The Abstract State Machines Method for High-Level System Design & Analysis

Egon Börger

Dipartimento di Informatica, Universita di Pisa

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

Finite State Machines (McCullock/Pitts 1943)

– The FSM Model of Computation

- finitely many internal states: externally invisible "control" states representing a bounded memory
- external input: elements from a finite set
- dynamic behavior: reaction to input consumed by
  - updating internal state (δ-function Nxtctl)
  - providing externally visible output (λ-function Nxtout)

MealyFsm (in, out, Nxtctl, Nxtout) =

control := Nxtctl(control,in)

out := Nxtout(control,in)

Extend FSMs to VMs by general notion of data/data change

memory: set of abstract updatable locations $l(a_1,\ldots,a_n)$

- containing values of whatever type: objs, sets, fcts,…
- updatable via assignments loc:= val (updates)

grouping of data ("modular memory structure")

- into tables: association of a value to each entry $l(a_1,\ldots,a_n)$
  - also called interpretation of an array (function or predicate)
- into states: sets of tables
  - also called (Tarski) structure of given signature

state change by updating some locations

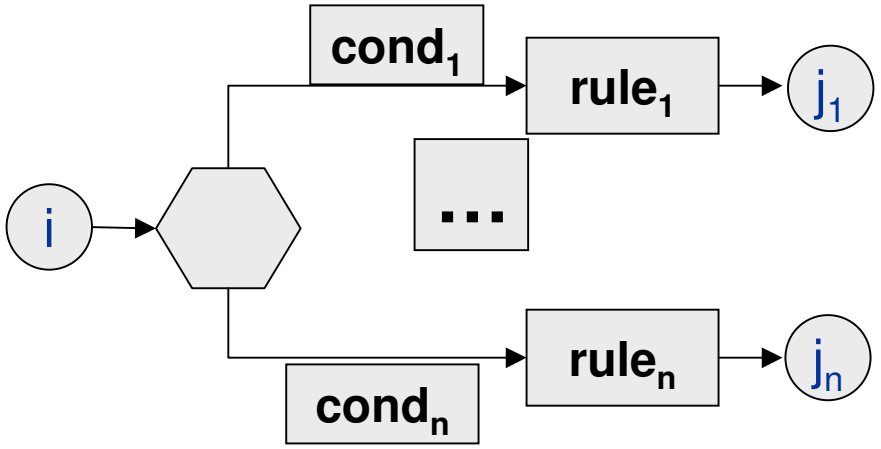
- i.e. using guarded updates via transition rules of form

If cond then Updates

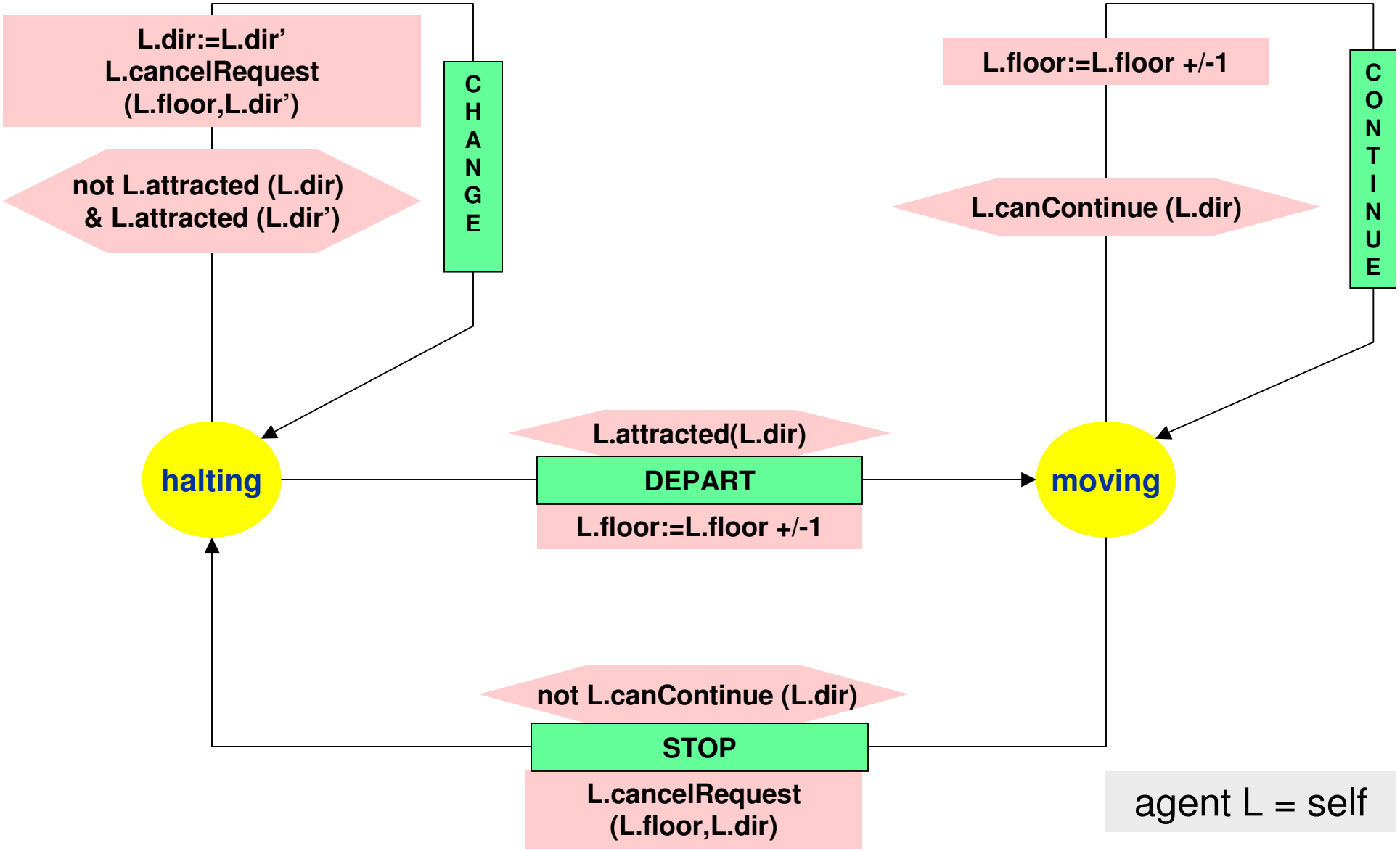
Abstract State Machines (mathematical VMs)

- state an arbitrary structure (Tarski)
- state transformation by simultaneous execution of finitely many rules of form
  - array updates f(s,…):=t (function updates)
  - if cond then rule (guarding)
  - choose x with P(x) in rule (non-determinism)
  - forall x with P(x) do rule (parallelism)
  - let x=t in rule (common notational shorthands)

control state ASM with rules
  if control=i and cond
  then control:=j
    rule



Example: Control State FSM/ASM for Lift



See AsmBook 2.3

What are ASMs good for?

Basis for rigorous system development method which allows one to effectively

couple specification & detailed design

- by constructing
  - accurate ground models (blueprints) at the appropriate level of rigour (LNCS 452 & 2772)
  - linked seamlessly to executable code via systematic stepwise refinement of models (LNCS 452 & J.FAC 2003)
- in a way the practitioner can verify & validate by reasoning & experimentation at the appropriate degree of detail

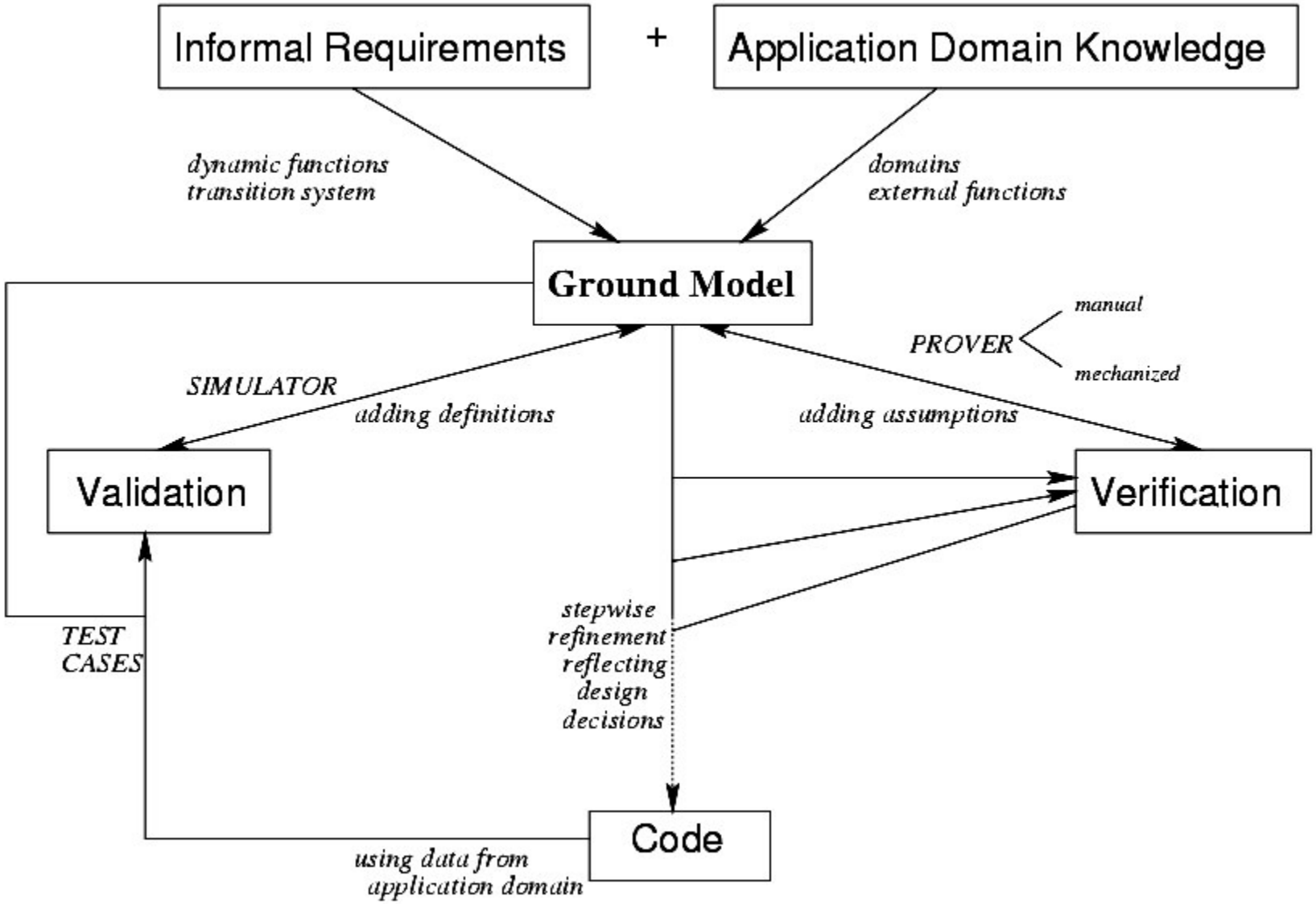
ASMs support major activities in sw lifecycle (1)

- requirements capture by constructing satisfactory ground models, i.e. accurate high-level system blueprints, serving as precise contract and formulated in a language which is understood by all stakeholders
- detailed design by stepwise refinement, bridging the gap between specification and code design by piecemeal, systematically documented detailing of abstract models down to executable code, modularizing orthogonal design decisions ("design for change")

ASMs support major activities in sw lifecycle (2)

- **validation** of models by their simulation
  - based upon the mathematical notion of ASM run, which is supported by numerous tools to execute ASMs:
    - ASM Workbench (ML-based, DelCastillo 2000)
    - AsmGofer (Gofer-based, Schmid 1999)
    - XASM (C-based, Anlauff 2001)
    - AsmL (.NET-based, MSR/FSE 2001)
- **verification** of model properties by techniques for proving or refuting (counterexample construction)
  - whether "by head" or tool supported
    - e.g. by KIV, PVS, Isabelle, AsmPTP, model checkers
- **documentation** for inspection, reuse and maintenance
  - by providing, through analysis of ground and intermediate models, explicit descriptions of the software structure and of the major design decisions

ASM Method: Seamlessly from Specs via Design to Code



Major applications of ASMs

- industrial standards: OASIS BPEL4WS, ECMA/ISO C#, ITU-T SDL-2000, IEEE VHDL93, ISO Prolog
- programmming languages: all major real-life languages, e.g. SystemC, Java/JVM bytecode, domain-specific languages (at Swiss Union Bank)
- architectural design: verification, e.g. of pipelining & VHDL-based hw design (Schmid at Siemens), architecture/compiler co-exploration (Teich Paderborn/Erlangen)
- reengineering and design of industrial control systems: Falko project & mobile telephony network components (Siemens), debugger specification & UPnP (Microsoft)
- protocols: authentication, cryptography, cache-coherence, routing-layers for distributed mobile ad hoc networks, group-membership
- compilation: verification of comp.schemes/compiler back-ends
- modeling e-commerce and web services
- simulation & testing: fire detection system in coal mines, simulation of railway scenarios (Siemens), implementation of behavioral interface specs on .NET platform & conformence test of COM components (Microsoft), compiler testing

ASM Method: Separation of Different Concerns

- Separating orthogonal design decisions
  - supported by combining abstract operational descriptions of system dynamics with declarative (e.g. functional or axiomatic) definitions of statics
    - to keep design space open (specify for change: avoiding premature design decisions & documenting design decisions to enhance maintenance)
    - to structure design space (rigorous interfaces for system (de)composition, laying the ground for the system architecture)

See E.B. in FDL2003

- Separating design from analysis (defn from proof)
  - validation (by simulation) from verification (by reasoning)
  - verification levels (degrees of detail)
    - reasoning for human inspection (design justification)
    - rule based reasoning systems
      - » interactive systems
      - » automatic tools: model checkers, automatic thm provers
- Linking system levels by abstraction and refinement

ASM method: 3 constituents

- notion of ASM
  - Y.Gurevich 1984-1995, in particular “Lipari Guide” 1995
- ground model technique
  - See E.Börger: The ASM ground model method as a foundation of requirements engineering, LNCS 2772 (2003) 145-160. See also LNCS 452
- ASM refinement technique
  - See E.Börger: The ASM refinement method, Formal Aspects of Computing 15 (2003) 237-257. See also LNCS 452
- to be combined with appropriate methods of
  - ASM validation
  - ASM verification

ASM notion: rigorous semantics for truly abstract (pseudo-) "code"

- Semantics of guarded function updates:

  if cond then … f $(t_1,\ldots,t_n)$:=t …

- In the current state (structure) S:
  - determine all the fireable rules (s.t. cond is true in S)
  - compute all update expressions t_i,t
  - execute simultaneously all the assignments
- The updating yields the next state S'

For a formalization by deduction rules see AsmBook

Semantics of ASM (cont'd): choose/forall rules

- Semantics of choose x satisfying cond
  R

  in the current state S:
  – select an element e which satisfies cond in S
  – execute R(e) in S to yield next state S'

- Semantics of forall x satisfying cond
  R

  in the current state S, execute simultaneously R(e) for each e satisfying cond in S to yield next state S'

Componentization via Classification of Locs/Fcts

- Static: values independent on states of M
- Dynamic: values depend on states of M
  - in (monitored) : only read (not updated) by M, written only by environment of M Exl: sensors
  - out: only written (not read) by M, only read (not written) by environment Exl: actuators
  - controlled: read and written by M
  - shared: read and written by M and by environment (protocol needed for consistency of updates)
- Derived: values computed by a fixed scheme

  from monitored/static fcts

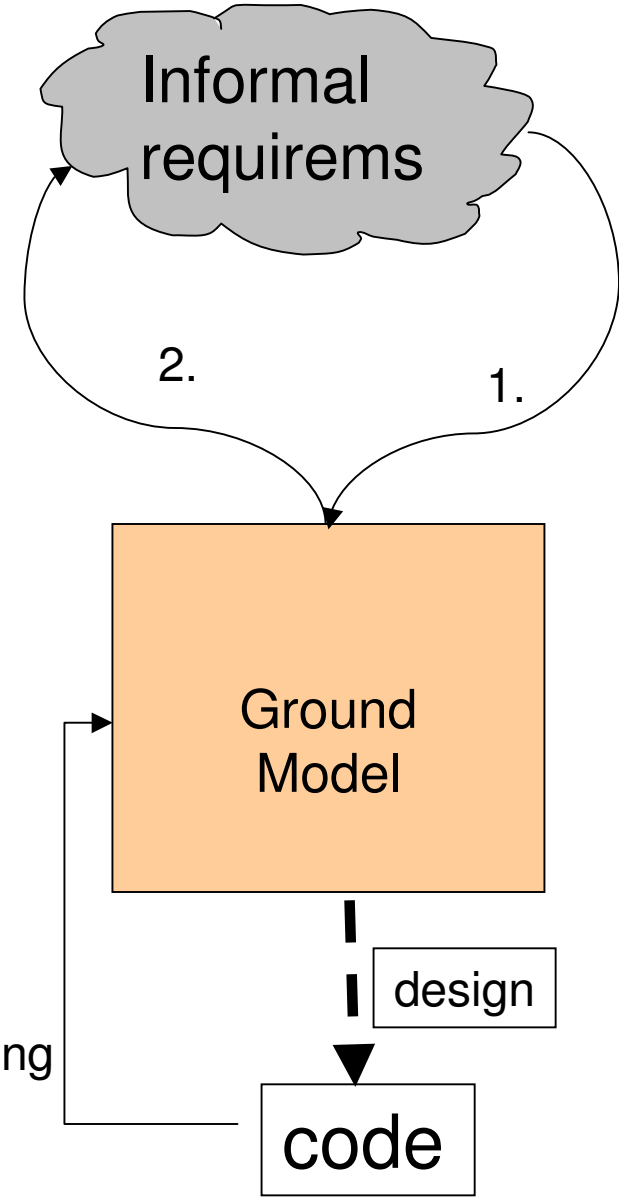
What ground models provide for requirements

1. **Requirements capture**
   documenting relevant application domain knowledge for designer

2. **Requirements inspection** makes correct/completeness checkable for user/customer (no infinite regress)
   - Verification of properties
   - Validation: mental/machine simulation (of user scenarios or components) supported by operational nature of model

3. Makes requirements traceable by relating them to design

4. Provides **test plan** basis



See E.B. LNCS 2772

Ground Model Exl: Falko Project (Siemens 3'98-5'99)

Main purpose of FALKO: Construct & validate timetables for tram/sub-/railway lines (system of 300 K loc)

- **Timetables** constructed offline from raw data which
  - **can be input manually via a GUI**
  - **can be read in from files written in established formats**
- **Validation** done by simulation
  - **3 main components + 1 hidden discrete event simulation kernel controlling the simulation**
- **Defective component implementing the railway process reengineered using ASMs (modeled and designed)**
  - **integrating an existing library for numerical computations (computing train velocities, trip times etc.) designed and hand coded conventionally**

Slides courtesy of Peter Päppinghaus

The FALKO Project: ASM Effort

Total Effort for Railway Process Model: 66 pw (person weeks)

- Ground Model Construction (High-Level Design)
  - **Requirement eliciation and specification based on predecessor system, developed in meetings of the design team, documented by minutes of the meetings (4 persons 2 weeks)**
  - **Design of 1st draft of executable ASM model (1 person 8 weeks)**
  - **Several cycles of testing and debugging using the ASM Workbench (developed by Giuseppe Del Castillo as part of his Doctoral Thesis at University of Paderborn) (1 person 8 weeks + 1 person 11 weeks)**
  - **Review of 2nd draft of ASM model by design team plus external reviewers (6 persons 1 week)**
  - **Several cycles of improving, testing and debugging (2 persons 5 weeks)**

- Implementation
  - **Development of ASM-SL to C++ Code Generator (developed by Joachim Schmid as part of his Doctoral Thesis at University of Ulm, 1 person 4 weeks)**
  - **Specification and implementation of additional handwritten C++ code (1 person 2 weeks)**
  - **Integration of FALKO system including testing and debugging (3 persons 3 weeks)**
  - **Documentation of railway process model component (as collection of HTML documents with literate programming features, linked to ASM Workbench) and final polish (1 person 6 weeks)**

Comparison: ca. 110% of estimated total effort for conventional software design

Size of ASM Ground Model and C++ Code for Railway Process Model

ASM Ground Model (source of C++ code generation)

- ca. 3 000 lines of ASM Workbench code
- 120 rules
- 315 functions and relations (240 functions, 75 relations)
  - 71 dynamic
  - 69 external
  - 59 static
  - 116 derived

C++ Code

- ca. 9 000 lines of generated C++ Code
- ca. 2 900 additional lines of handwritten C++ Code, consisting of
  - ca. 400 lines wrapper code for interfacing to other components of FALKO
  - ca. 2 500 lines low-level library code
- Railway process model of prototypical predecessor system:

  ca. 20 000 lines of (handwritten) C++ code

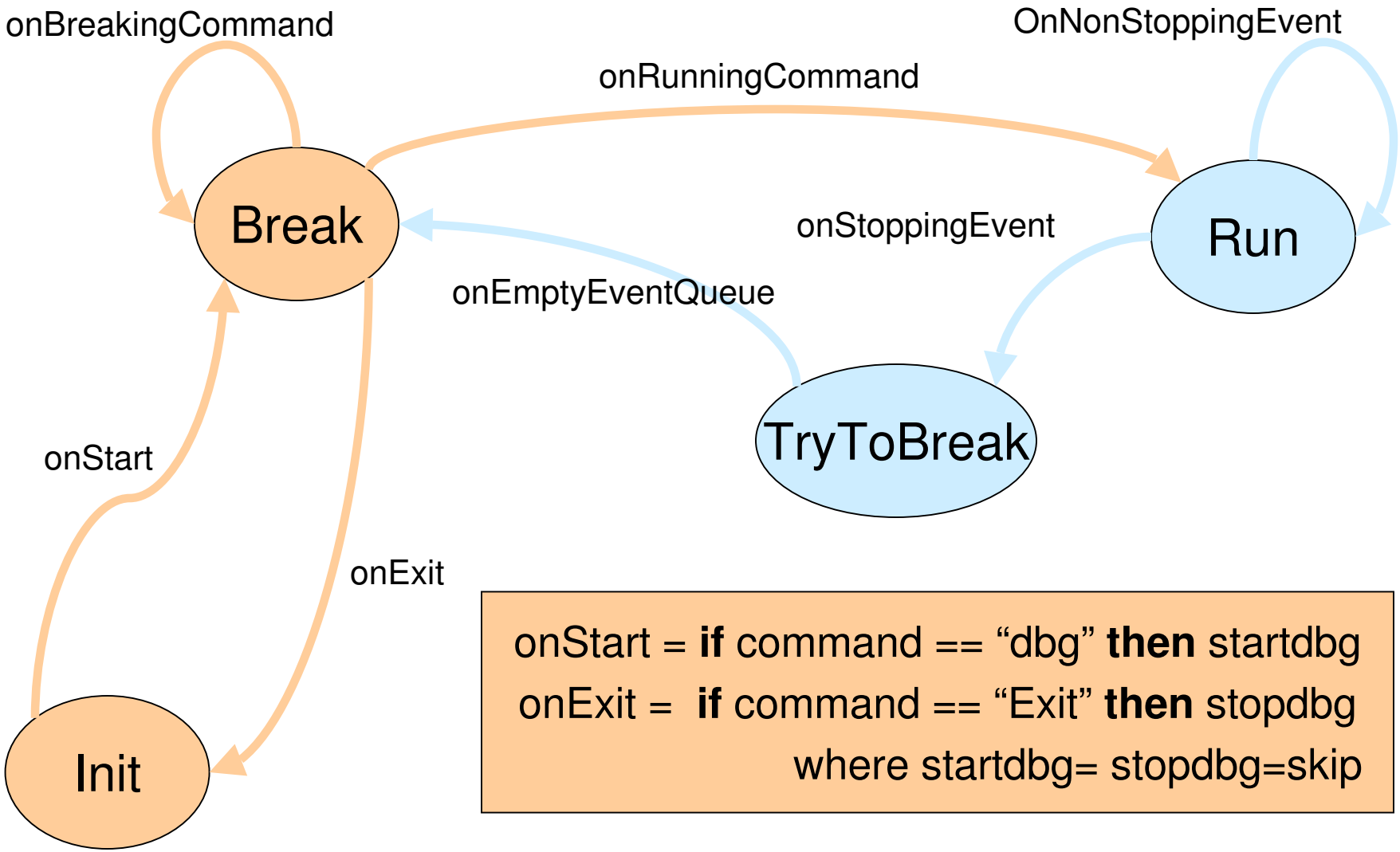
Falko Experience in Use of ASMs for Software Design

- Ground Model Construction (High-Level Design)
  - **Developers and reviewers had no problems to understand the formal specification (ASM model)**
  - **Tests with ASM model uncovered, at an early stage, bugs also in other components of the package**
- Implementation
  - **Coherence of specification and implementation by seamless tool support**
  - **Performance loss of generated code tolerable even for product quality code**
- Maintenance
  - **4 installations at Subway Vienna Operator since March 1999, 1 in daily use**
  - **Customer reported no bugs so far**
  - **2 bugs discovered in tests run by FALKO developers, temporary fixes for the 2 bugs (by handhacking the generated C++ code) later replaced by recompiling updated ASM model**
- Development Environment **built-up, supporting seamless flow from spec to code**

References to Falko Project

- Egon Börger, Peter Päppinghaus, Joachim Schmid**: Report on a Practical Application of ASMs in Software Design**
  - In: Y. Gurevich et al.(Eds.): Abstract State Machines. Theory and Applications. Springer LNCS 1912, 2002, 361-366
- Giuseppe Del Castillo**: The ASM Workbench. A Tool Environment for Computer-Aided Analysis and Validation of Abstract State Machine Models**
  - Dissertation, Heinz Nixdorf Institut, Universität Paderborn, 2001, pp.iv + 212, ISBN 3 - 9311 466 – 82 – 5
- J. Schmid: **Compiling Abstract State Machines to C.**
  - In: Journal of Universal Computer Science 7 (11), 2001, 1069-1088

ExI: High-Level Debugger Model (MS/MSR 2000)

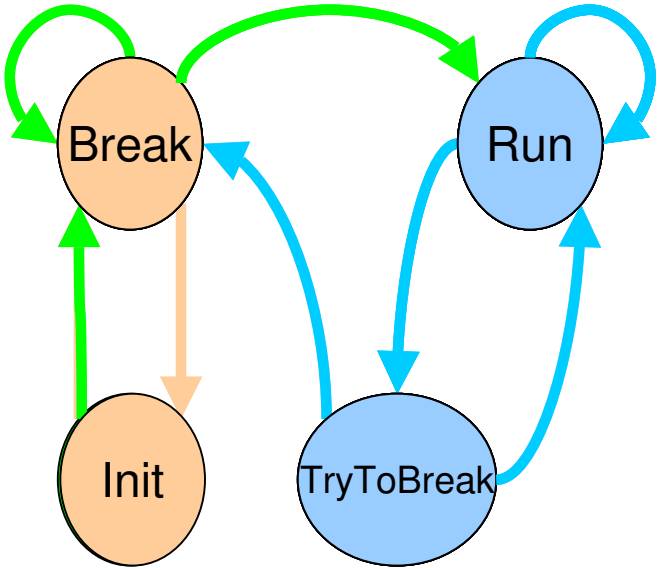


onStart = **if** command == “dbg” **then** startdbg
onExit = **if** command == “Exit” **then** stopdbg
where startdbg= stopdbg=skip

Slide courtesy
M. Barnett, M. Veanes

Experimenting user scenarios with ASM model

Row t exhibits state after t-th step of control model

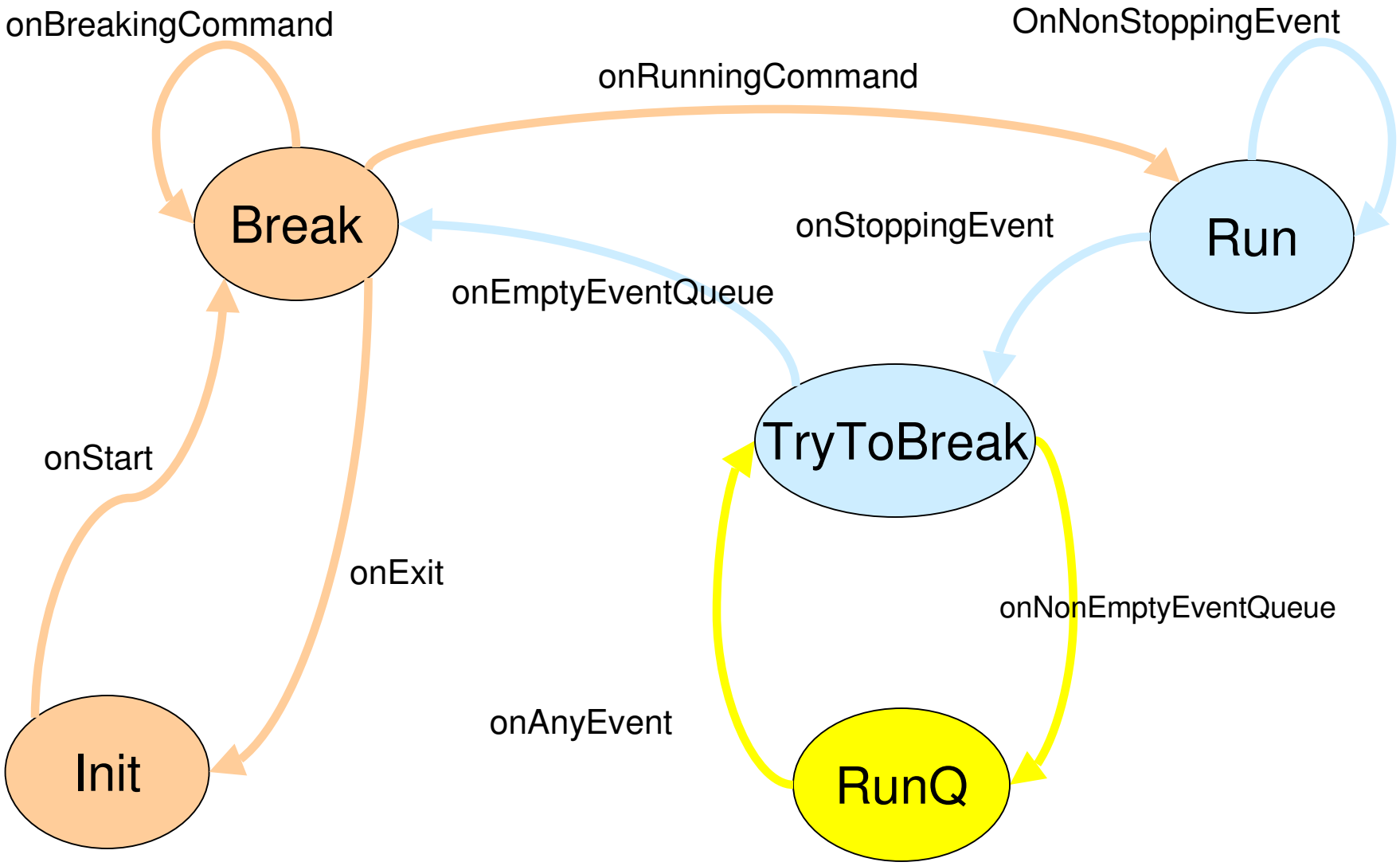


User	Environment	Events in queue
start		
b hello.cpp:13		
run hello.exe		
	Created process	
	.	
	.	
	Loaded module	
	Created 1st thread	
		False
continue		
	Hit breakpoint	
		True
	Loaded Class	
		False

Test of the Control Model showing an undesired behavior (which then was found to have been fixed at the same time also in the C++ code)

After non-stopping class loading event in Run mode, Break mode is unreachable although breakpoint was reached

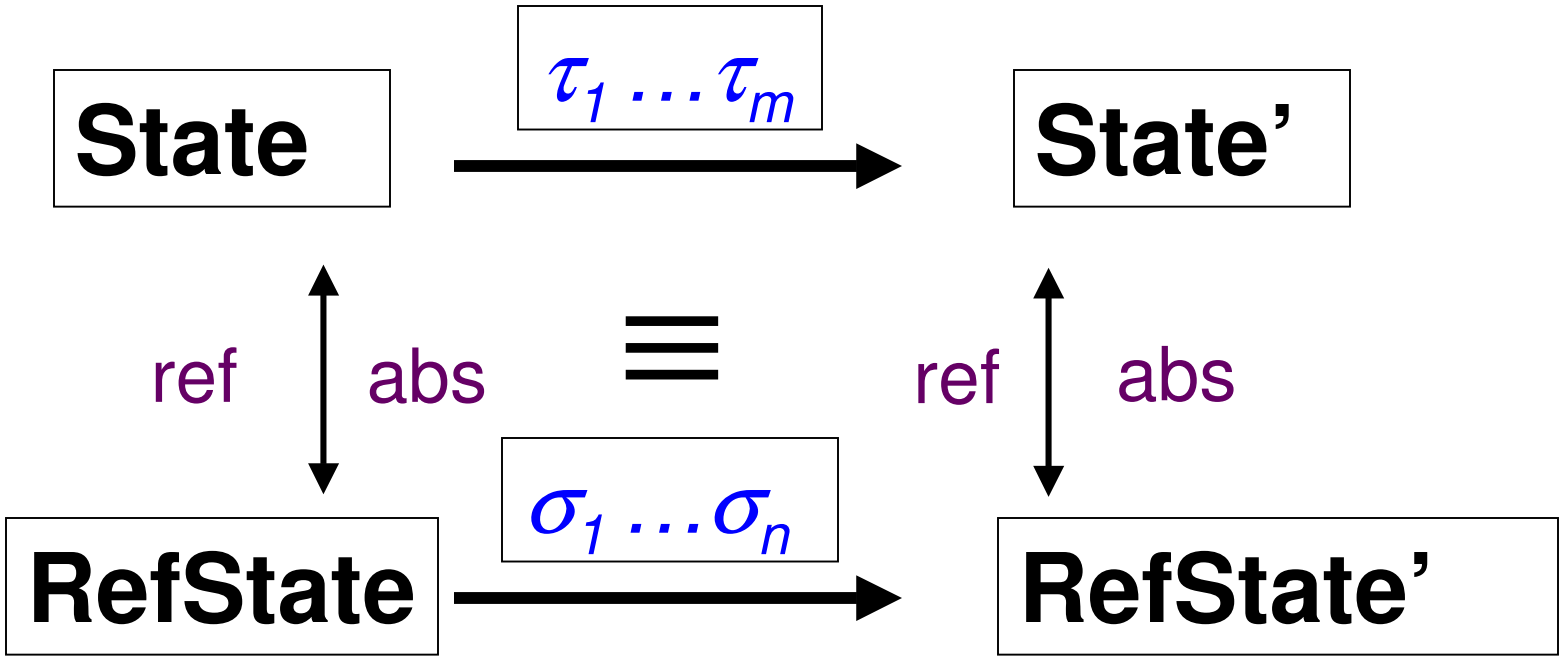
ASM model for debugger: revised



See Proc. ASM2000 in LNCS 1912

Scheme for Correct ASM Refinement/Abstraction Step

Supports design communication, reuse, system documentation and maintenance



≡ defined

- relating the locations of interest
- in states of interest
- reached by comp segments of interest

See E.B. in Formal Aspects of Computing 2003

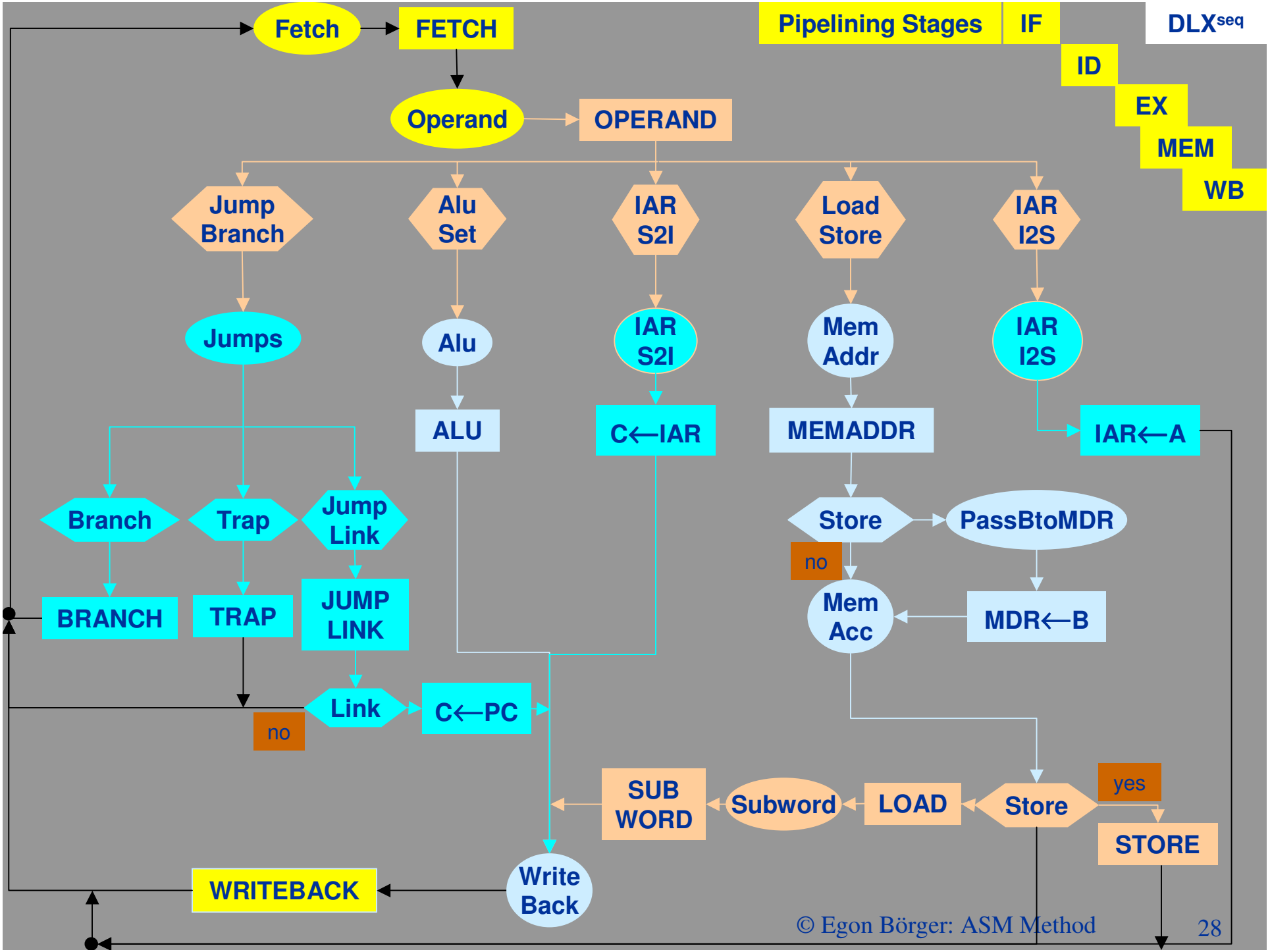
Defining **correctness of a refinement** M* of M

- Fix any notions ≡ of equivalence of states & of initial/final states
- Idea of correctness: refined runs simulate abstract ones
- Definition. M* is a correct refinement of M iff

  every (infinite) refined run simulates an (infinite) abstract run with equivalent corresponding states
  - i.e. for each M*-run S*(0), S*(1),… there is an M-run S(0), S(1),… , either both terminating or both infinite, with infinite sequences $i_0 < i_1 < \ldots$, $j_0 < j_1 < \ldots$ such that $S(i_k) \equiv S^*(j_k)$ for each k, including the initial states ($i_0 = j_0 = 0$) and the final ones (if any)
    - Wlog at final states, the state sequence becomes constant
      i.e. S(r) = S(r+k) for each final S(r) and each k, same for S*

Completeness condition for ASM refinements

- Completeness idea: abstract runs are simulated by (correspond to) refined ones, symmetrically to how for correctness refined runs simulate abstract ones
- Def. M* is a complete refinement of M
  iff M is a correct refinement of M*
- Related concepts/terminology:
  - "preservation of partial correctness" for correct refinement (wrt terminating runs)
  - "preservation of total correctness" for complete refinement (adding to the correctness condition for terminating runs that every infinite refined run admits an infinite abstract run with an equivalent initial state)
  - "bisimulation" or "interpreter equivalence" for correct and complete refinement (wrt terminating runs considering only the input/output behavior)



Pipelining Stages

IF

ID

EX

MEM

WB

DLX^{par} Rules

Verified stepwise refinement see LNCS 1212

FETCH

- Ideally all rules fire simultaneously
  - one per instruction in its successive stages

OPERAND **Preserve**

- The problem: how to guarantee that no conflicts arise when an instruction exec uses data which have to be computed by a preceding instruction whose pipelined execution is not yet terminated

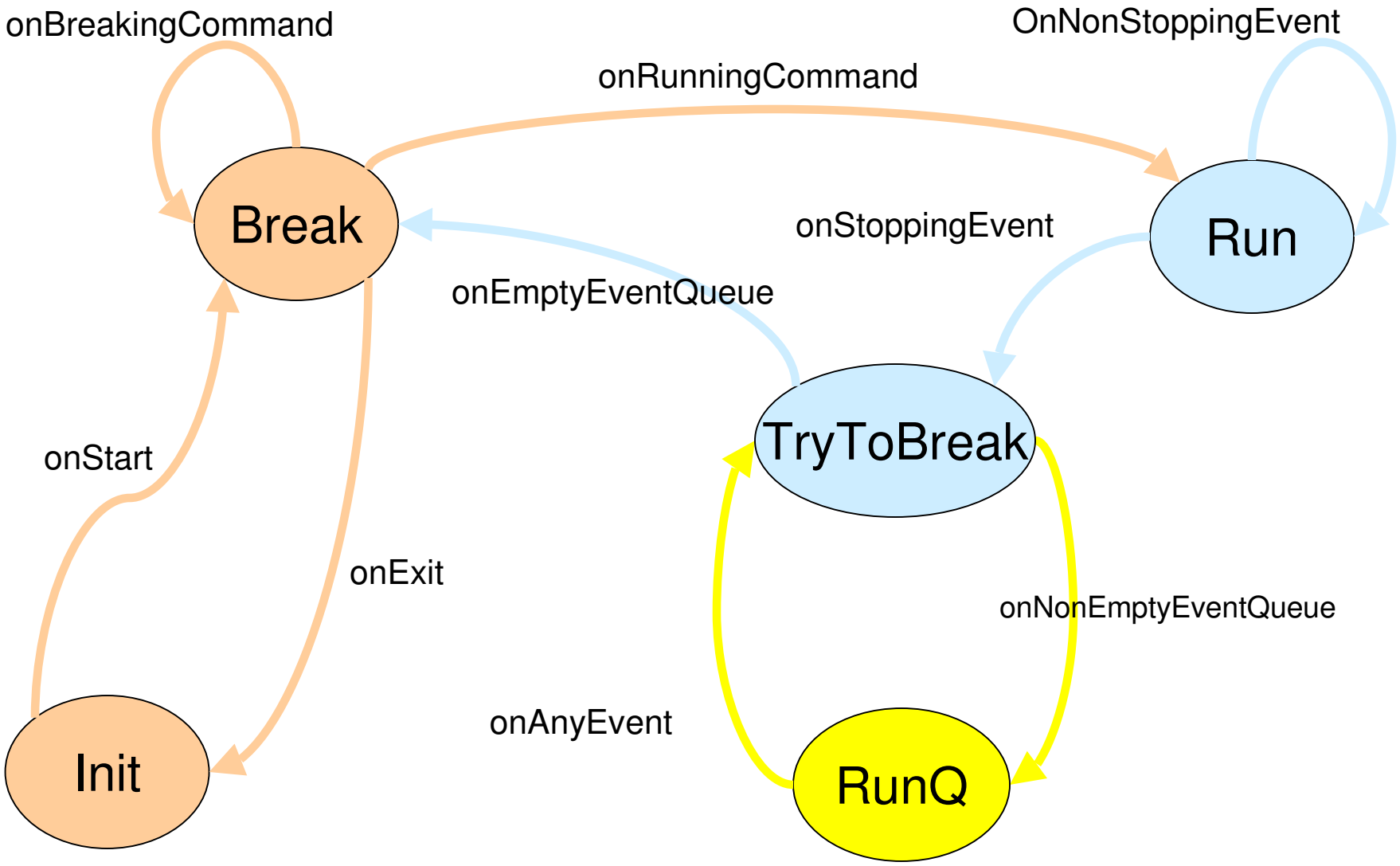
ALU **MEMADDR** **PassBtoMDR** **MOVI2S** **MOVS2I**

BRANCH **TRAP** **JUMP LINK** **LINK** **Preserve1**

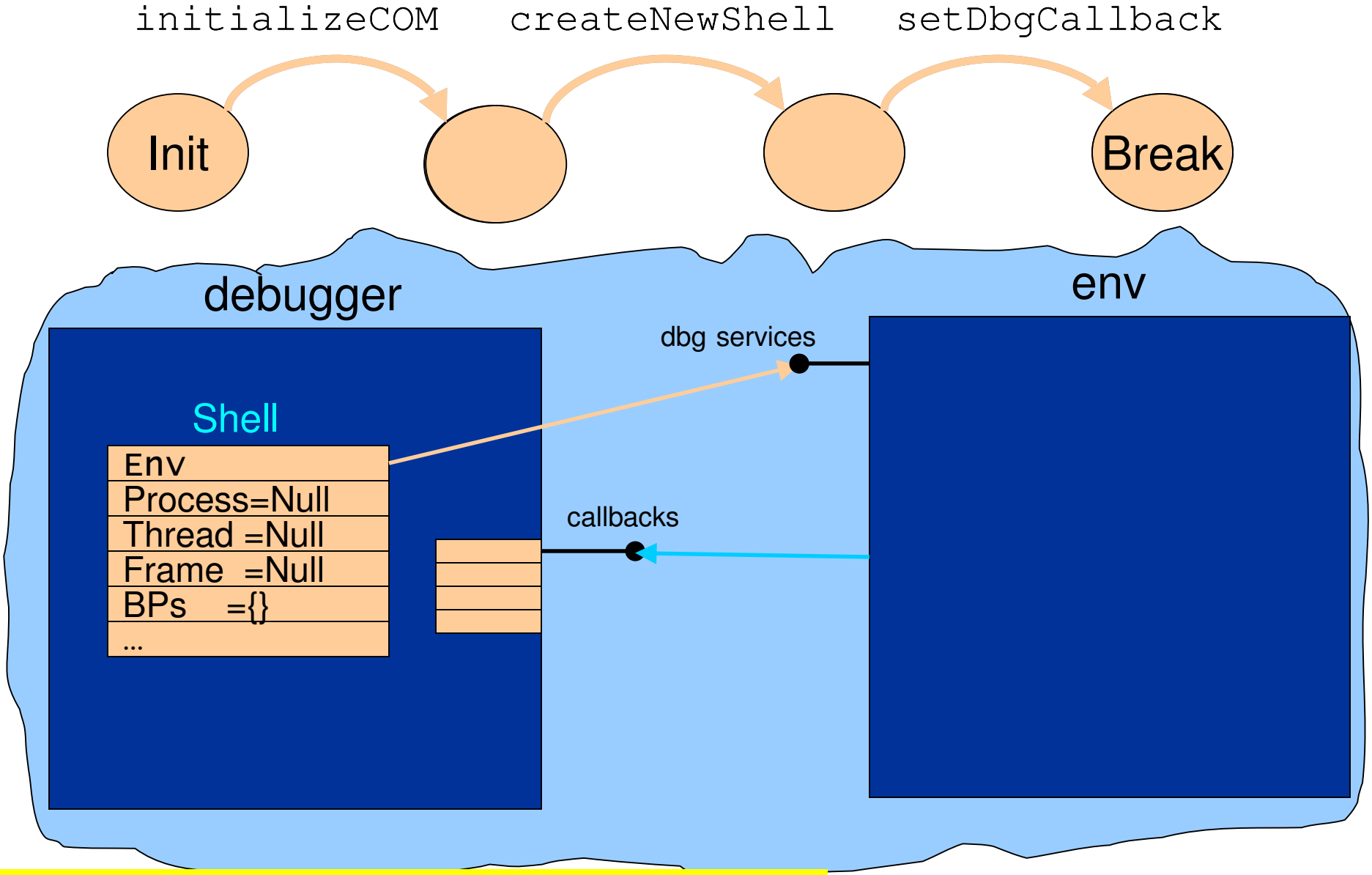
LOAD **STORE** **Preserve2**

WRITEBACK

To-be-Refined Debugger Model



Sequential submachine refinement of machine **onStart** into a sequence of three submachines



Submachine Concept: Proc. ASM2000 in LNCS 1862

More Exls for Design & Verification of ASM Hierarchies

Control Systems: Production Cell (model checked, refinement to C++ code:JUCS 1997), Steam Boiler (refinement to C++ code, LNCS 1165), Light Control (executable requirements model, JUCS 2000)

Compiler correctness

ISO Prolog to WAM: 12 refinement steps, KIV verified

reflecting backtracking, structure of predicates, structure of clauses, structure of terms & substitution, optimizations.

reused for PROTOS-L (poly types) and CLP (R) (constraints) (FAC '96)

Occam to Transputer :15 models (Computer Journal '96)

exhibiting channels, sequentialization of parallel procedures, pgm ctrl structure, env, transputer datapath and workspace, relocatable code (relative instr addresses & resolving labels)

Java to JVM: language and security driven decomposition (Jbook)

horizontal sublanguage levels (reused for C#)

imperative, modules, oo, exceptions, concurrency (delegates, unsafe)

vertical JVM levels (modular compositional structuring)

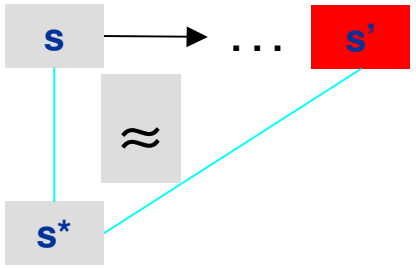
trustful execution, defensive run time checks, diligent link time checks, loading

Looking for invariants to prove ASM refinement correctness

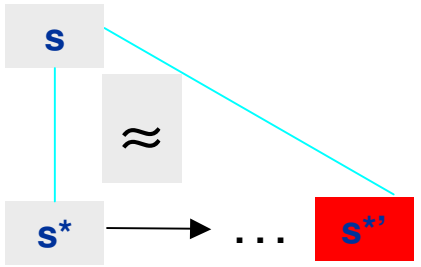
- Idea: find commuting diagrams with end points s, s* which satisfy an invariant ≈ implying the to be established equivalence ≡
- Realization: for each pair of corresponding states - not both final - satisfying ≈, follow the two runs to find a successor pair s', s*' (of corresponding states satisfying ≈)
- Two cases are possible for such run extensions:
  - only one of the two runs can be extended
    - the abstract one, producing an (m,0)-diagram
    - the refined one, producing a (0,n)-diagram
  - both runs can be extended

Extending runs by triangles and trapezoids

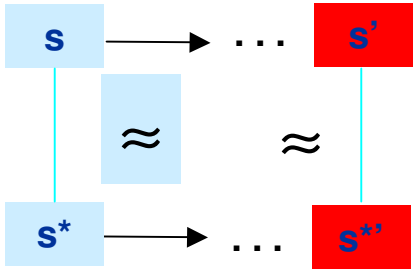
(m,0)-triangle: compute segment leading in m>0 steps to an s'≈s*



(0,n)-triangle: compute segment leading in n>0 steps to an s*'≈s



(m,n)-trapezoid:
compute segment leading
in m>0 steps to an s'
in n>0 steps to an s*'
such that s'≈s*'
where m>n or m=n or m<n



Definition of the forward simulation condition FSC(s,s*)

If $s \approx s^*$ and not both s,s^* are final states, then

- either the abstract run can be extended

  by an (m,0)-triangle

  leading in m>0 steps to an $s' \approx s^*$ with $(s',s^*) <_{m0} (s,s^*)$

- or the refined run can be extended

  by a (0,n)-triangle

  leading in n>0 steps to an $s^{*'} \approx s$ with $(s,s^{*'}) <_{0n} (s,s^*)$

applying triangles successively must be well-founded

- or both runs can be extended

  by an (m,n)-trapezoid leading

  in m>0 abstract steps to an s'

  in n>0 refined steps to an $s^{*'}$

  such that $s' \approx s^{*'}$

NB. A minor modification covers also nondeterministic ASMs

Schellhorn's coupling invariant for correct ASM refinements

Theorem. M* is a correct refinement of M

wrt an equivalence notion $\equiv$ and a notion of initial/final states

if there is a relation $\approx$ such that

- the coupling invariant $\approx$ implies equivalence $\equiv$
- each refined initial state s* is coupled by the invariant to an abstract initial state $s \approx s^*$
- the forward simulation condition FSC holds for every pair (s,s*) of abstract and refined states

This theorem 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

Illustrating ASM Ground Model Construction and Provably Correct Refinements for asynchronous ASMs

Leader Election: a distributed network algorithm

See AsmBook Ch.6.1.5. (Springer-Verlag 2003)

Compare with Petri net formalization in

W.Reisig: Elements of Distributed Algorithms

Sect. 32 (Fig.32.1/2), 76 (Springer-Verlag 1998)

Compare with event-B development in:

J.-R.Abrial, D. Cansell, D. Mery: A mechanically proved and incremental development of IEEE 1394 tree identify protocol. Formal Aspects of Computing 14 (2003) 215-227

Defining Asynchronous Multi-Agent ASMs

- An **async ASM** is a family of pairs (a, ASM(a)) of
  - agents a $\in$ Agent (a possibly dynamic set)
  - basic ASMs ASM (a)
- A **run** of an async ASM is a partially ordered set (M, $<$) of "moves" m of its agents s.t.:
  - finite history: each move has only finitely many predecessors, i.e. {m' | m' $<$ m } is finite for each m$\in$ M
  - sequentiality of agents: for each agent a $\in$ Agent, his moves {m | m$\in$ M, a performs m} are linearly ordered by $<$
  - coherence: each (finite) initial segment X of (M, $<$) has an associated state σ (X) – think of it as the result of all moves in X with m executed before m' if m<m'– which for every maximal element m$\in$ X is the result of applying move m in state σ(X-{m})

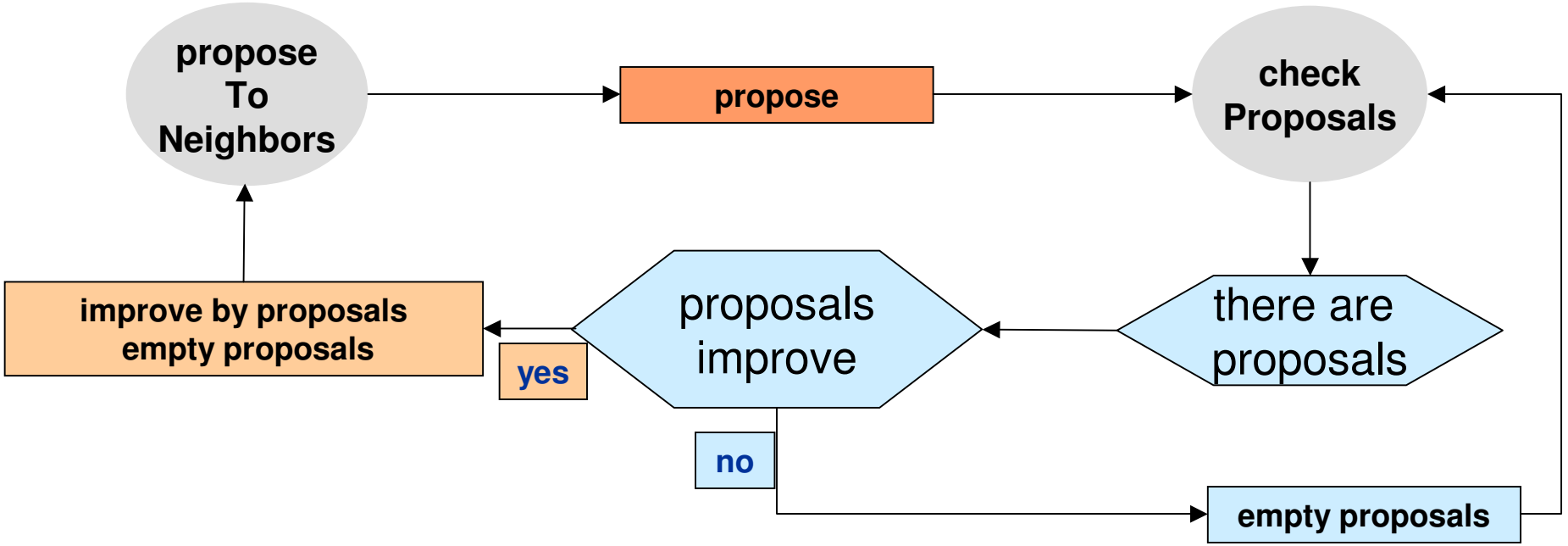
Leader Election: problem statement

- Goal: Design a distributed algorithm for the election of a leader in finite connected networks of homogeneous agents, using only communication (message passing) between neighbor nodes.
- Assumptions:
  - network nodes (agents) are connected & linearly ordered
  - leader = max (Agent) wrt the linear order <
- Algorithmic Idea: every agent
  - proposes to his neighbors his current leader candidate
  - checks the leader proposals received from his neighbors
    - upon detecting a proposal which improves his leader candidate he improves his candidate for his next proposal
- Eventually cand=max(Agent) holds for all agents

Leader Election: Agent Signature

- Agent: set of nodes of a finite connected graph
  - < linear order of Agent (external function)
    - leader = max (Agent) wrt the linear order <
- Each agent equipped with:
  - neighb ⊆ Agent (external function)
  - cand: Agent (controlled function)
  - proposals ⊆ Agent (controlled function)
  - ctl_state : {proposeToNeighbors, checkProposals}
- Initially ctl_state=proposeToNeighbors, cand=self, proposals = empty

Leader Election ASM Ground Model



propose ≡ forall n∈ self.neighb insert cand to n.proposals

proposals improve ≡ max (proposals) > cand

improve by proposals ≡ cand := max (proposals)

Leader Election: Correctness property

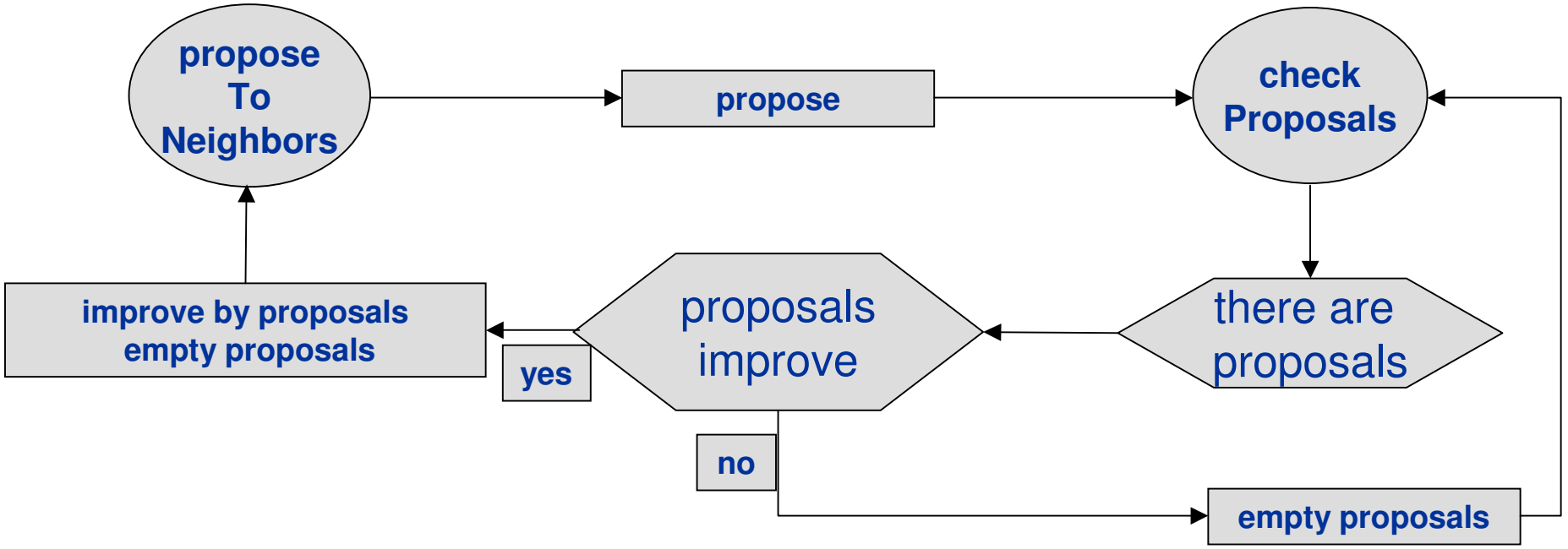
- Proposition: In every distributed run of agents equipped with the leader election ASM, eventually for every agent holds:
  - cand=max(Agent)
  - ctl_state = checkProposals
  - proposals = empty
- Proof (assuming that every enabled agent will eventually make a move): induction on runs and on $\Sigma\{\text{leader - cand}(n) \mid n \in \text{Agent}\}$
  - measuring “distances” of candidates from leader

Refining Leader Election: compute minimal path to leader

- Goal: refine the leader election algorithm to compute for each agent also a shortest path to the leader, providing
  - a neighbor (except for leader) which is closest to the leader
  - the minimal distance to the leader
- Idea: enrich cand and proposals by a neighbor with minimal distance to the leader candidate
  - nearNeighb: Agent
  - distance: Distance (e.g. = Nat $\cup\{\infty\}$)
  - proposals $\subseteq$ Agent x Agent x Distance
- Initially nearNeighbor = self distance = ∞ (0 for leader)

NB. This is a typical example of a pure data refinement

ASM Computing Minimal Path To Leader



propose ≡ forall n ∈ 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)

Max taken over agents

update PathInfo to m ≡
choose (n,d) with (m,n,d) ∈ proposals
d=minDistance(proposalsFor m) **in**
nearNeighb := n
distance := d+1

improve by proposals ≡
cand := Max (proposals)
update PathInfo to Max (proposals)

Minimal Path Computation: Correctness

- 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 (assuming that every enabled agent will eventually make a move): induction on runs and on $\Sigma\{\text{leader} - \text{cand}(n) \mid n \in \text{Agent}\}$ with side induction on the minimal distances in proposalsFor Max(proposals)

Exercises

- Refine the CHECK submachine of the leader election ASM by a machine which checks proposals elementwise. Prove that the refinement is correct.
  - Hint: See Reisig op.cit. Fig.32.1
- Adapt the ASM for the election of a maximal leader and for computing a minimal path wrt a partial order $\leq$ instead of a total order.
- Reuse the leader election ASM to define an algorithm which, given the leader, computes for each agent the distance (length of a shortest path) to the leader and a neighbor where to start a shortest path to the leader.
  - Hint: See Reisig op.cit. Fig.32.2

Relating Event-B Models to ASMs

References

J.-R.Abrial: Discrete System Models.
Manuscript, September 2004 (Version 2)
J.-R.Abrial: Event Driven Distributed Program Construction.
Manuscript, August 2004 (Version 6)

Event-B Models as ASMs: states, events, invariants

- States: structures of given signature with
  - static part ("context"):
    - sets s ("universes"), constants c, properties(c,s) ("axioms")
  - dynamic part: variables v and env (viewed as another model)
    - "Inputting is done by non-determinacy"
  - initialization (via a special event with guard true)
- Events of form If guard Then action
  - guard: closed fst-order set theory formula with =
  - action: one of the three forms
    - Updates
      - read: simultaneous substitution $v_1,\ldots,v_n := e_1,\ldots,e_n\ (v)$
    - skip
    - choose x with P(x,v) in Updates
      - where Updates = $v_1,\ldots,v_n := e_1,\ldots,e_n\ (x,v)$
- Invariant: property holding in every state which is reachable from initial state

Event-B Models as ASMs: rule normal form

- Normal form $Rule_1$ or … or $Rule_n$ with $Rule_i$ of form
  - if cond then choose x with P(x) in Updates
    - possible cases: P = true or Updates = ∅ (skip)
- NB. R or S = (choose X∈{R,S} in X)
  - no two events can occur simultaneously. Also distributed event-B models are based on this interleaving view.
  - splitting into events implies some non-deterministic scheduling of events with overlapping guards
  - no parallel update allowed for the same variable
  - no rules of form: forall x with P(x) do rule
  - external choose only on rules (interleaving model), no further nesting of choices allowed in Updates

Event-B Models as ASMs: Refinement Notion

- **only (1,n) refinements with n>0** satisfying:
  - in a refinement $F_1,\ldots, F_n,F$ of E, each F_i is supposed to refine SKIP
  - the new events F_i do not diverge
  - if the refinement deadlocks, then the abstraction deadlocks
- Variables in abstract & refined model are pairwise different
- no (1,0) refinement ("each abstract event must be refined by at least one refined event")
- no (n,m) refinement with $n \neq 1$
- **observables** = locations of interest ("projections of state variables"), here: variables which are
  - fresh (for state vars and for invariants)
  - modifiable only by "observer event" a:=A(v)
  - dependent only on state variables v
  - abstract observables A(v) can be "reconstructed" from refined ones by an equation A(v)=L(B(w)) ("invariant gluing the abstract observables to the refined ones")

References

E. Börger, R. Stärk

Abstract State Machines

A Method for High-Level System Design and Analysis

Springer-Verlag 2003. See http://www.di.unipi.it/AsmBook

R. Stärk, J. Schmid, E. Börger

Java and the Java Virtual Machine

Definition, Verification, Validation

Springer-Verlag 2001. See http://www.inf.ethz.ch/~jbook

For papers see http://www.di.unipi.it/~boerger