Egon Börger (Pisa)

The ASM Refinement Method

Sources:

- E.Börger: *The ASM Refinement Method*
  Formal Aspects of Computing 15 (2003) 237-257
- AsmBook Ch.3.2: Incremental Design by Refinements

and the references indicated below.

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

Refinement concept

Refinement is a general methodological principle:

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

ASM refinement notion exploits the arguably most general 'freedom of abstraction' (read: availability of arbitrary structures to directly reflect states and operations) offered by ASMs: it is *NOT limited by the principle of substitutivity* which drives traditional refinement notions:

> The intuition behind refinement is just the following: Principle of Substitutivity: it is acceptable to replace one program by another, provided it is impossible for a user of the programs to observe that the substitution has taken place. [Derrick&Boiten 2001, pg.47]

Why restriction to not-observable substitutions?

Motivation for the ASM refinement concept

ASM refinement notion is problem-oriented. Its goal is to:

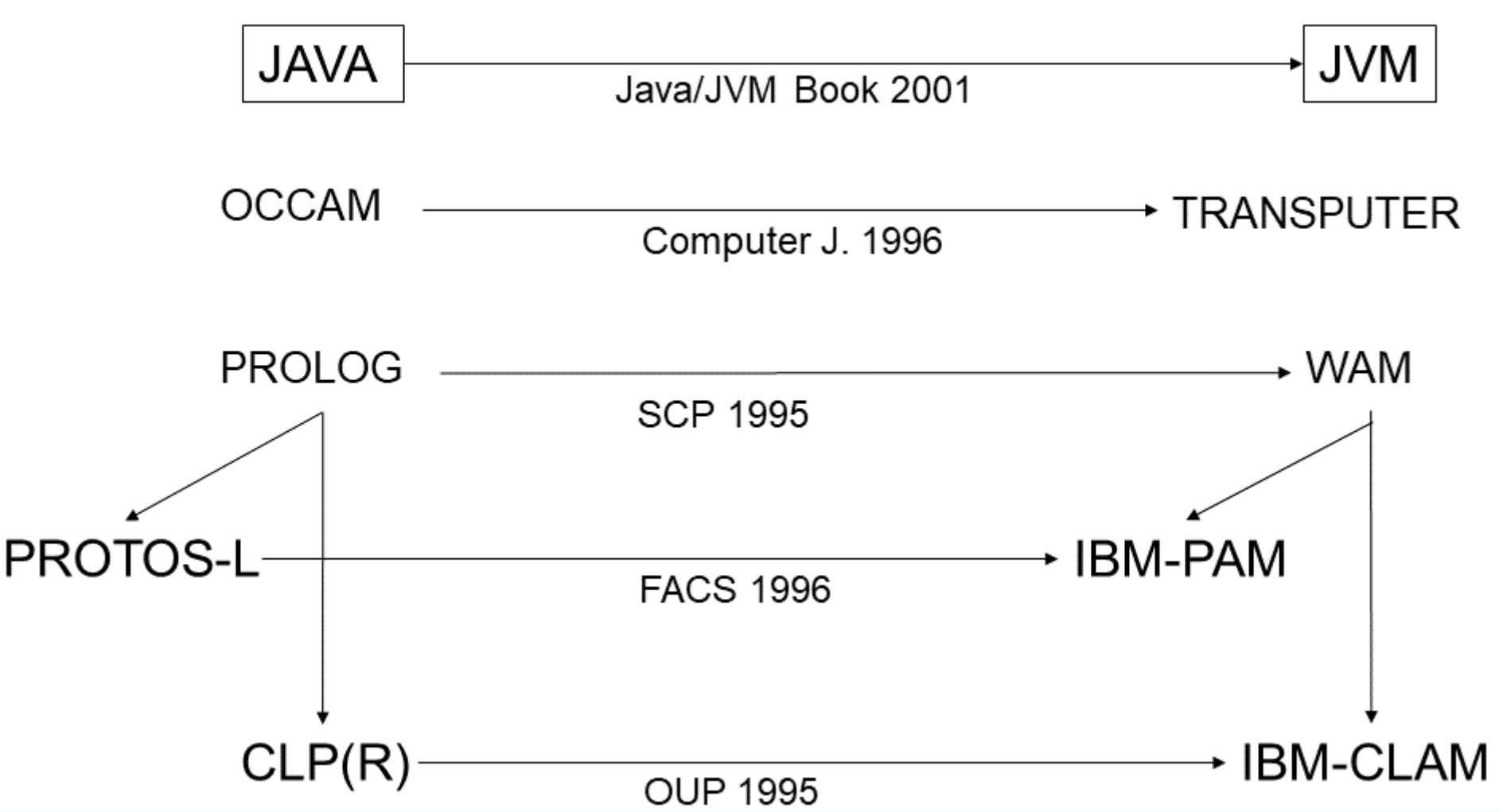
- support divide-and-conquer techniques for system *design and analysis*
  - without privileging one to the detriment of the other
    - traditional approaches link design and verification (sic) by relating program constructs and proof principles at the price of restricting the design space (ADT, Z, CSP, B,...)
- allow the designer to tailor refinement/abstraction pairs which
  - faithfully *reflect a design decision* or reengineering idea
  - provide means to *justify an implementation* or abstraction as 'correct' (via verification or simulation)
  - support design *communication*, design *reuse* and system *maintenance* thru accurate, precise, indexed and searchable docu

NB. Development of the ASM refinement method was driven by practical refinement tasks occurring in real-life system development

Main usages of ASM refinements: early examples

- capture orthogonalities by modular (maintainable) components
  - e.g. Production Cell (model checked), Steam Boiler (refined to C++ code), pipelining RISC DLX and APE architecture
- 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 (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) for system properties
  - e.g. reusing Prolog-to-WAM compiler correctness proof for IBM's CLP(R)-to-CLAM, Protos-L-to-PAM, etc.

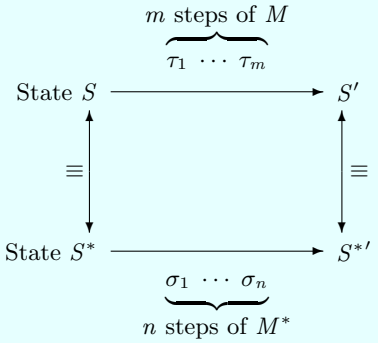
Early examples for reuse of ASM refinement hierarchies



NB. Both models and proofs could be reused (see references below).

Defining ASM refinement: 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



ASM refinement scheme a meta-framework for refinements

ASM refinement scheme[1] combines change of signature (correspondence of states/locations and equivalence of data) and change of control (flow of operations in corresponding computation segments), thus *generalizing*

- pure $(1,1)$ data refinements (e.g. VDM, Z, B)
- opn $(1,n)$-refinements with fixed n (e.g. Z, Object-Z, B)
- purely sequential or structural-equivalence view of pgs with corresponding ops in same places (ADT,Z)
- simple input/output or pre-/post-state view ('observations') of pgs

and *avoiding*

- pure logic or proof-rule oriented view, tailored to fit proof principles (which typically restrict design space)
  – see refinement book by de Roever&Engelhardt
- global view of ops (avoids frame pblm when combining local effects)

[1] Above figure from AsmBook, © 2003 Springer-Verlag Berlin Heidelberg, reused with permission

Syntactical orientation in traditional refinement notions

- logic or proof-rule orientation, tailored to fit proof principles
  - spec perceived as a (huge!) logical expression
  - implementation understood as implication
  - composition defined as conjunction
- possibly restricts the design space
  - e.g. refinements should be pre-congruences: for every context C: $x \leq y$ implies $C[x] \leq C[y]$. This can be achieved for example by monotonicity of pgm constructors wrt refinement.
    - commits to uniform context-independent 'algebraic' refinements
  - e.g. operation refinement by combining multiple operations 'conjunctively' or 'disjunctively' ('alphabet translation')

Linking program constructs and proof principles in B

Exl: machine inclusion concept (B-Book pg.317)

- Let M include M'. Then 'at most one operation of the included machine can be called from within an operation of the including machine. Otherwise we could break the invariant of the included machine.'
- Let M' have the following operations, satisfying the invariant $v \leq w$:
  - increment $=$ (If $v < w$ then $v := v + 1$)
  - decrement $=$ (If $v < w$ then $w := w - 1$)
- Let M include M' and contain the following operation:
  - If $v < w$ then increment $||$ decrement
- Then the invariant $v \leq w$ is broken by M for $w = v + 1$

NB. The ASM method allows parallel invocations of submachines (at the price of having to care about the correctness proofs)

Linking program constructs and proof principles in CSP

Refining processes by adding assignment is restricted to certain assignments

- Hoare CSP Book 1985, pg. 188

Exl. When two processes P and Q are put into parallel, it is required that the variables P assigns to are disjoint from the variables of Q: $Write(P) \cap Var(Q) = \emptyset$

- Otherwise the CSP laws would not work (Sic)

Correctness/Completeness of ASM refinements

Fix any notions $\equiv$ of equivalence, initial and final (if any) states.

M^* is a correct refinement of M ('partial correctness') iff for each M^*-run $S_0^*, S_1^*, \ldots$ there are an M-run $S_0, S_1, \ldots$, index sequences $i_0 < i_1 < \ldots, j_0 < j_1 < \ldots$ (of corresponding states of interest) s.t.

- initial states are corresponding states of interest: $i_0 = j_0 = 0$
- equivalence of corresponding states of interest: $S_{i_k} \equiv S_{j_k}^*$ for each k
- either both runs terminate and their final states are the last pair of corresponding states of interest, or
- both runs and both sequences $i_0 < i_1 < \ldots$, $j_0 < j_1 < \ldots$ are infinite

NB. The M^*-run $S_0^*, S_1^*, \ldots$ is said to simulate the M-run $S_0, S_1, \ldots$. Wlog bw two sequence elements there are no other equivalent states.

M^* is called a complete refinement of M ('total correctness') iff M is a correct refinement of M^*.

Rational of stepwise refinement for proving system properties

Overall task (0): show that an implementation S^* satisfies a desired property P^*.

For a complex system this task may be impossible to be tackled at a single blow.

Decompose (0) into 3 subtasks which usually are more manageable:

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^*.

Remarks on some ASM refinement types

- local data equivalence often accumulates to a notion of *equivalence of corresponding states of interest*
- refinement correctness and completeness for terminating runs imply the equivalence of any input/output behavior (called *bisimulation* or *interpreter equivalence*)
- size of m and n in (m, n)-refinements is allowed to dynamically depend on the state
  - in Java exception handling correctness proof, m is determined by the number of Java statements jumped over during the search for the exception handler
- by a theorem of Schellhorn, every (m, n)-refinement with $n > 1$ can be reduced to $(m, 1)$-refinements—typically at the price of having more involved equivalence notions which typically complicate the proofs

Conservative ASM refinements

Conservative extension: purely incremental refinement, analogous to conservative theory extensions in logic.

Typically used to introduce new behavior in a modular fashion.

$\textsc{ConservativeExtension}(M, NewCase, \textsc{NewMachine}) =$

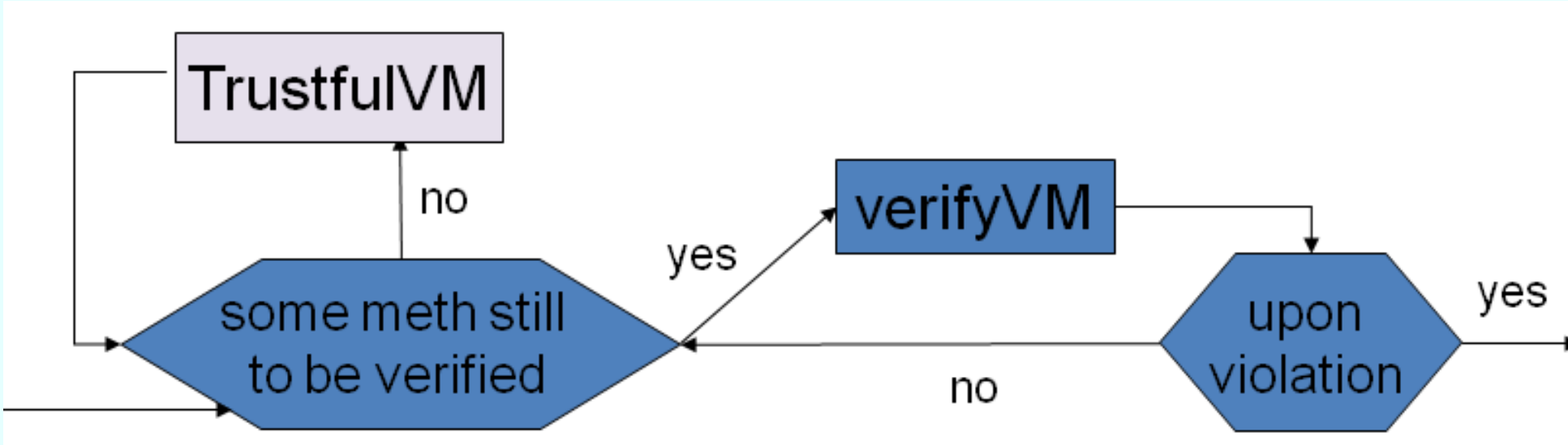
if $NewCase$ **then** $\textsc{NewMachine}$ // additional new behavior

else M // given machine with 'old' behavior

NB. Conservativity simplifies both design and correctness proofs.

Conservative ASM refinements: JBook example

- refinement of Java interpreter by a proven to be correct exception handling mechanism
- incorporation of a bytecode verifier into the JVM interpreter



Procedural ASM (Submachine) refinements

$(1, n)$-refinements replacing in a given machine M one submachine P by another (usually more complex) machine Q.

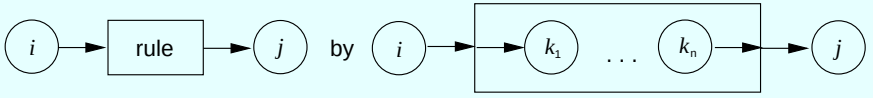
$\textsc{ProceduralExtension}(M, P, Q) = M[P/Q]$

NB. Substitution does not imply the principle of substitutivity:

- depending on the granularity of the submachine, a refined machine typically provides new details or features one wants to see
- even for $n = 1$ Q may be a parallel composition of multiple machines

Examples of ASM submachine refinements

- $(1, n)$-refinements with $n > 1$ in *compiler verification* when replacing a source code instruction by a chunk of target code (e.g. $unify$ refinement in Prolog-to-WAM with a priori unbounded n)
- refinement of P by a *turbo ASM* (built from basic ASMs using **seq** and **iterate**)
- refining a $rule$ in control-state transition $\mathrm{FSM}(i, rule, j)$ by a control-state machine diagram M to $\mathrm{FSM}(i, M, j)$[2]
- refining an atomic action by multiple actions which are executed in an asynchronous manner (exl: refinement of Occam process communication by an asynchronous channel communication)



[2] Figure from AsmBook, © 2003 Springer-Verlag Berlin Heidelberg, reused with permission

Data refinement

- typically $(1, 1)$-refinements where abstract states/rules are mapped to concrete ones s.t. the effect of an abstract/concrete operation on abstract/concrete data types is the same
- basis of numerous algebraic and set-theoretic refinement notions (e.g. VDM, Z, B), tailored to provide 'unchanged' properties for corresponding operations with 'unchanged' signature
- frequently used exl: ASM instantiation where ASM rules remain unchanged but abstract (mostly external) auxiliary functions/ predicates occurring in them are specified further
  - particularly useful for transition from a use case model with abstract (symbolic) rules to a model which assigns a state transformation meaning to the rule names

Exl: BACKTRACK scheme for tree generation/traversal

$$\text{BACKTRACK} = \text{RAMIFY}\ \textbf{par}\ \text{SELECT}\ \textbf{where}$$

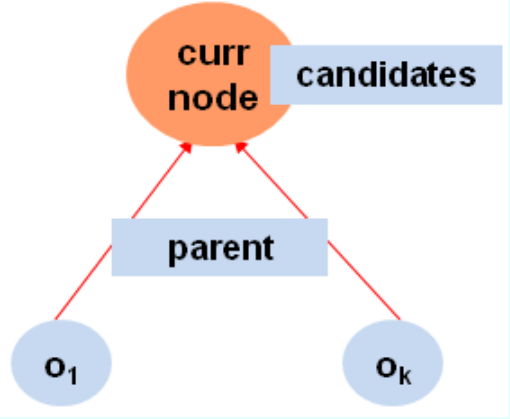
$$\text{RAMIFY} = (\textbf{if}\ mode = ramify)\ \textbf{then}$$

$$\textbf{let}\ k = |alternatives(Params)| \quad o_1, \ldots, o_k = new(NODE)$$

$$candidates(currnode) := \{o_1, \ldots, o_k\}$$

$$\textbf{forall}\ 1 \leq i \leq k \quad \begin{array}{l} parent(o_i) := currnode \\ env(o_i) := i\text{-th}(alternatives(Params)) \end{array}$$

$$mode := select$$



BACKTRACK scheme for tree generation/traversal (Cont'd)

SELECT =

if $mode = select$ **then**

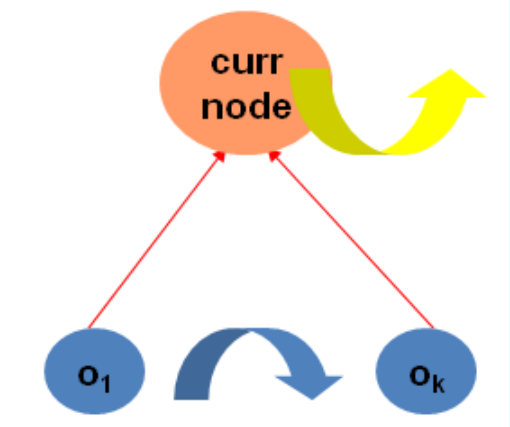
if $candidates(currnode) = \emptyset$

then BACK

else

TRYNEXTCANDIDATE

$mode := execute$



BACKTRACK scheme for tree generation/traversal (Cont'd)

$\textsc{Back} =$

if $parent(currnode = root)$

then $mode := Stop$

else $currnode := parent(currnode)$

$\textsc{TryNextCandidate} =$

$currnode := next(candidates(currnode))$

$\textsc{Delete}(next(candidates(currnode)), candidates(currnode))$

NB. Scheme data refinable by instantiating its external fcts:

- $alternatives$ determines the solution space
- $next$ determines the order for trying out the alternatives

Data refined BACKTRACK in logic programming

- backtrack engine for *ISO Prolog*: exec engine switches from $ramify$ to $select$
  - $alternatives = procdef\,(stm, pgm)$
    - yields the sequence of clauses in pgm to be tried out in this order to execute the current goal stm
  - $next = head$
    - reflects depth-first left-to-right tree traversal strategy of ISO Prolog
- *constraint logic programming* $CLP(R)$: dto with CLP(R) exec engine extending $procdef$ by param for current $constraint$ set
- *functional logic language* Babel: dto with Babel exec engine and $alternatives = fundef\,(currexp, pgm)$
  - $fundef$ yields the list of defining rules provided in pgm for the outer function of $currexp$ (to reduce it to normal form by narrowing)

Data refined BACKTRACK in context-free grammars

- *context-free grammar*: generation of leftmost derivations
  - $alternatives(currnode, G)$ yields conclusion of a G-rule $X \to Y_1, \ldots, Y_k$ where X labels $currnode$
    - env records node label (variable X or terminal a), $next = fst$

EXECUTE(G) = **if** $(mode = execute)$ **then**

if $env(currnode) \in VAR$ // tree expansion

then $mode := ramify$ **else**

$output := output * env(currnode)$ // extract yield

$currnode := parent(currnode)$ // continue derivation

$mode := select$

NB. further data refinements capture *attribute grammars* and *tree adjoining grammars*, generalizing Parikh's analysis of context free languages by 'pumping' of context free trees

Mealy automata refinement of TuringLikeMachine

A TuringLikeMachine is a set of rules of form:

$\textsc{Fsm}(i, \textbf{if } Cond \textbf{ then } \textsc{Action}, j)$

$= \textbf{if } mode = i \textbf{ and } Cond \textbf{ then } \textsc{Action} \textbf{ par } mode := j$

where // parameterization by functions mem, env, pos

$Cond = Cond(mem(env(pos)))$

$\textsc{Action} = \textsc{Action}(mem(env(pos)), pos)$

Instantiations of mem, env, pos and of $Cond, Update$ yield:

- Mealy automaton: $mem = input$ function, $env = identity$
  $Cond$ **iff** $mem(pos) = a$ // reading head on $input$ tape position
  $\textsc{Action} = pos := pos + 1$ // shift input reading head to the right
- 2-way Mealy automaton: dto with $\textsc{Action} = pos := pos + move$
- Eilenberg's X-machine: dto, adding to $\textsc{Action}$ updates $currstate := f(currstate)$ for a global state function $currstate$

FSM and Pushdown refinements of TuringLikeMachine

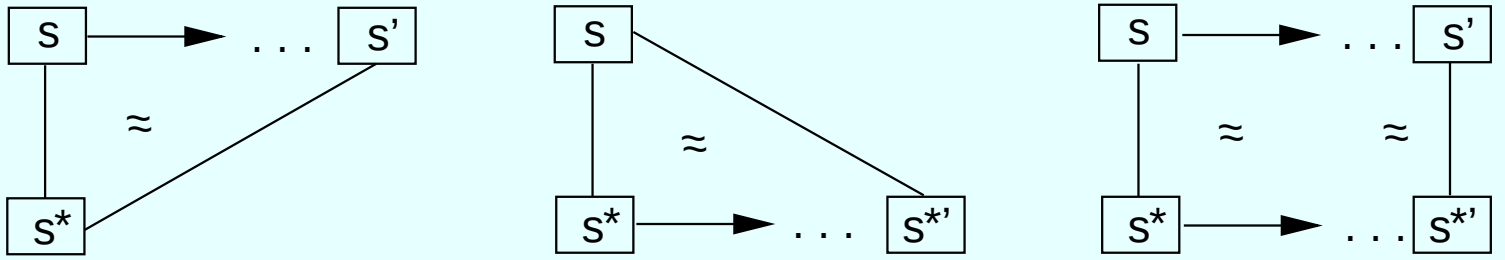
- Moore automaton: refine Mealy by additional $output$ function and additional updates $output := b$ in Action
  - If $output$ is viewed as tape, add shifting the writing head position $pos_{output} := pos_{output} + 1$ to Action
- Pushdown automaton: refine Mealy by adding the $stack$ function with
  $Cond$ **iff** $[in = a] \wedge [top(stack) = b]$ // [ ] optional
  Action $= stack := push(w, [pop](stack))$
- Turing machine: refine 2-way Mealy with $mem = tape$ and additional update $tape(pos) := b$ in Action
- Interactive TM (Wegner): refine TM by additional param $input$ for all fcts
- refine input/output from letters a, b to words, streams, trees, etc.

Schellhorn's refinement correctness pf modularization

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

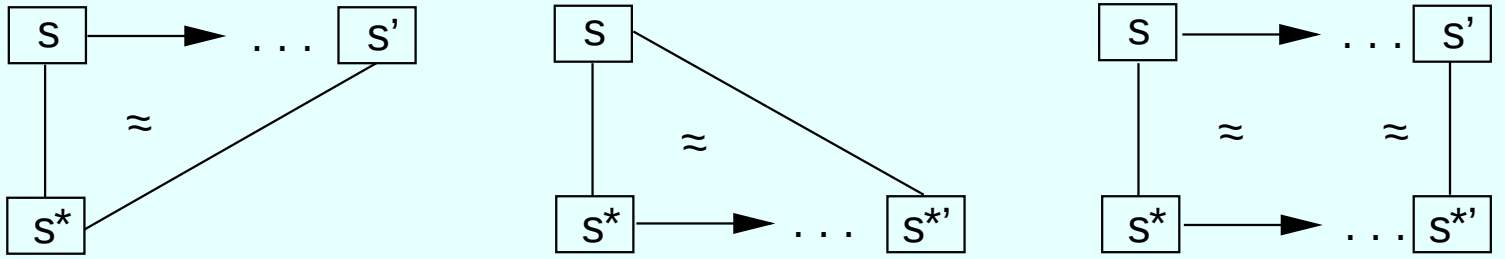
Method: follow the two runs, for each pair of corresponding states—not both final—satisfying $\approx$, looking for a successor pair $s', s^{*\prime}$ (of corresponding states satisfying $\approx$).

Run extension may be possible for both or only for one run: *3 diagram types*[3]



[3] Figure from AsmBook, © 2003 Springer-Verlag Berlin Heidelberg, reused with permission

Schellhorn's basic diagram types



- $(m, 0)$-triangles: computation segments where only the abstract run makes progress performing a positive number m of steps to reach an $s' \approx s^*$,
- $(0, n)$-triangles: computation segments where only the concrete run makes progress performing a positive number n of steps to reach an $s^{*\prime} \approx s$,
- (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 $s' \approx s^{*\prime}$. Any of the three possible subcases $m < n$, $m > n$ (typical for optimizations) or $m = n$ is allowed here.

Defining the 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 $s' \approx s^{*\prime}$.

Decomposition Thm for 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.

NB. Thm proved by Schellhorn (1997) using KIV as part of the KIV-verification of the Prolog-to-Wam refinement hierarchy

Further examples of ASM refinements

See lectures on ASM refinement examples, section AdditionalMaterial:

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

References to ASM refinement concept

E. Börger: The ASM Refinement Method.
Formal Aspects of Computing J. 15 (2003), 237-257

E. Börger and R. F. Stärk: Abstract State Machines.
Springer 2003. See *http://www.di.unipi.it/AsmBook/*

R. Stärk, J. Schmid, E. Börger: Java and the Java Virtual Machine.
Definition, Verification, Validation.
Springer 2001. See *http://www.di.unipi.it/ boerger/jbook*

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

References to first ASM refinements

E. Börger and D. Rosenzweig: A Mathematical Definition of Full Prolog. Science of Computer Programming 24 (1995) 249–286

E. Börger and D. Rosenzweig: The WAM – Definition and Compiler Correctness. In: C. Beierle and L. Plümer (eds): Logic Programming: Formal Methods and Practical Applications. Studies in Computer Science and Artificial Intelligence 11 (1995) 20-90

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

References to early ASM refinement applications

E. Börger (Ed.): Specification and Validation Methods. Oxford University Press 1995

D. E. Johnson and L. S. Moss: Grammar Formalisms Viewed As Evolving Algebras. Linguistics and Philosophy 17 (1994) 537–560

C. Beierle and E. Börger: Specification and correctness proof of a WAM extension with abstract type constraints. Formal Aspects of Computing 8.4 (1996) 428–462

C. Beierle and E. Börger: Refinement of a typed WAM extension by polymorphic order-sorted types.Formal Aspects of Computing 8.5 (1996) 539–564

References to Schellhorn's work on ASM refinement concept

- G. Schellhorn: Verification of ASM Refinements Using Generalized Forward Simulation. JUCS 7.11 (2001) 952–979
- G. Schellhorn: ASM Refinement and Generalizations of Forward Simulation in Data Refinement: A Comparison. TCS 336/2-3 (2005) 403-436
- G. Schellhorn: Completeness of ASM Refinement. Electr. Notes TCS 214 (2008) 25-49
- G. Schellhorn: ASM Refinement Preserving Invariants. J. UCS 14.12 (2008) 1929-1948
- G. Schellhorn: Completeness of fair ASM refinement. SCP 76.9 (2011) 756-773

References to books on traditional refinement concepts

J. Derrick, E. Boiten: Refinement in Z and Object-Z. Springer 2001

W. de Roever, K. Engelhardt Data Refinement: Model-Oriented Proof Methods and their Comparison. Cambridge University Press 1998

J. C. P. Woodcock, J. Davies Using Z: Specification, Refinement, and Proof. Prentice-Hall 1996

R. J. R. Back, J. von Wright Refinement Calculus: A Systematic Introduction. Springer 1998

J.-R. Abrial: The B-Book. Cambridge University Press 1996

C. A. R. Hoare: Communicating Sequential Processes. Prentice-Hall 1985

Copyright Notice