Egon Börger (Pisa)

Università di Pisa, Dipartimento di Informatica, boerger@di.unipi.it

The ASM Method for System Design and Analysis.

A Tutorial Introduction

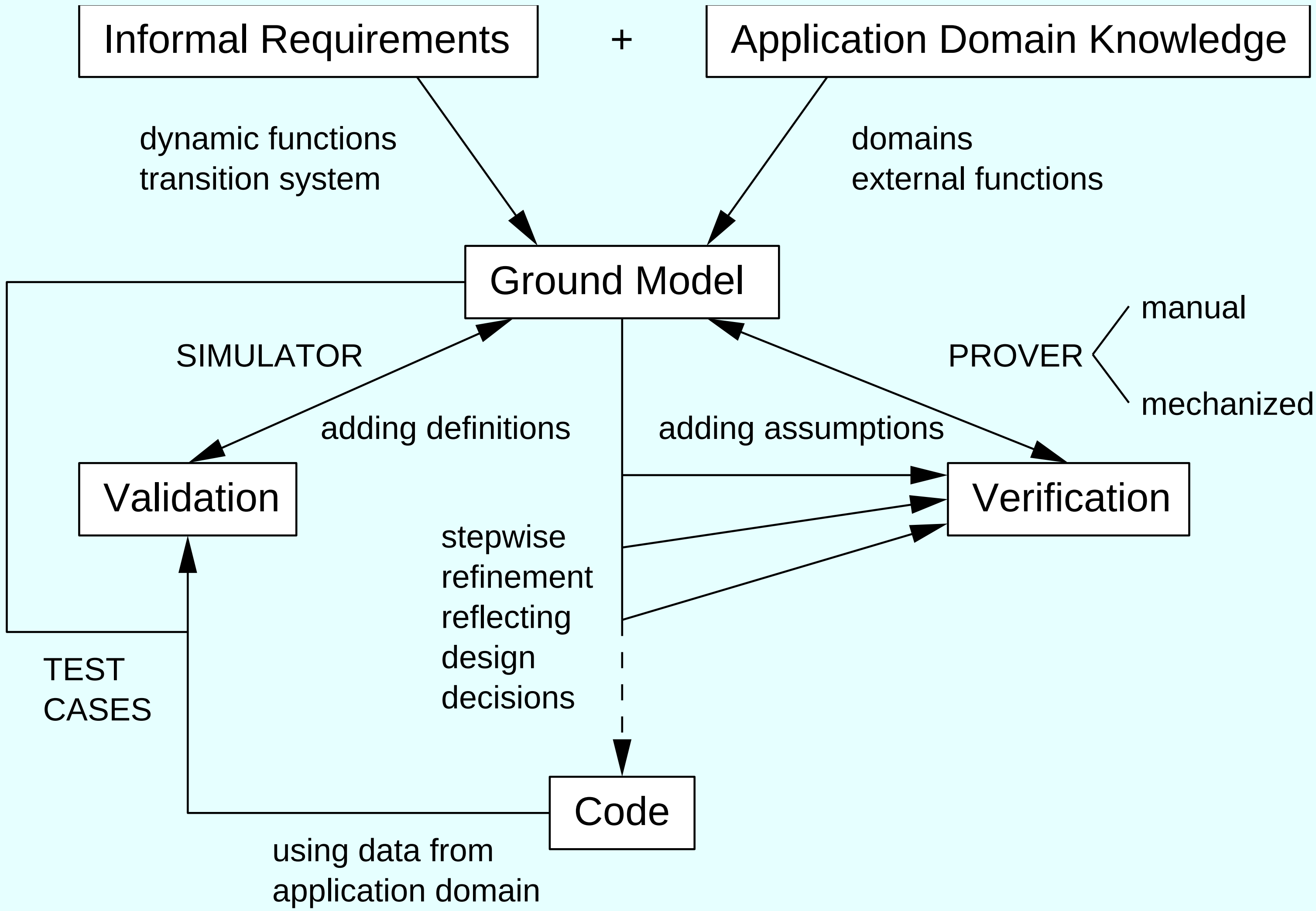
Paper printed in: B. Gramlich (Ed.): Frontiers of Combining Systems.
Springer LNAI 3717 (2005), 264-283

Scope and Achievements 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* of abstract models via stepwise refined models to code
  - **documentation** for *inspection*, *reuse*, *maintenance* 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*

Models and methods in the ASM-based development process



Variety of applications of ASMs (1)

- **industrial standards**: *ground models* for the standards of
  - OASIS for Business Process Execution Language for Web Services
  - ECMA for C#
  - ITU-T for SDL-2000
  - IEEE for VHDL93
  - ISO for Prolog
- **design, reengineering, testing of industrial systems**:
  - railway and mobile telephony network component software (at Siemens)
  - fire detection system 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)
  - compiler testing and test case generation tools

Variety of applications of ASMs (2)

- **programmming languages**: definition and analysis of the semantics and the implementation for the major real-life programmming languages, among many others for example
  - SystemC
  - Java/JVM (including bytecode verifier)
  - domain-specific languages used at the Union Bank of Switzerland

  including the verification of numerous compilation schemes and compiler back-ends
- architectural design: verification (e.g. of pipelining schemes or of VHDL-based hardware design at Siemens), architecture/compiler co-exploration
- protocols: for authentication, cryptography, cache-coherence, routing-layers for distributed mobile ad hoc networks, group-membership etc.
- modeling e-commerce and web services (at SAP)

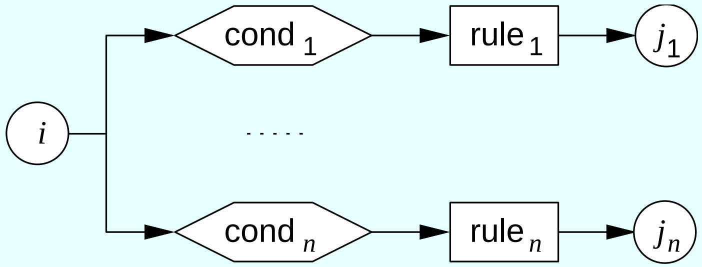
Three Ingredients of the ASM Method

ASM method comes with a rigorous scientific foundation:

- *ASM* = FSM with generalized state
- *ASM ground models*: mathematical blueprints (instead of loose human-centric UML descriptions)
- *ASM refinements* accurately link models at successive stages of system development cycle in an organic and effectively maintainable chain of coherent system views (fills gap in UML-based techniques)

The resulting documentation maps the structure of the blueprint to compilable code, providing a road map for system use and maintenance.

Turning FSMs into Abstract State Machines




```
if ctl_state = i then
   if cond_1 then
      rule_1
      ctl_state := j_1
         ...
   if cond_n then
      rule_n
      ctl_state := j_n
```

instructions $\textsc{Fsm}(i, \textbf{if } cond_\nu \textbf{ then } rule_\nu, j_\nu)$ updating

- a single internal ctl_state assuming values $i, j_1, \ldots, j_n$ in a not furthermore structured finite set
- in/output locations in, out assuming values in a finite alphabet

are extended by allowing

- a *set of parameterized locations holding values of whatever types*
- simultaneous updates of arbitrary many locations via *multiple assignments* $loc(x_1, \ldots, x_n) := val$

resulting in rules of form **if** $cond$ **then** $assignments$ with

- non-determinism replaced by synchronous parallelism

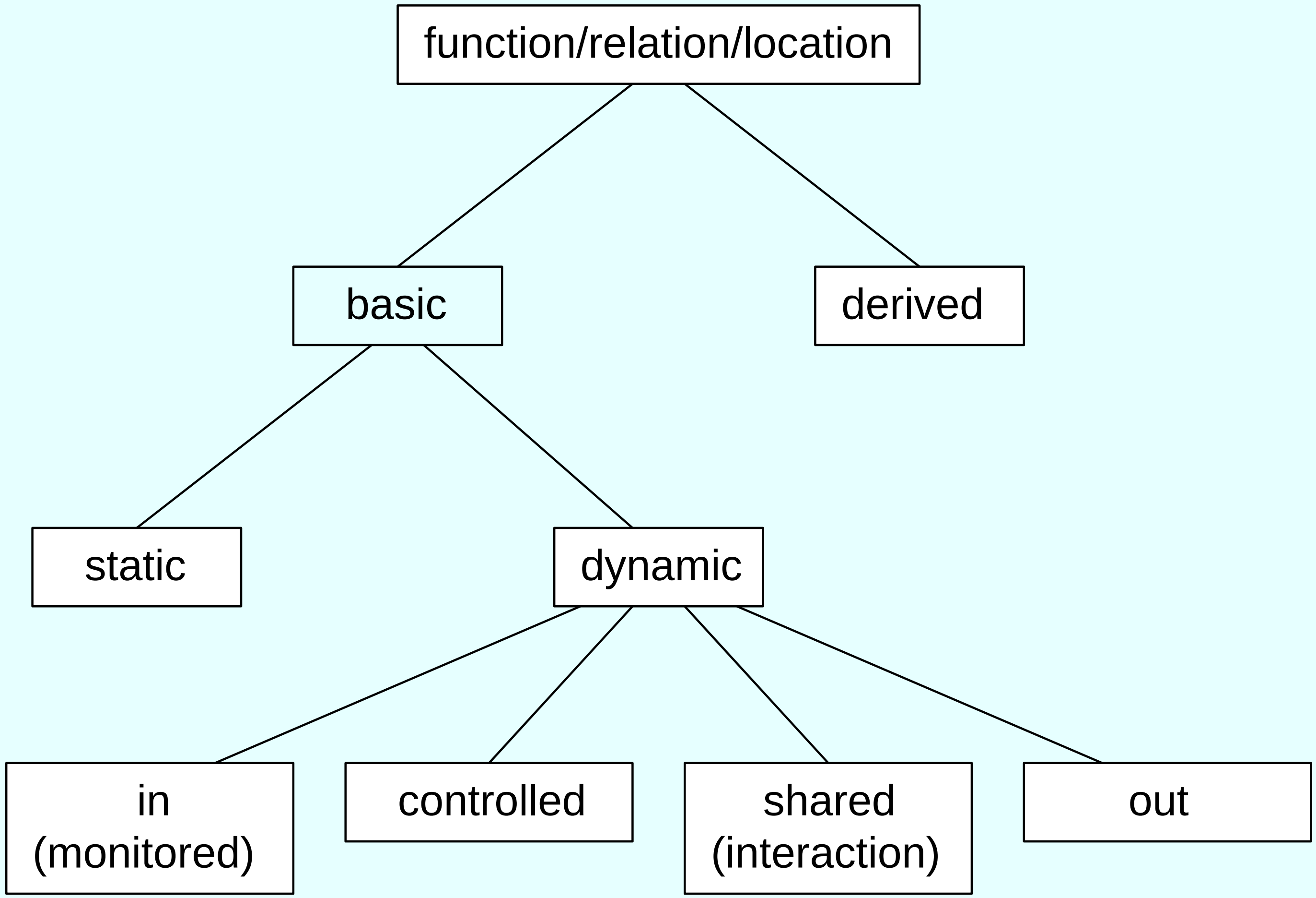
ASMs viewed as transforming Tarski structures

- group subsets of locations into tables (array variables f) of fixed dimension n
- associating to each table entry $(f, (a_1, \ldots, a_n))$ a value $f(a_1, \ldots, a_n)$ yields the current interpretation of the table f as an n-ary "dynamic" function or predicate (boolean-valued function)
- ASM state = set of tables = (multisorted) Tarski structure

Consequently the FSM-input $cond$ition $in = a$ is extended to arbitrary ASM-state expressions (first-order formulae), called *guards*.

Reassuming the ASM semantics: to execute one step of an ASM in a given state S, determine all the fireable rules in S (s.t. $cond$ is true in S), compute all expressions t_i, t in S occuring in the updates $f(t_1, \ldots, t_n) := t$ of those rules and then perform simultaneously all these location updates if they are consistent. In the case of inconsistency, the run is considered as interrupted.

Classification of ASM Functions and Locations



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

Notational Shorthand for Selection Functions

Nameless notation for *selection functions* f to select out of a collection X of objects satisfying a property φ one element $f(X)$ (in a way that may depend on the current state) to execute $rule(f(X))$:

choose x **with** φ

$\quad rule(x)$

A typical application: denoting abstract scheduling policies, e.g. for thread handling of Java

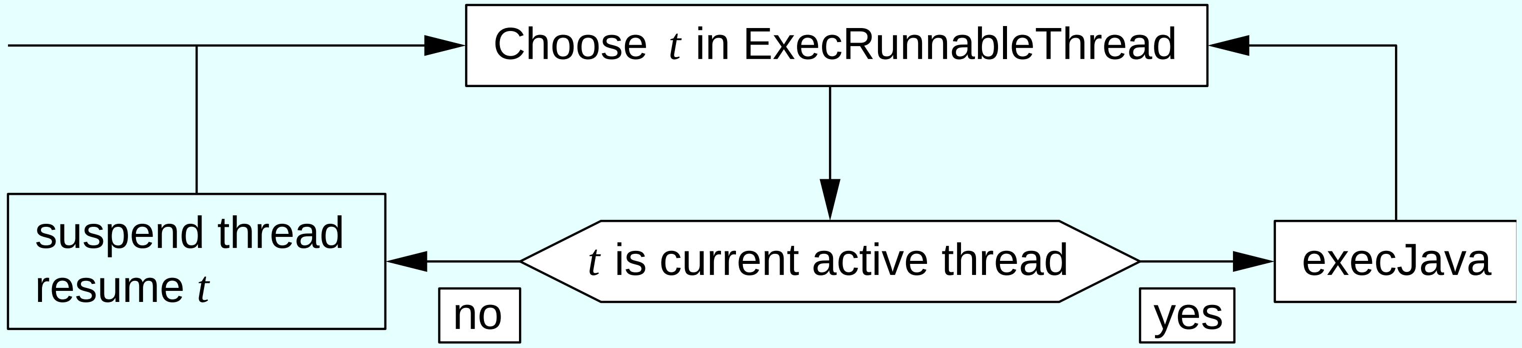


Fig. 0.1. Multiple thread Java machine

Further Standard Notational Shorthands

- expressing *synchronous parallelism* in terms of arbitrary properties:

  forall x **with** φ

  $\quad rule(x)$

  standing for the simultaneous execution of $rule(x)$ for every element x satisfying φ
- **if** $cond$ **then** R_1 **else** R_2
- **let** $x = t$ **in** R
- ...

For a further formalization see the AsmBook

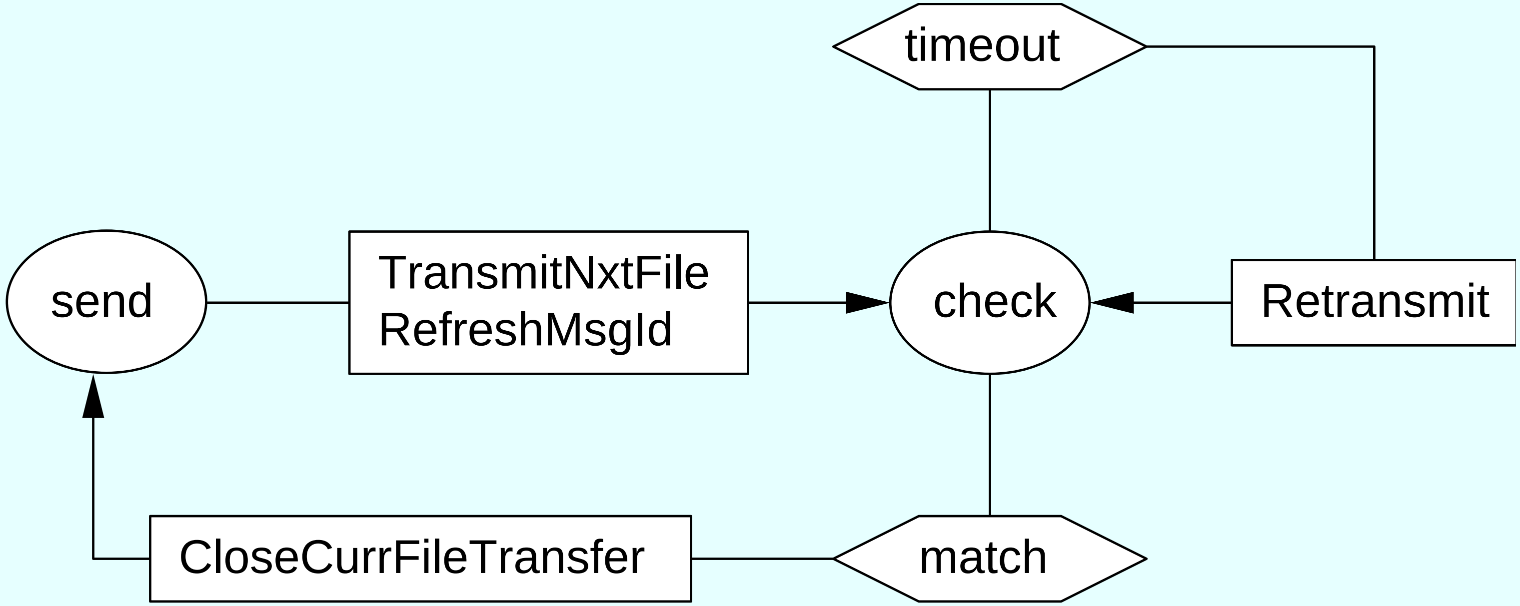


Fig. 0.2. Kermit protocol sender ASM (Alternating Bit and Sliding Window)

Example: Control State ASMs

ASM where all rules have the form

$$\textsc{Fsm}(i, \textbf{if } cond \textbf{ then } rule, j)$$

Typical for industrial control systems, protocols, business processes, etc., with a concept of *status* or *mode* or *phase* that directs complex state transformations

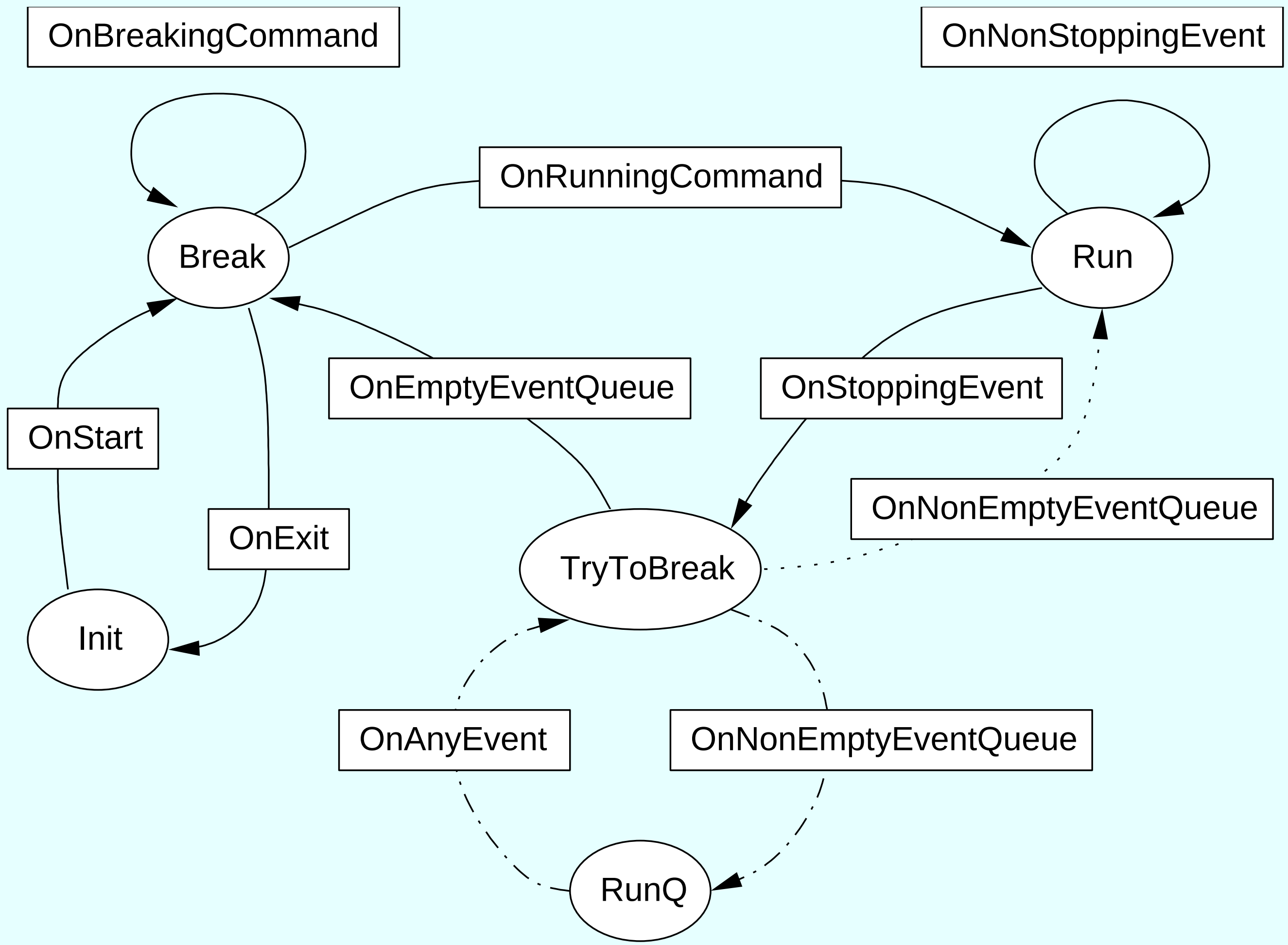


Fig. 0.3. Debugger control state ASM

Debugger Control State ASM

From a reverse-engineering case study at MSR to model a command-line debugger of CLR. Led to the discovery of a flaw in the code.

ASM Ground Models (System Blueprints)

Capture changing requirements ("what to build") in a way that is:

- *consistent and unambiguous* ('precise'),
- *simple and concise* ('flexible': abstractions that "directly" reflect the structure of the real-world problem without extraneous encoding),
- *minimal (abstract) and complete*, making all and only semantically relevant features present (model "closed" modulo some appropriately circumscribed "holes", e.g. for auxiliary functionality)

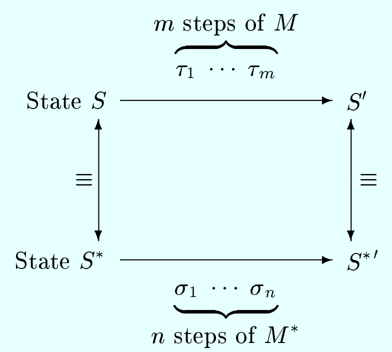
so that the resulting documentation "grounds the design in reality" as

- *understandable and checkable* (for correctness and completeness) by both domain experts (for inspection) and system designers (for verification)

ASMs allow one to calibrate the degree of precision of a ground model to the conceptual frame of the given problem domain, supporting the concentration on domain issues instead of issues of notation

ASM Refinements (Reflecting Design Decisions)

- practice-oriented method to systematically separate, structure and document orthogonal design decisions, relating different system aspects and (system architect's to programmer's) views
- supports cost-effective system maintenance and management of system changes
- supports piecemeal system validation and verification techniques



With an equivalence notion $\equiv$ between data in locations of interest in corresponding states.

The parameters for defining an ASM refinement step

- a notion of *refined state*
- a notion of *states of interest* and of *correspondence* between M-states S and M^*-states S^* of interest, including usually initial/final states (if there are any)
- a notion of abstract *computation segments* $\tau_1, \ldots, \tau_m$, where each τ_i represents a single M-step, and of corresponding refined computation segments $\sigma_1, \ldots, \sigma_n$, of single M^*-steps σ_j, which in given runs lead from corresponding states of interest to (usually the next) corresponding states of interest (the resulting diagrams are called (m, n)-diagrams and the refinements (m, n)-refinements)
- a notion of *locations of interest* and of *corresponding locations*, i.e. pairs of (possibly sets of) locations one wants to relate in corresponding states
- a notion of *equivalence* $\equiv$ of the data in the locations of interest

Definition of Correct ASM Refinement Step

Fix any notions $\equiv$ of equivalence of states and of initial and final states.
M^* is called a *correct refinement* of M if and only if for each M^*-run $S_0^*, S_1^*, \ldots$ there are an M-run $S_0, S_1, \ldots$ and sequences $i_0 < i_1 < \ldots, j_0 < j_1 < \ldots$ such that $i_0 = j_0 = 0$ and $S_{i_k} \equiv S_{j_k}^*$ for each k and either

- both runs terminate, their final states are equivalent, or
- both runs and both sequences $i_0 < i_1 < \ldots$, $j_0 < j_1 < \ldots$ are infinite

M^*-run $S_0^*, S_1^*, \ldots$ is said to simulate the M-run $S_0, S_1, \ldots$, where $S_{i_k}, S_{j_k}^*$ are the corresponding states of interest

- in (m, n)-refinements m, n may dynamically depend on states
- (m, n)-refinements with $n > 1$ and including $(m, 0), (0, n)$-steps support the feasibility of decomposing complex (global) actions into simpler (locally describable) ones
- procedural $(1, n)$-refinements with $n > 1$ have their typical use in compiler verification

Refinement and Verification Example: Leader Election

- Goal of the protocol: achieve the election of a leader among finitely many homogeneous agents in a connected network, using only communication between neighbor nodes
- $leader = max(Agent)$ with respect to a linear order $<$ among agents
- algorithmic idea: every agent proposes to its $neighb$ors its current leader *cand*idate, checks the leader $proposals$ received from its $neighb$ors and upon detecting a proposal which improves its leader candidate, it improves its candidate for its next proposal
- Correctness property to be proved: if initially every agent is without $proposals$ from its neighbors and will $proposeToNeighbors$ it$self$ as $cand$idate, then eventually every agent will $checkProposals$ with empty set $proposals$ and $cand = max(Agent)$
- Side goal: make algorithm and correctness proof extendable (e.g. to compute a shortest path to leader, its length, etc.)

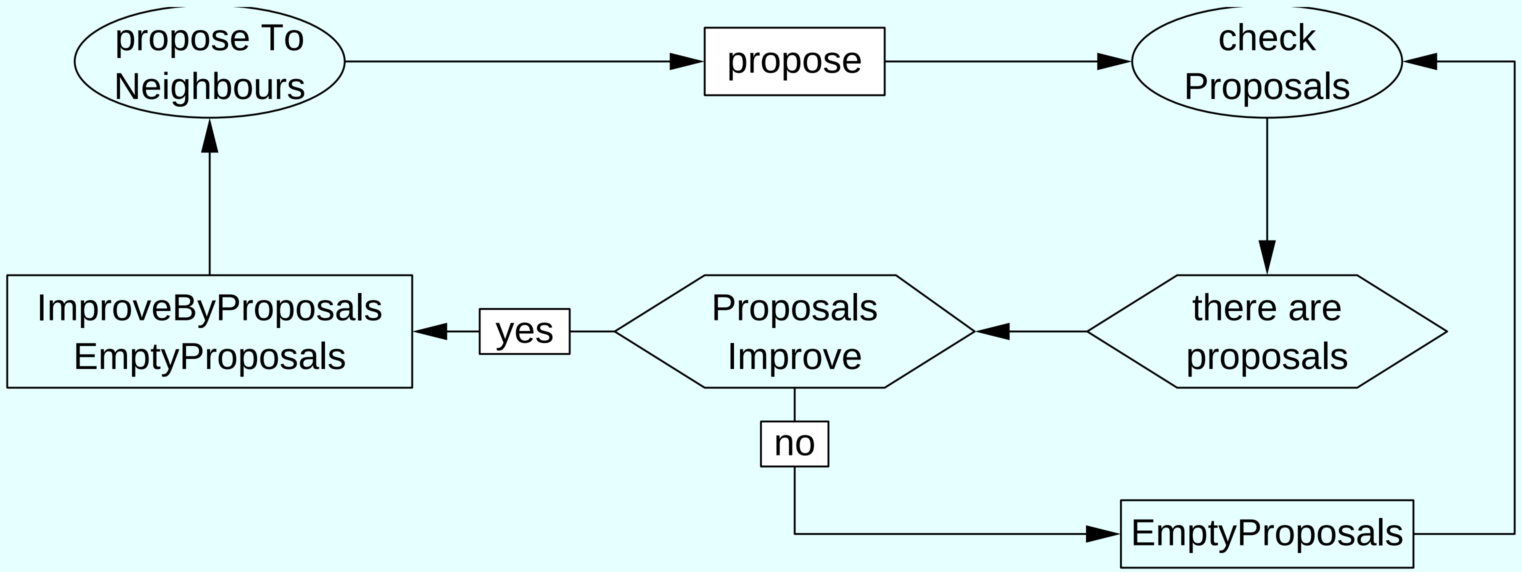


Fig. 0.4. Basic ASM of LeaderElection agents

Basic Leader Election ASM

LeaderElectionMacros =

$propose =$ **forall** $n \in neighb$ insert $cand$ to $proposals(n)$

$proposals$ improve $= max(proposals) > cand$

improve by $proposals = cand := max(proposals)$

$EmptyProposals = (proposals := empty)$

there are proposals $= (proposals \neq empty)$

Correctness Proof for LeaderElection ASM

- to be proved: if initially every agent is without $proposals$ from its neighbors and will $proposeToNeighbors$ it$self$ as $cand$idate, then eventually every agent will $checkProposals$ with empty set $proposals$ and $cand = max(Agent)$
- assume: every enabled agent will eventually make a move
- use an induction on
  - runs and
  - $\sum\{leader - cand(n) \mid n \in Agent\}$, measuring the distances of candidates from the leader

Refinement Step to Compute a Shortest Path

- Refinement idea: provide for every agent (except for the leader), in addition to the leader candidate, also a neighbor which is currently known to be closest to the leader, together with the minimal distance to the leader via that neighbor
- Pure data refinement: enrich $cand$ and $proposals$ by
  - a *nearNeighb* $: Agent$ with minimal distance to the leader,
  - the *distance* $: Distance$ to the leader candidate (e.g. $Distance = \mathbb{N} \cup \{\infty\}$)

  so that *proposals* $\subseteq Agent \times Agent \times Distance$ (triples of leader $cand$, $nearNeighbor$ and $distance$ to the candidate leader)
- initially assume $nearNeighbor =$ **self** and $distance = \infty$, except for the $leader$ where $distance = 0$.

Refined MinPathToLeader Macros

MinPathToLeaderMacros =

$propose =$ **forall** $n \in neighb$

insert $(cand, nearNeighb, distance)$ to $proposals(n)$

$proposals$ improve $=$ **let** $m = Max(proposals)$ **in**

$m > cand$ **or**

$(m = cand$ **and** $minDistance(proposalsFor\ m) + 1 < distance)$

improve by $proposals =$

$cand := Max(proposals)$

update PathInfo to $Max(proposals)$

update PathInfo to $m =$ **choose** (n, d) **with**

$(m, n, d) \in proposals$ **and** $d = minDistance(proposalsFor\ m)$

$nearNeighb := n$

$distance := d + 1$

Extending Correctness Proof by Shortest Path Property

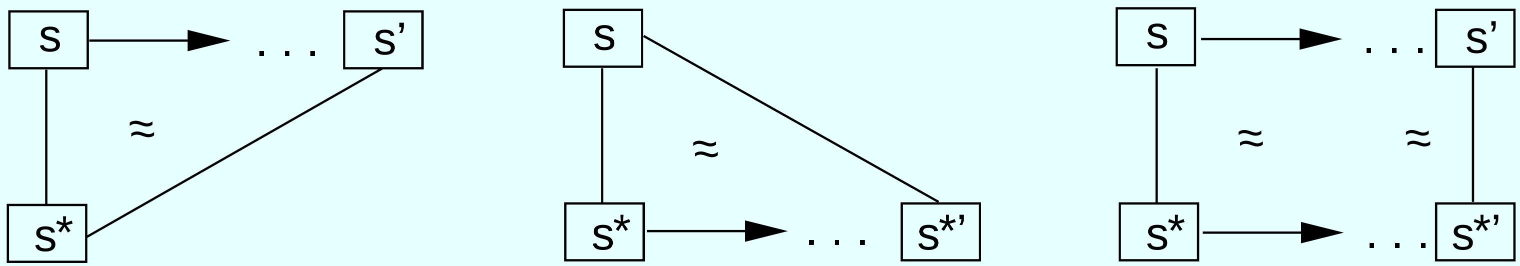
- Proposition: In every distributed run of agents equipped with the ASM computing a minimal path to the leader, eventually for every agent holds:
  - cand=max(Agent)=leader
  - *distance=minimal distance of a path from agent to leader*
  - *nearNeighbor = a neighbor of agent on a minimal path to the leader (except for leader where nearNeighbor=leader)*
  - ctl_state = checkProposals
  - proposals = empty
- Proof: induction on runs and on $\sum\{leader - cand(n) \mid n \in Agent\}$ enhanced by side induction on the minimal distances in $proposalsForMax(proposals)$.

Method of Splitting Complex Proofs by ASM Refinements

Split the overall task of proving P^* for S^*, which for real-life systems is usually too complex to be tackled in a single blow, into a series of manageable subtasks (1)–(3), each step reflecting a part of the design

1. build an abstract model S,
2. prove a possibly abstract form P of the property in question to hold under appropriate assumptions for S,
3. show S to be correctly refined by S^* and the assumptions to hold in S^*.

Looking for invariants for ASM refinement correctness proofs



Idea: decompose commuting diagram into more basic diagrams with end points s, s^* which satisfy an invariant $\approx$ implying the to be established equivalence $\equiv$

- *(m,0)-triangles*: computation segments where only the abstract run makes progress reaching an $s' \approx s^*$ by a positive number m of steps
- *(0,n)-triangles*: computation segments where only the concrete run makes progress reaching an $s^{*\prime} \approx s$ by a positive number n of steps
- *(m,n)-trapezoids*: representing a computation segment which leads in $m > 0$ steps to an s' and in $n > 0$ steps to an $s^{*\prime}$ such that $s' \approx s^{*\prime}$. NB. Cases $m < n$, $m > n$ (typical for optimizations), $m = n$ allowed

Schellhorn's Forward Simulation Condition FSC

For every pair (s, s^*) of states, if $s \approx s^*$ and not both are final states, then

- either the abstract run can be extended by an $(m, 0)$-triangle leading in $m > 0$ steps to an $s' \approx s^*$ satisfying $(s', s^*) <_{m0} (s, s^*)$ for a well-founded relation $<_{m0}$ limiting successive applications of $(m, 0)$-triangles,
- or the refined run can be extended by a $(0, n)$-triangle leading in $n > 0$ steps to an $s^{*\prime} \approx s$ satisfying the condition $(s, s^{*\prime}) <_{0n} (s, s^*)$ for a well-founded relation $<_{0n}$ limiting successive applications of $(0, n)$-triangles,
- or both runs can be extended by an (m, n)-trapezoid leading in $m > 0$ abstract steps to an s' and in $n > 0$ refined steps to an $s^{*\prime}$ such that $s' \approx s^{*\prime}$.

Theorem on Decomposition of ASM Refinement Diagrams

M^* is a correct refinement of M with respect to an equivalence notion $\equiv$ and a notion of initial/final states if there is a relation $\approx$ (a coupling invariant) such that

1. the coupling invariant implies the equivalence,
2. each refined initial state s^* is coupled by the invariant to an abstract initial state $s \approx s^*$,
3. the forward simulation condition FSC holds.

This theorem, proved by Schellhorn using KIV, constitutes the basis of

- G. Schellhorn, W. Ahrendt: The WAM Case Study: Verifying Compiler Correctness for Prolog with KIV. In W.Bibel, P. Schmitt (Eds): Automated Deduction A Basis for Applications. Vol.3, Ch.3, Kluwer 1998
- G. Schellhorn, W. Ahrendt: Reasoning About Abstract State Machines: The WAM Case Study. JUCS 3 (4) 1997, 377-413

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, e.g. abstract operational and declarative/functional/axiomatic
- 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 (Glässer et al. 2005, Java-based)
- 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

References

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

Springer 2003. pp.X+438. **Slides for courses** on single chapters, themes and case studies are to be found in ppt and pdf format on the CD coming with the book and are also downloadable from the website:

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

Comprehensive case study (ASM modeling, validation, verification): R. F. Stärk and J. Schmid and E. Börger: *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer 2001.

E. Börger: *The ASM Refinement Method*

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

E. Börger: *The ASM ground model method as a foundation of requirements engineering*. In: LNCS 2772 (2003), 145-160