaviour through simulation (model-checking, run-time verification, testing) of rigorous models
- reuse justifications (proofs) for system properties, e.g. Batory/Börger `http://www.jucs.org/jucs_14_12/modularizing_theorems_for_software`

ASM Analysis Techniques (Validation and Verification)

Practitioner supported to analyze ASM models by reasoning and experimentation at the appropriate degree of detail, separating

- orthogonal design decisions and complementary methods: abstract operational vs declarative/functional/axiomatic, state- vs event-based
- design from analysis (definition from proof)
- validation (by simulation) from verification (by reasoning)
  - e.g. ASM Workbench (ML-based, DelCastillo 2000), AsmGofer (Gofer-based, Schmid 1999), XASM (C-based, Anlauff 2001), AsmL (.NET-based, MSR 2001), CoreASM (since 2005) `https://github.com/CoreASM`, Asmeta (Milan/Bergamo)
- verification levels (degrees of detail)
  - reasoning for human inspection (design justification)
  - rule based reasoning systems (e.g. Stärk's Logic for ASMs)
  - interactive proof systems, e.g. KIV, PVS, Isabelle, AsmPTP
  - automatic tools: model checkers, automatic theorem provers

Models and methods in an ASM-based development process



References: 3 books

R. Stärk, J. Schmid, E. Börger: Java and the Java Virtual Machine. Definition, Verification, Validation. Springer 2001

http://www.di.unipi.it/ boerger

E. Börger and R. F. Stärk: Abstract State Machines. Springer 2003

http://www.di.unipi.it/ boerger

E. Börger and A. Raschke: Modeling Companion for Software Practitioners. Springer 2018

`http://modelingbook.informatik.uni-ulm.de`

References: Ground Model and Refinement Concepts

E. Börger: *The ASM* Refinement *Method*.

- Formal Aspects of Computing 15 (2003), 237-257

See also ASM refinement analysis papers by G. Schellhorn in J.UCS 2001, TCS 2005, J.UCS 2008, ENTCS 2008, SCP 2011

E. Börger: Construction and Analysis of Ground Models and their Refinements as a Foundation for Validating Computer Based Systems.

- Formal Aspects of Computing J. 19 (2007), 225-241

See also Sect.2 of E. Börger: Why Programming Must Be Supported by Modeling and How.

- Proc. ISoLA 2018, Springer LNCS 11244, pg.89–110

For the other sources mentioned in the slides one can find exact references in the three books quoted above.

Copyright Notice