Refinement Method for Abstract State Machines

Egon Börger

Dipartimento di Informatica, Universita di Pisa
http://www.di.unipi.it/~boerger

For details see Chapter 3.2 (Incremental Design by Refinements) of:

E. Börger, R. Stärk

Abstract State Machines

A Method for High-Level System Design and Analysis

Springer-Verlag 2003

For update info see AsmBook web page:

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

"The intuition behind refinement"

- "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 should "acceptable" refinements be restricted to those which guarantee that the substitution of one program by a refined one is not observable?
  - e.g. imagine one wants to
    - observe the desired improvement provided by a refinement (an executable instead of an abstract pgm, a faster or more general pgm serving also other purposes, a strengthening…)
    - delimit the exact boundaries within which the refined program performs in the intended way

Characteristics of refinement notions in the literature

- Traditionally, refinement notions guided by the substitutivity principle come with additional restrictive assumptions:
  - programs describe sequences of operations
    - precluding parallelism of multiple simultaneous updates or iterative compositions of programs
  - operations are global (binary) state relations
    - yielding the frame problem for combinations of local effects
  - observations are pairs of input/output sequences or of pre-post-states representing what is considered to be of interest before/after program execution
    - making it difficult to look at arbitrary segments of computation

Role of syntactical issues in refinement notions in the literature

- numerous program refinement notions (e.g. for ADT, Z) are formulated for structurally equivalent programs with corresponding operations in the same places
  - precluding the analysis of more complex relations bw operations
- invariants in refinements are often viewed as changing the state scheme or the operations, in terms of pre/post condition strengthenings or weakenings
  - instead of analysing their effect as restricting the class of models

Role of syntactical issues in refinement notions in the literature

- most refinement notions are logic or proof-rule oriented, tailored to fit proof principles [de Roever&Engelhardt]
  - spec perceived as a (huge!) logical expression
  - implementation understood as implication
  - composition defined as conjunction
- thus possibly restricting 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 refinement and proof principles illustrated by B

- B links design & proofs by relating pgm constructs & proof principles **at the price of restricting the design space**
- Machine inclusion example (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 $w = v+1$
- The ASM method allows parallel invocations of submachines
  - at the price of having to care about the correctness proofs

Linking refinement and proof principles illustrated by CSP

- CSP links design & proofs by relating pgm constructs & proof principles **at the price of restricting the design space**
- Refining processes by adding assignment is restricted to certain assignments (Hoare CSP Book 1985, pg. 188)
- 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

Introducing refinement techniques into ASMs

- Refinements, one of the 3 building blocks of the ASM method, were introduced into ASMs in 1989 through Börger's ASM models defining the ISO Prolog standard, triggered by the simple observation that

  exploiting the freedom of abstraction ASMs offer, one can tailor ASM refinements to solve given design & analysis problems also for complex real-life systems as they occur in industrial practice

- Consequently, the ASM refinement method is problem-oriented and its development was driven by
  - practical refinement tasks, occurring in real-life system development
  - the goal to support divide-and-conquer techniques for both design and verification without privileging one to the detriment of the other

See E. Börger: The Origins and the Development of the ASM Method for High Level System Design and Analysis. JUCS 8 (1) 2001

Problem oriented tasks guiding the ASM refinement method

- In each case, "listen to the subject" to find/formulate an appropriate refinement /abstraction that
  - faithfully reflects the intended design decision (or reengineering idea) for the system under study
  - can be justified to correctly implement the given model (or to abstract from the given code), namely through
    - **verification**
    - **validation** testing model-based runtime assertions to show by simulation that design assumptions hold in the implementation
- Effect (scaling to industrial-size systems): enhancement of
  - communication of designs and system documentation (report of analysis)
  - effective reuse (exploiting orthogonalities, hierarchical layers)
  - system maintenance based upon accurate, precise, richly indexed & easily searchable documentation

E.Börger: High Level System Design and Analysis using ASMs
LNCS 1012 (1999) 1-43

Main usages of ASM refinements

- capture orthogonalities by modular machines (components)
  - e.g. ASMs for sublanguages of Java and JVM instructions
- 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), object-orientation (Müller), parallelism (Parlog, Concurrent Prolog etc), abstract execution strategy (Gödel)
  - vertical stepwise detailing of models (design for reuse) in a proven to be correct way down to their implementation, e.g. model chains leading from
    - Prolog to WAM
    - Occam to Transputer
    - Java to JVM
    - ASMs to executable ASMs (Workbench, AsmGofer, AsmL, XASM)
- exploit reusable proof techniques for system properties
    - e.g. reusing Prolog to WAM proof for
      - CLP(R) to CLAM
      - Protos-L to PAM
    - using variety of logics for ASMs, KIV, PVS, Isabelle, model checkers

Examples of ASM Refinement & Verification Hierarchies

Architectures: Pipelining of RISC DLX: model checking, PVS verification

Control Systems: Production Cell (model checked), Steam Boiler (refinements to C++ code) Light Control (executable requirements model)

Compiler correctness

ISO Prolog to WAM: 12 refinement steps, KIV verified

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

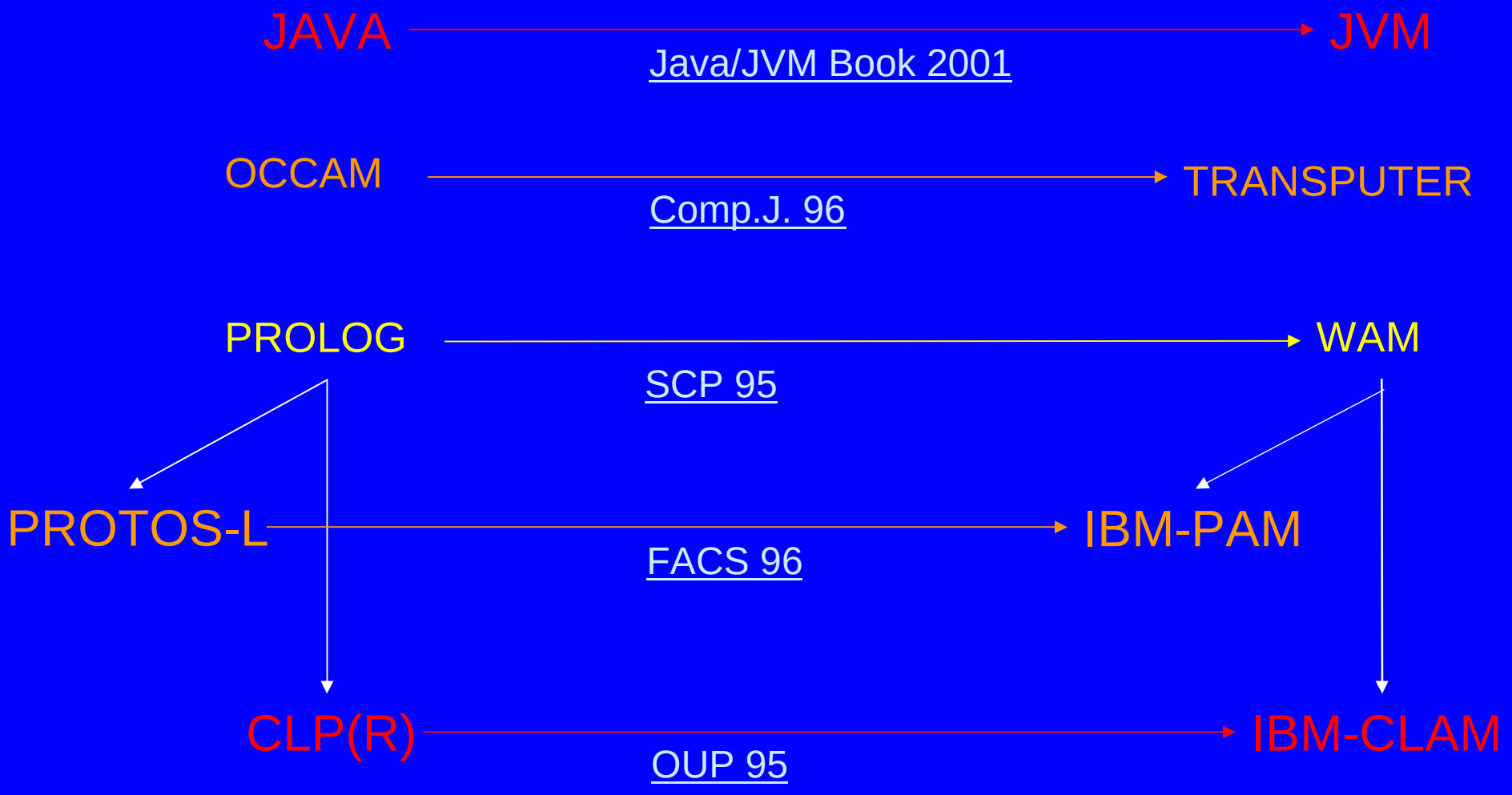
Occam to Transputer :15 models exhibiting channels, sequentialization of parallel procedures, pgm ctrl structure, env, transputer datapath and workspace, relocatable code (relative instr addresses & resolving labels)

Java to JVM: language and security driven decomposition into

5 horizontal sublanguage levels (imperative, modules, oo, exceptions, concurrency) and

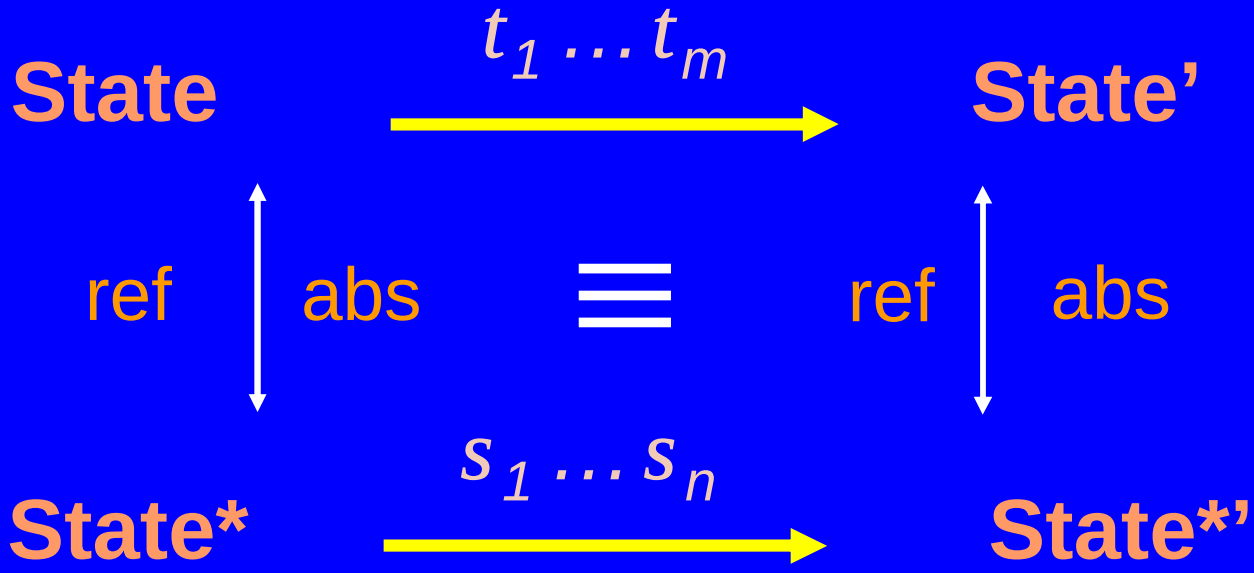
4 vertical JVM levels for trustful execution, checking defensively at run time and diligently at link time, loading (modular compositional structuring)

Illustrating Reusability of ASM Refinement Hierarchies



Reuse of submachines (layered components) and of lemmas

The ASM Refinement Scheme: Commuting Diagrams



with an equivalence notion ≡ definable to relate

- the locations of interest ("corresponding locations")
- in states of interest ("corresponding states")
- reached by (m,n) computation segments of interest

combining **change of signature** (data in locations) **& of control** (flow of operations), generalizing data refinements, (1,n)-refinements, I/O automata refinements (by forward or backward simulations), etc.

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 (correspond to) abstract ones
- Def. M* is a complete refinement of M
  iff M is a correct refinement of M*
- Related terminology:
  - "bisimulation" or "interpreter equivalence" for correct and complete refinement (wrt terminating runs considering only the input/output behavior)
  - "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)

Remarks on the correctness conditions for ASM refinements

- Corollary. Refinement correctness implies for terminating runs the equivalence of the input/output behavior of the abstract and the refined machine.
- $S(i_k)$, $S^*(j_k)$ are the corresponding states (those of interest), end points of the corresponding computation segments (those of interest), for which the equivalence is defined in terms of a relation between their corresponding locations (those of interest).
- Wlog the sequences of corresponding states are minimal in the sense that between two sequence elements there are no other equivalent states
  - i.e. there are no $i_k < i < i_{k+1}$, $j_k < j < i_{k+1}$ with $S(i) \equiv S^*(j)$

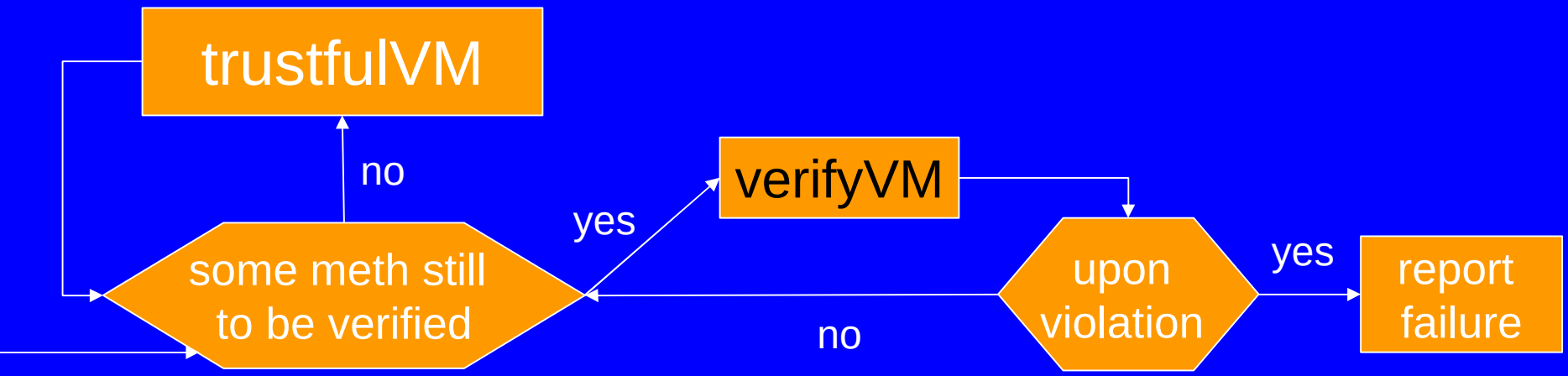
Refinement notions in the literature as cases of ASM refinements

- Considering only the input/output behavior, restricting correctness (essentially) to terminating runs
  - e.g. preservation of partial/total correctness (as used in compiler correctness verifications) or bisimulation
- Data refinement considering as initial/final the pre/post states of an operation
  - (1,1)-refinements for corresponding operations (with unchanged signature, tailored to provide "unchanged" properties)
    - forward simulation carries over ≡ from pre-states to post-states
    - backward simulation carries over ≡ from post-states to pre-states
    - see Hoare 1972, VDM, Z, B, de Roever & Engelhardt 1998
  - NB. Under a monolithic view (of each ASM as defining just one total operation on structures), ASM refinement becomes data refinement
- Non-atomic operation refinement
  - (1,n)-refinements with fixed n (in Z, Object-Z, see Derrick & Boiten 2001)
  - (1,1)-refinements for external operations with (1,0),(0,1)-refinements for finitely many invisible internal operations
  - alphabet extension/translation, I/O automata refinements, etc. see details in [Schellhorn2001]

Conservative ASM refinement: incrementally adding machines

- Adding an entire machine M - not limited to a single "operation" - to another machine

  Exl. Adding a bytecode verifier to the Java interpreter in JVM



- verifyVM itself is defined from submachines check, propagateVM, succ by a parallel ASM refinement which follows the language extensions for the JVM

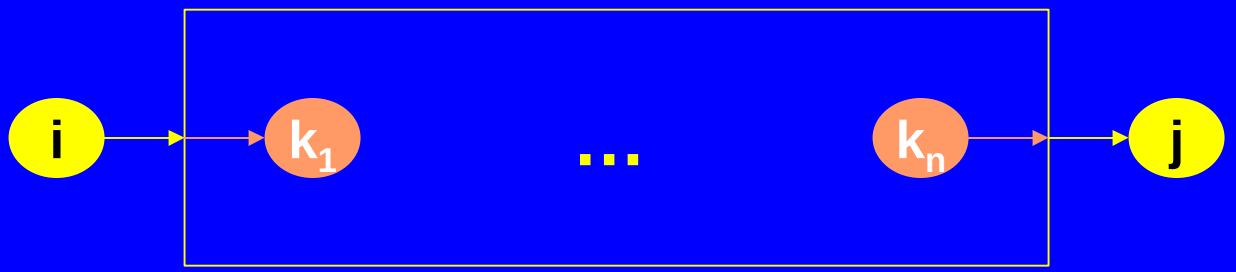
Procedural refinements & their specialization to sequential submachine refinements of ctl state ASMs

Procedural refinement: replacing a machine by another (usually more complex) machine

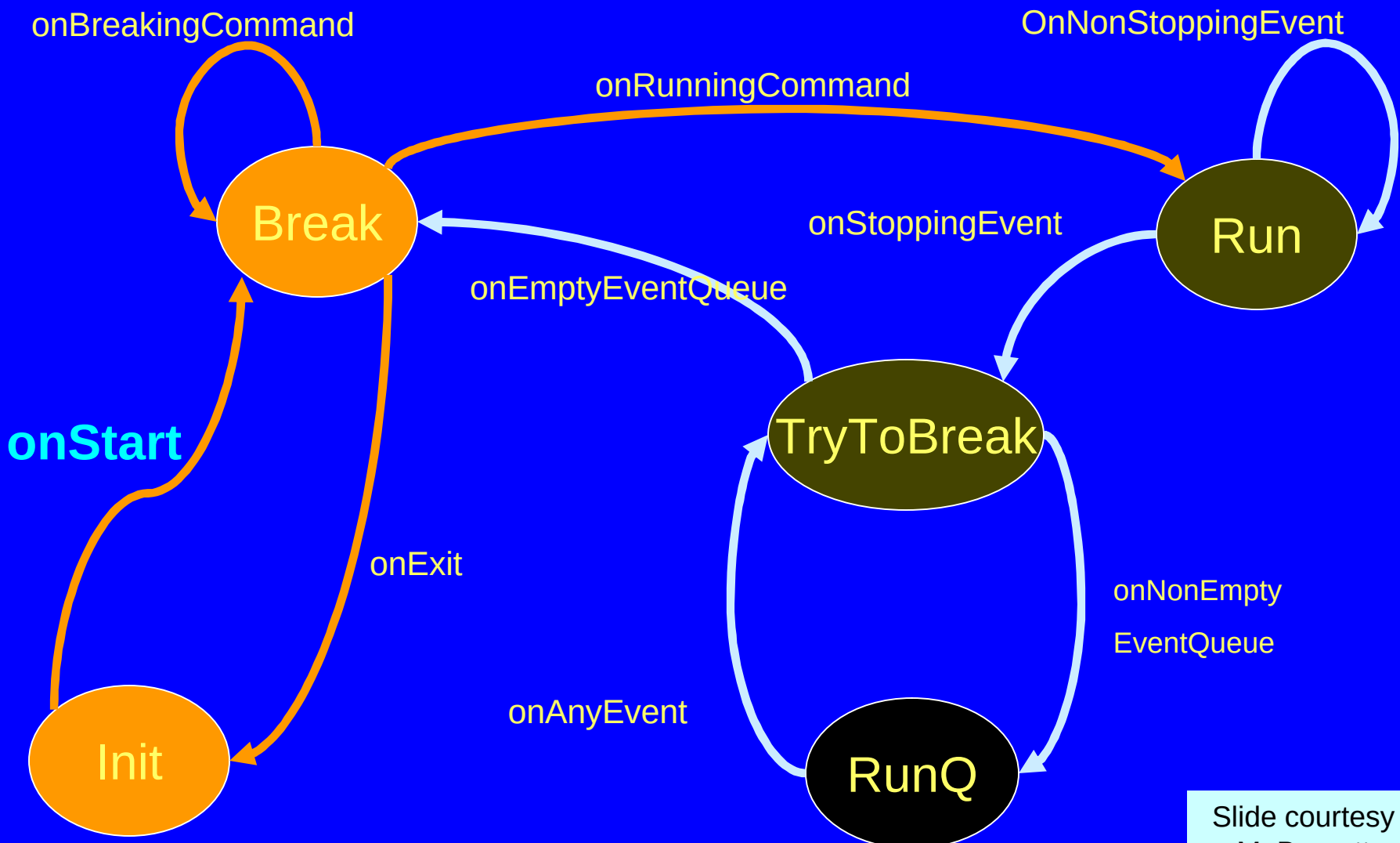
Specialization for control state ASMs: replacing control state transitions (machines at nodes) by submachine diagrams with entry/exit nodes

The Scheme:

⇒



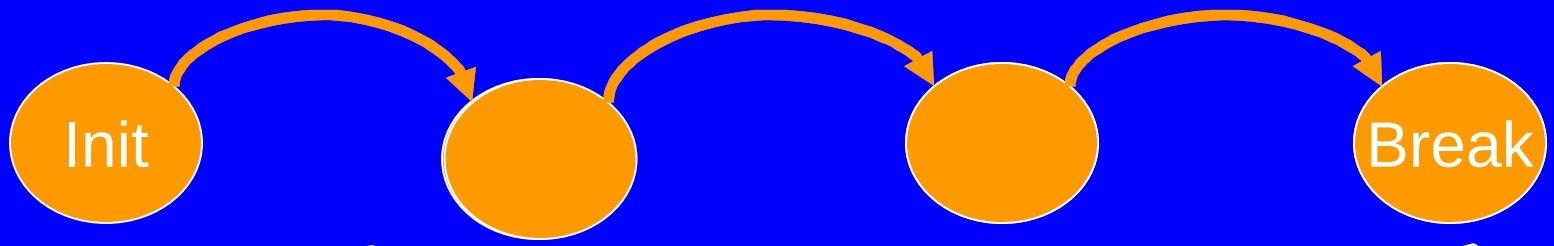
Illustrating sequential submachine refinements refining the control state ASM model for a debugger

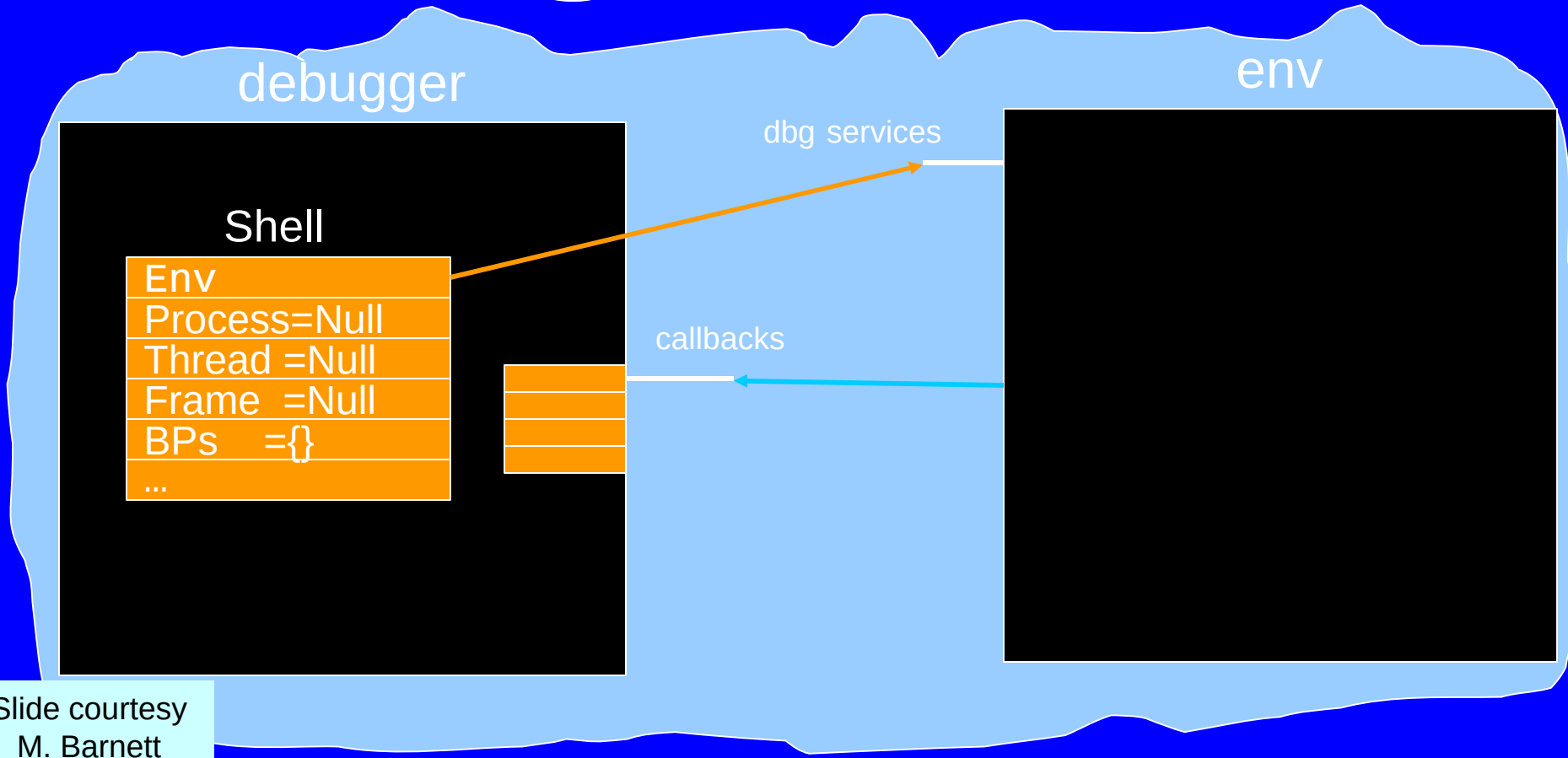


Slide courtesy
M. Barnett
M. Veanes

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

`initializeCOM` `createNewShell` `setDbgCallback`



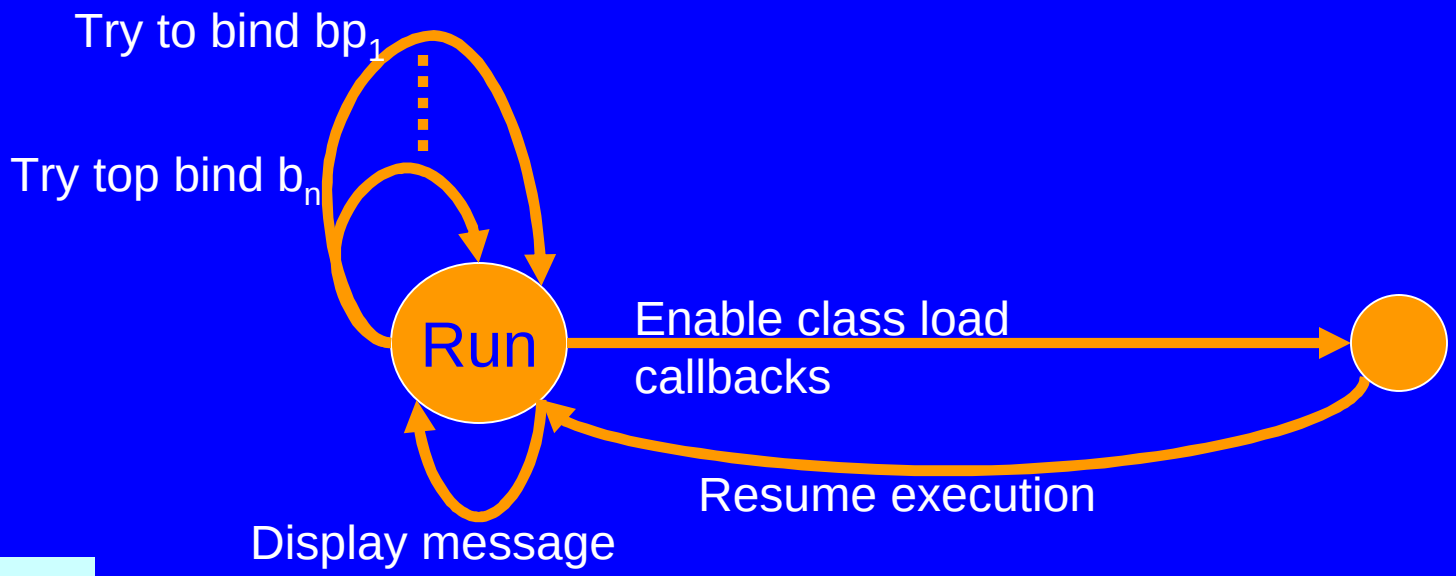


Slide courtesy
M. Barnett
M. Veanes

Parallel and sequential refinement of callback(LoadModule)

```
callback(LoadModule (proc,mod)) =
    displayMessage("Loaded module: " ++ mod.name())      record mod in shell
    forall bp in shell.BPs                  bind all breakpoints to the mod (in any order)
      bp.bind(mod)
    seq
        mod.enableClassLoadCallbacks()
        proc.resume()                                    continue via external call
```

Analogously for UnloadModule



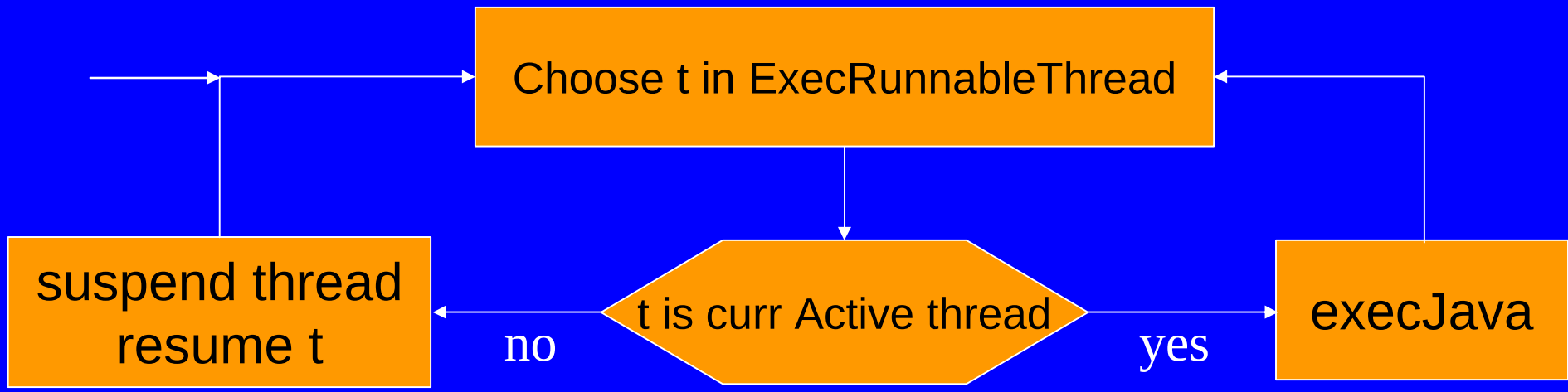
Slide courtesy
M. Barnett
M. Veanes

(1,1)-refinements of ASMs allow parallelism

- replacing an action - part of a parallel step, not limited to a single "operation" - by multiple parallel actions (not viewed as a new "operation", but as part of a new parallel step) e. g. rule by $rule_1$ … $rule_n$

Exl. Defining submachine execJava of **execJavaThread** by parallel submachines

separating semantics of thread execution from thread scheduling



(1,1)-refinement of execJava as parallel composition of language driven submachines

execJava =

execJava$_I$ — **imperative control constructs**

execJava$_O$ — **oo features**

execJava$_E$ — **exception handling**

execJava$_T$ — **concurrent threads**

where each **execJava**$_{sub}$ =

execJavaExp$_{sub}$ — **expression evaluation**

execJavaStm$_{sub}$ — **statement execution**

allowing semantics to be defined instructionwise

Backtracking Machine (for Tree Computations)

- **If mode = ramify then**

  Let k = |alternatives (Params)|

  Let o_1 ,..., o_k =new (NODE)

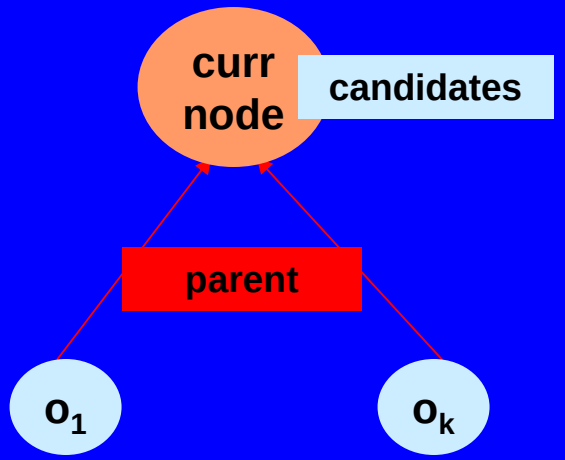
  candidates (currnode) := { o_1 ,..., o_k }

  forall $1 \leq i \leq k$ do

  parent (o_i) := currnode

  env (o_i) := i-th (alternatives (Params))

  mode := select



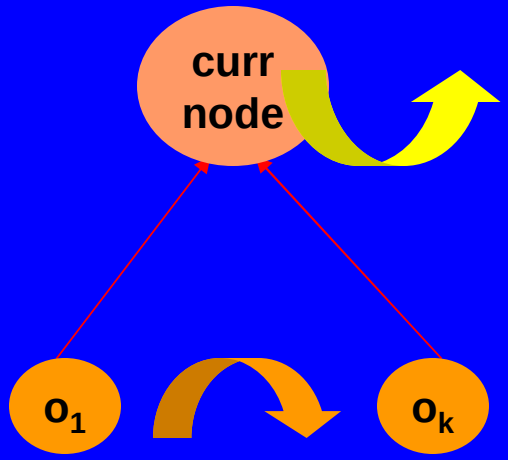
- **If mode = select then**

  If candidates (currnode) = ∅

  then backtrack

  else try-next-candidate

  mode := execute



Backtracking Machine

- backtrack ° if parent (currnode) = root
  then mode := Stop
  else currnode := parent (currnode)
- **try-next-candidate** ° depth-first tree traversal

  currnode:= next (candidates(currnode))

  delete next (candidates(currnode)) from candidates (currnode)
- The fctn next is a choice fct, possibly dynamic, which determines the order for trying out the alternatives.
- The fct alternatives, possibly dynamic and coming with parameters, determines the solution space.
- The execution machine may update mode again to ramify (in case of successful exec) or to select (for failed exec)

Backtracking Machine: logic instantiation

- Prolog Börger/Rosenzweig Science of Computer Programming 24 (1995)
  - alternatives = procdef (act,pgm), yielding a sequence of clauses in pgm, to be tried out in this order to execute the current statement ("goal") act
    - procdef (act,constr,pgm) in CLAM with constraints for indexing mechanism Börger/Salamone OUP 1995
  - next = first-of-sequence (depth-first left-to-right tree traversal)
  - execute mode resolves act against the head of the next candidate, if possible, replacing act by that clauses' body & proceeding in mode ramify, otherwise it deletes that candidate & switches to mode select

Backtracking Machine: functioal progg instantiation

- Babel Börger et al. IFIP 13 World Computer Congress 1994, Vol.I
  - alternatives = fundef (currexp,pgm), yielding the list of defining rules provided in pgm for the outer fct of currexp
  - next = first-of-sequence
  - execute applies the defining rules in the given order to reduce currexp to normal form (using narrowing, a combination of unification and reduction)

Backtracking Machine: context free grammar instantiation

- Generating leftmost derivations of cf grammars G
  - alternatives (currnode,G), yields sequence of symbols Y1...Yk of the conclusion of a G-rule with premisse X labeling currnode. Includes a choice bw different rules X→w
  - env yields the label of a node: variable X or terminal letter a
  - next = first-of-sequence (depth-first left-to-right tree traversal)
  - execute mode
    - for nodes labeled by a variable triggers tree expansion
    - for terminal nodes extracts the yield, concatenating terminal word to output, continues derivation at parent node in mode select

If mode = execute then
  If env (currnode)∈ VAR
    then mode:=ramify
    else output:=output * env(currnode)
         currnode:= parent(currnode)
         mode := select

alternatives can be a dynamic fct (possibly monitored by the user) or static (with first argument in VAR)

Initially NODE = {root}
root=currnode
env(root)=G-axiom
mode=ramify

Backtracking Machine: instantiation for attribute grammars

- Synthesis of node attribute from children's attributes via backtrack ° if parent (currnode) = root then mode := Stop

  else currnode := parent (currnode)

  $X.a := f(Y_1.a_1, ..., Y_k.a_k)$

  - where X = env(parent(currnode)), Y_i =env(o_i) for children nodes
- Inheriting attribute from parent and siblings
  - included in update of env (e.g. upon node creation) generalized to update also node attributes
- Attribute conditions for grammar rules
  - included in execute-rules as additional guard to yielding output

Johnson/ Moss Linguistics &Philosophy 17 (1994) 537-560

If mode = execute then …

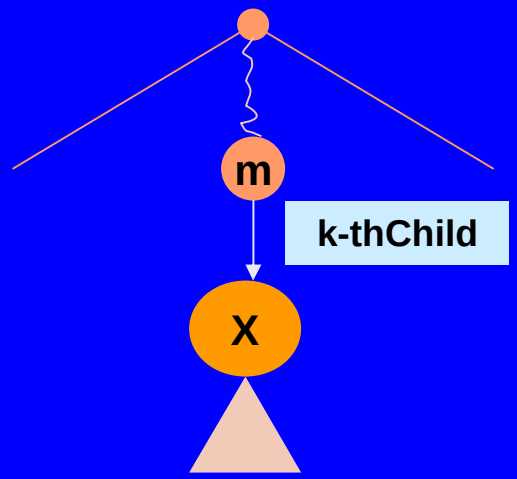
else If Cond(currnode.a, parent(currnode).b, siblings(currnode).c)

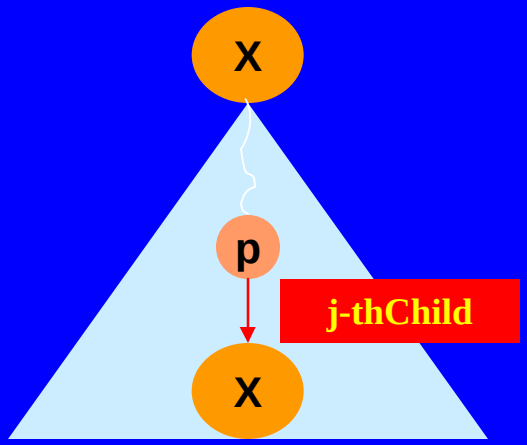
then output:=output * env(currnode)

currnode:= parent(currnode) , mode := select

Tree Adjoining Grammars

Generalizing Parikh's analysis of context free languages by pumping of cf trees from basis trees (with terminal yield) and recursion trees (with terminal yield except for the root variable)

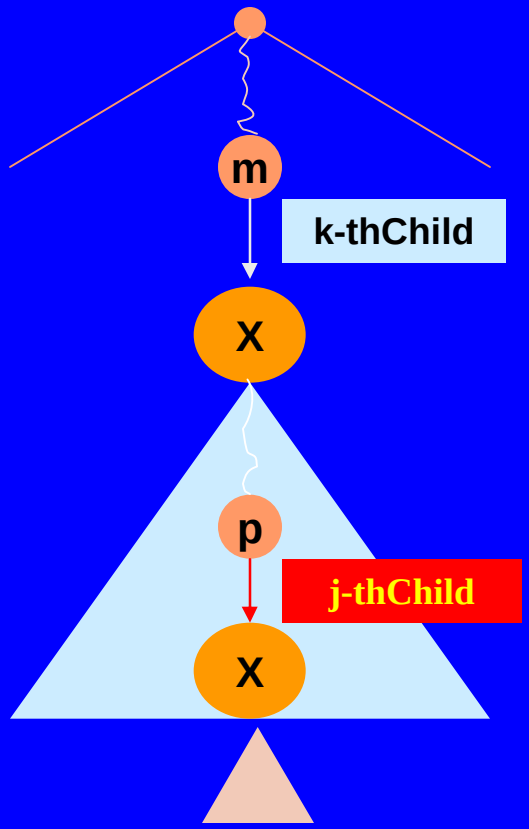




If n=k-thChild(m) & symb(n)=symb(root(T)) & T Î RecTree & foot(T) = j-thChild(p)

Then

Let T'=new copy(T) in k-thChild(m):=root(T') j-thChild(p'):=n

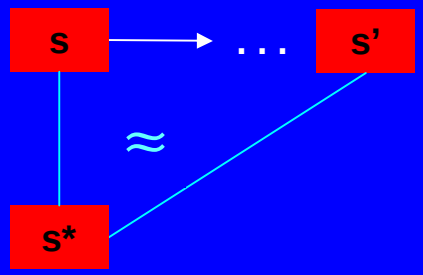


Looking for invariants to prove ASM refinement correctness

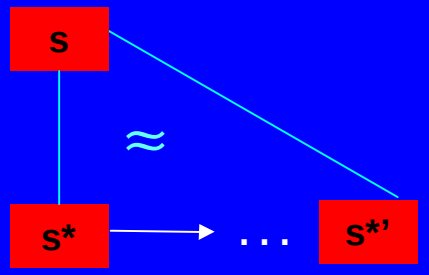
- Idea: find commuting diagrams with end points s, s* which satisfy an invariant ≈ implying the to be established equivalence ≡
- Realization: for each pair of corresponding states - not both final - satisfying ≈, follow the two runs to find a successor pair s', s*' (of corresponding states satisfying ≈)
- 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

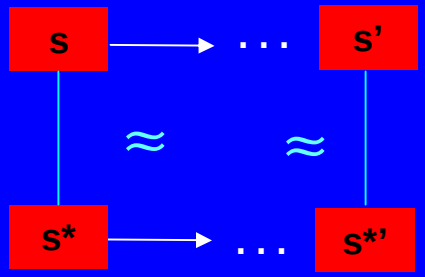
(m,0)-triangle: comp segment
leading in m>0 steps to an s'≈s*



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



(m,n)-trapezoid: computation
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 $(s,s^{*'}) <_{0n} (s,s^*)$

- 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 $s' \approx s^{*'}$

applying triangles successively must be well-founded

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

Exercise

- Prove that in the correctness definition of ASM refinements one can assume without loss of generality that the sequences of corresponding states are minimal, in the sense that between two sequence elements there are no other equivalent states
  - i.e. there are no $i_k<i<i_{k+1}$, $j_k<j<i_{k+1}$ with $S(i) \equiv S^*(j)$

References

AsmBook 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

ASM Refinement Case Study Book R. Stärk, J. Schmid, E. Börger Java and the Java Virtual Machine: Definition, Verification, Validation Springer 2001, see http://www.inf.ethz.ch/~jbook

ASM Refinement Analysis G. Schellhorn Verification of ASM Refinements Using Generalized Forward Simulation J. Universal Computer Science 7 (11) 2001

ASM Survey E. Börger High Level System Design and Analysis using ASMs LNCS Vol. 1012 (1999), pp. 1-43

ASM History E. Börger The Origins and the Development of the ASM Method for High Level System Design and Analysis JUCS 8 (1) 2002

References on Backtracking Machine

- E.Börger and D. Rosenzweig: Mathematical Definition of Full Prolog
  - In: Science of Computer Programming 24 (1995) 249-286
- E.Börger and R.F.Salamone: CLAM Specification for Provably Correct Compilation of CLP (R) Programs
  - In: E.Börger (Ed.) Specification and Validation Methods. Oxford University Press, 1995, 97-130
- E.Börger, F.J.Lopez-Fraguas, M.Rodrigues-Artalejo: A Model for Mathematical Analysis of Functional Programs and their Implementations
  - In: B.Pehrson and I.Simon (Eds.): IFIP 13 World Computer Congress 1994, Vol.I: Technology/Foundations, 410-415
- D. Johnson and L. Moss: Grammar Formalisms Viewed als Evolving Algebras
  - Linguistics and Philosophy 17 (1994) 537-560

References

Four Books on Refinement Methods

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