Computation and Specification Models
A Comparative Study

Egon Börger

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

For details see Chapter 7.1 (Integrating Computation and Specification Models) 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

Goal: comparative analysis of spec and comp systems

- We look for standard reference descriptions for the principal current models of computation and of high-level system design, which
  - faithfully capture each system's fundamental characteristic intuitions
    - about the objects of computation and the nature of a basic computation step
  - are uniform enough to allow explicit comparisons of established system modeling methods
    - to contribute to rationalize the scientific evaluation of different system specification approaches, clarifying their advantages and disadvantages

Current Models of Computation to be Compared

- UML Diagrams for System Dynamics
- Classical Models of Computation
  - Automata: Moore-Mealy, Stream-Processing FSM, Co-Design FSM, Timed FSM, PushDown, Turing, Scott, Eilenberg, Minsky, Wegner
  - Substitution systems: Thue, Markov, Post, Conway
  - Structured programming
    - Programming constructs: seq, while, case, alternate, par
    - Gödel-Herbrand computable fcts (Böhm-Jacopini)
  - Tree computations: backtracking in logic & functional programming, context free grammars, attribute grammars, tree adjoining grammars
- Specification and Computation Models for System Design
  - Executable high-level design languages: UNITY, COLD
  - State-based specification languages
    - distributed: Petri Nets
    - sequential: SCR (Parnas Tables), Z/B, VDM
  - Virtual Machines: Active Db, Data Flow (Neural) Machines, JVM
  - Stateless modeling systems
    - Logic based (axiomatic), denotational (functional pgg paradigm), algebraic (process algebras, CSP, LOTOS, etc.)

Thesis: ASMs a universal class of algorithms

- The ASM thesis in its original form reads:
  - Every computational device can be simulated by an appropriate dynamic structure – of appropriately the same size – in real time (Y. Gurevich, Notices American Mathematical Society 85T-68-203, 1985).
- For the synchronous parallel case of this thesis Blass and Gurevich (ToCL 2002) discovered a small number of postulates from which every synchronous parallel computational device can be proved to be simulatable in lock-step by an ASM.
- So why do we not compare different systems via the ASMs as given by that proof, machines which "can simulate" the given systems "step-by-step"?

"Abstract" nature of ASMs derived from postulates

- Postulating (by an existential statement) e.g. that
  - states are appropriate equivalence classes of structures of a fixed signature (in the sense of logic)
  - evolution happens as iteration of single "steps"
  - the single-step exploration space is bounded (i.e. that there is a uniform bound on memory locations basic computation steps depend upon, up to isomorphism)

  does not by itself provide, for a given computation or specification model, a standard reference description of its characteristic
  - states
  - objects entering a basic computation step
  - next-step function
- No proof is known to include distributed systems

A price for "proving" computational universality

- If one looks for explicitly stated assumptions, to prove by a mathematical argument the step-for-step-universality of ASMs for every theoretically possible system, the focus in stating the postulates unavoidably is on generality and uniformity, to capture the huge variety of data structures and of ways of using them in a basic computation step.
- As side effect of the generality of the postulates, the application of the general proof scheme to established models of computation
  - may yield ASMs which are more involved than necessary
  - may blur distinctions which pragmatically differentiate concrete systems
    - The construction by Blass and Gurevich in op.cit., "transforming" any imaginable synchronous parallel computational system into an ASM simulating the system step-by-step, depends on the way the abstract postulates capture the amount of computation (by every single agent) and of the communication (between the synchronized agents) which are allowed in a synchronous parallel computation step.

The epistemological character of the ASM thesis

- The epistemologically relevant unfolding of the concrete objects and steps for any theoretically conceivable computational system, by deriving ("decoding") them from the general concepts appearing in the postulates for a proof of the thesis, yields some en/decoding overhead one can avoid by concentrating on - the great variety of - relevant (established or desirable) concrete classes of systems.
- Focus on modeling significant classes of systems allows us to follow a pragmatically important principle the ASM design and analysis approach emphasizes, namely to model concrete systems "closely and faithfully", "at their level of abstraction",
    - laying down the essential computational ingredients completely and expressing them directly,

  without using any encoding which is foreign to the device under study.

Goal of naturally modeling systems of specification & computation

- We look for "natural" ASM descriptions of the principal current models of computation and of high-level system design, including asynchronous distributed systems, which
  - directly reflect the basic intuitions and concepts of every framework
    - By gently capturing the basic data structures & single computation steps which characterize each significant system, we provide a strong argument for the ASM thesis which
      - avoids a sophisticated existence proof for the ASM models from abstract postulates
      - avoids decoding of concrete concepts from abstract postulates
      - avoids a sophisticated correctness proof for the ASM models
  - are formulated in a way which is "uniform" enough to allow explicit comparisons bw the classical system models
    - By providing a mathematical basis for technical comparison we
      - contribute to rationalize the scientific evaluation of different system specification approaches, clarifying their advantages and disadvantages
      - offer a powerful yet simple framework for teaching computation theory

Classes of ASMs Reflecting UML Notations

- UML offers an ensemble of notations with loose semantics
- "Behavioral" diagrams for describing system dynamics can be equipped with a rigorous semantics by defining them as special ASMs, e.g.
  - Activity diagrams (see Cavarra/Börger/Riccobene LNCS 1816)
  - State diagrams (see Cavarra/Börger/Riccobene LNCS 1912)
  - Use case, sequence, collaboration diagrams
- "Structural" diagrams for describing system statics can be used for specifying static parts of ASMs, e.g.
  - Class and object diagrams (organized in packages)
  - Implementation (component and deployment) diagrams

For the modeling purpose here, we generalize FSMs to

ASMs tailored to UML diagram visualizable machines

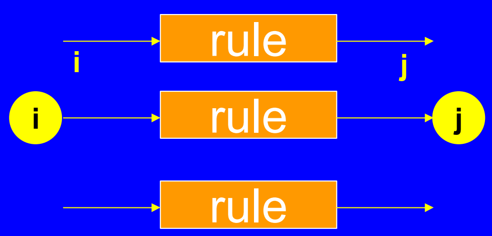
UML Action Nodes: diagram notations for action flow

Idea: in a given situation, perform an action and proceed

UML notation

FSM notation

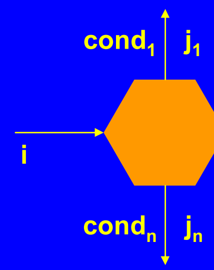
Flowchart notation



Meaning: if control = i then rule
control := j

Interpreting "action" as application of an ASM rule

UML Branching Nodes: diagram notations for control flow



... **meaning**

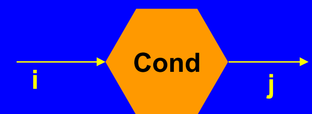
if control = i then

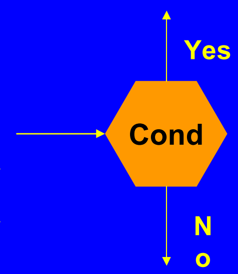
if $cond_1$ then control := j_1

...

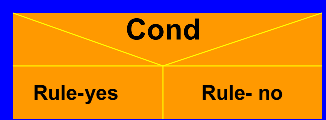
if $cond_n$ then control := j_n

Special notation for n=1:



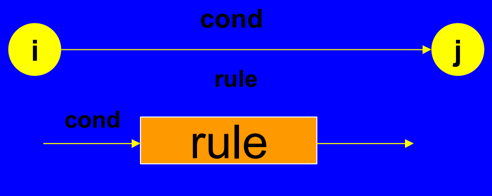


Special notations for n=2:



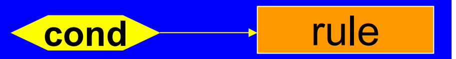
Control State ASMs: combining action/branching nodes

FSM notations



Flowchart notation

cond often inscribed into a rhomb



if control = i and cond then rule

control := j

Control State ASM (Abstract FSM): all rules have this form

NB. Evaluation of Cond and firing rule "controlled" as ONE ASM STEP

UML Activity Diagrams with Concurrent Nodes

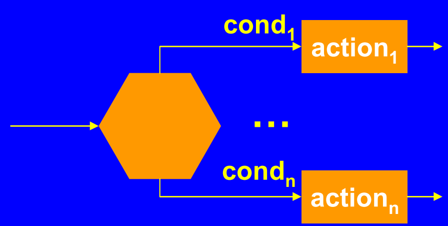
UML Activity Diagram **graph connecting action & branching nodes**

- Concurrent nodes of UML, in the synchronous understanding, are a special case of action nodes where
  rule = $rule_1$
  …
  $rule_n$ (all rules fire simultaneously)
- Concurrent nodes of UML, in the asynchronous understanding, are calls of asynchronous multi-agent ASMs
  - work with a priori unrelated clocks, but
  - are (expected to be) synchronized after each of them has returned a result (similar to the par construct of Occam)

Def. Synchronous UML Activity Diagram: synchronous concurrent nodes

Synchronous UML activity diagrams have a normal form of multi-agent control state ASMs

Each synchronous UML activity diagram is built up from control state ASM rules i.e. alternating branching and action nodes of the following form for each of the synchronized agents (where n=1 is allowed):



Therefore every synchronous UML activity diagram can be viewed as a synchronous multi-agent ASM whose agents are control state ASMs with rules representing alternating branching and action nodes

Classical Models of Computation

- Automata
  - Moore-Mealy, Stream-Processing FSM, Co-Design FSM, Timed FSM
  - PushDown
  - Turing, Scott, Eilenberg, Minsky, Wegner
- Substitution systems
  - Thue, Markov, Post, Conway
- Structured programming
  - Programming constructs (seq, while, case, alternate, par)
  - Gödel-Herbrand computable functions (Böhm-Jacopini)
- Tree computations
  - backtracking in logic & functional programming
  - context free grammars
  - attribute grammars
  - tree adjoining grammars

Mealy/Moore automata as control state ASMs

in_put | a | → control → | b | out_put

Program of rules of the form	→ in = a → out := b →
Moore automata: without output	

Writing programs in standard tabular form (i,a,j,b) yields a guard-free FSM rule scheme updating control, out:

control := Nxtctl(control, in)
out := Nxtout(control, in)

NB. Since "in" is a monitored fct, it is not updated in the rule scheme

1-way or 2-way is a question of Moves of input head

replacing in by in(head) and adding head := head + Move(control, in(head))

Specializing Mealy to Stream Processing Ctl State ASMs (Janneck 2000)

Computing Stream Functions $S^m \rightarrow S^n$ (data set $S = A^*$ or $S=A^N$)
yielding an output stream out resulting from consumption of the input stream in

non-deterministically in each step these automata :
- read (consume) at every input port a prefix of the input stream in
- produce at each output port a part of the output stream out (concatenation)
- proceed to the next control state

Prefix: Ctl $\times$ $S^m \rightarrow$ PowerSet($S^m{}_{fin}$) yielding sets of finite prefixes
Transition: Ctl x ($S^m{}_{fin}$) $\rightarrow$ PowerSet(Ctl $\times$ $S^n{}_{fin}$) yielding finite output

rules of form

choose pref $\in$ Prefix (control, in)
choose (c,o) $\in$ Transition (control, pref)
in:= in$-$pref
out := out.o
control := c

Co-design FSMs = distributed Mealy-ASMs Sangiovanni-Vincentelli

Mealy-ASM: rules of form → in = a → rule →

i.e. Mealy FSM update "out:=b" replaced by "rule"
needed for arbitrary combinational (external & instantaneous) fcts

Often with global agent scheduler
and/or with timing conditions
for agents performing durative instead of atomic actions

Nondeterministic versions are of form
choose R $\in$ Rule
R
where Rule is the set of rules to be chosen from

Timed Automata (Alur & Dill) as ctl state ASMs

- letter input enriched by real-valued occurrence time
- transitions enriched by clocks (recording time-Δ wrt previous input)
  - fire under clock constraints
  - update clocks (reset or adding time-Δ of input)

Rules of form → in = a & constr ($time_\Delta$) → forall c $\in$ reset c:=0 ; forall c $\notin$ reset c:=c + $time_\Delta$ →

where $time_\Delta$ = occurrenceTime (in) $-$ occurrenceTime (previousIn)

NB. Typically the constraints are about input to occur within (<,≤) or after (>, ≥) a given (constant) time interval, leaving some freedom for timing runs – i.e. choosing sequences of occurrenceTime (in) to satisfy the constraints.

Push Down Automata as control state ASMs

Reading from input and/or stack and writing on stack

stack ...... a **i** b ...... **input**

control

i.e. rules of form (states may be no-input-/no-stack-reading) :

[in = a] & [top(stack) = b] → stack := push(w, [pop](stack)) →

control := Nxtctl(control, in, top(stack))
stack:=Pop&Push(stack,Write(control, in, top(stack)))

Turing automata as Control State ASMs

Turing machines combine in_put and out_put into one memory:

Program of rules of the form

→ Cond (mem(env(pos)) → update mem(env(pos)) update pos →

instantiated for Turing's original machines to

→ tape(head) = a → tape(head) := b [head := head ± 1] →

control := Nxtctl(control, tape(head))
tape(head) := Write(control, tape(head))
head := head + Move(control, tape(head))

Variants of TMs instantiating mem,env,pos

memory = tape → tape(head) = a → tape(head) := b [head := head ± 1] →

memory = k tapes — **pos : Z or pos: Z^k (k-head TM)**

memory = n-dim pattern — **env(pos) $\subseteq_{fin}$ Z^n including pos**

memory = $N^n/(A^*)^n$ (registers) — **pos = 1,…,n "softwired in instrs"**

Minsky 1961, Sheperdson& Sturgis 1963

→ reg(i) = 0 —yes→
—no→ reg(i) := reg(i) +/- 1 →

Eilenberg's X-Machines as control state ASMs

Eilenberg's X-machines (1974) add to Mealy machines
global memory with global memory update functions
- depending on input and control state, they modify memory and control state and provide output

Rules of form → in = a → out := b mem:=f(mem) →

control := Nxtctl(control, in)
mem:=Opern(control, in)(mem)
out := Nxtout(control, in)

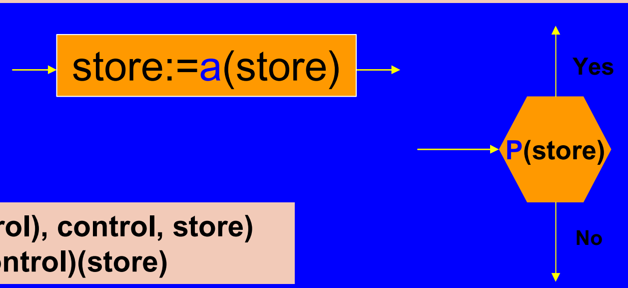
- global memory yields **frame problem**
- global mem functions f make appropriate local **updating of data structures difficult**

Similarly for Stream X-Machines (Holcombe J.SE 1998)

Scott Machines (J.CSS 1967) as **control state ASMs**

Instrs trigger actions or test Predicates on abstract store

i.e. each rule has one of the two forms



control := IF(Test(control), control, store)
store:=Action(control)(store)

- global store yields **frame problem**
- global store functions/predicates make appropriate **test/updating of data structures difficult**

Extending TM to Wegner's Interacting Turing Machines

New: at each step TM may - receive input from environment
- yield output to environment

control := Nxtctl(control, tape(head), input)
tape(head) := Write(control, tape(head), input)
head := head + Move(control, tape(head), input)
output (control, tape(head), input)

Considering the output as written on the in-out tape means defining the output action by :
output:= input*out(control, tape(head), input)

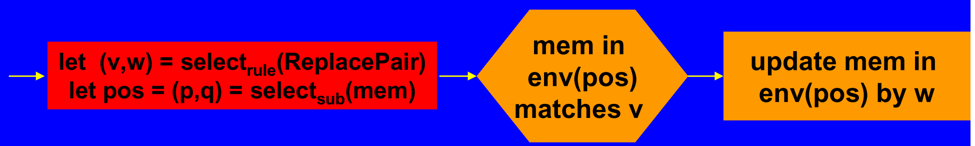
Viewing input as a combination of preceding inputs/outputs and the new user input :
input = combine (output, user_ input)

Single versus Multiple Stream Interacting TMs (SIM/MIM) is only a question of instantiating input to $(inp_1,\ldots,inp_n)$

Local Substitution: Thue , Post, Markov systems

Thue

mem: A*, ReplacePair $\subseteq$ A* $\times$ A*
choose (v,w), choose interval of mem where v occurs, to replace that occurrence of v by w



Exls: regular grammars, context free grammars, context sensitive grammars,…

Markov

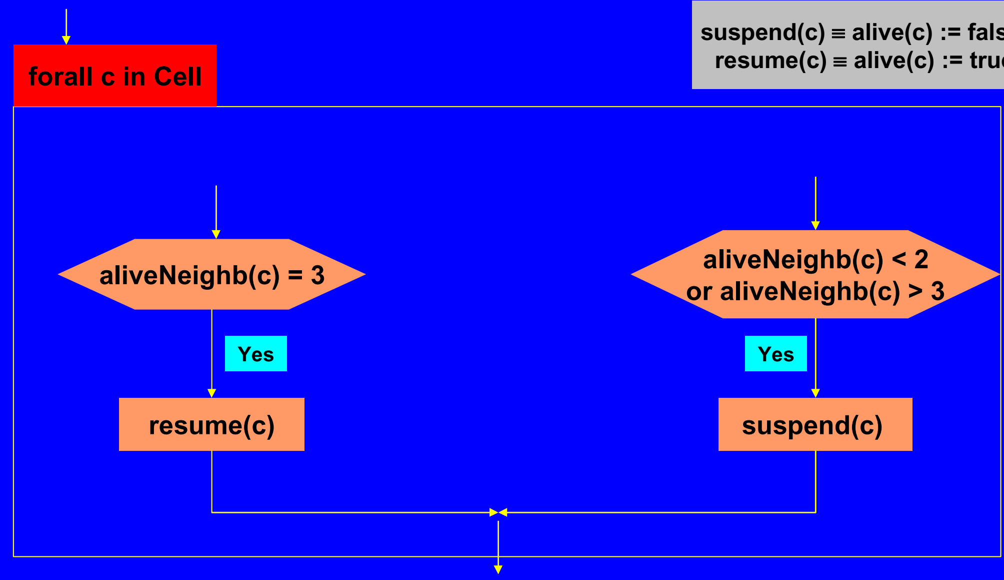
Deterministic Thue system: ReplacePair is ordered
select$_{rule}$(ReplacePair, mem) takes first pair with premise, say v, in mem
select$_{sub}$ (mem, v) takes the leftmost occurrence of subword v in mem

Post normal

select$_{sub}$ (mem) takes an initial subword of mem
updating mem deletes initial subword v and copies w at end

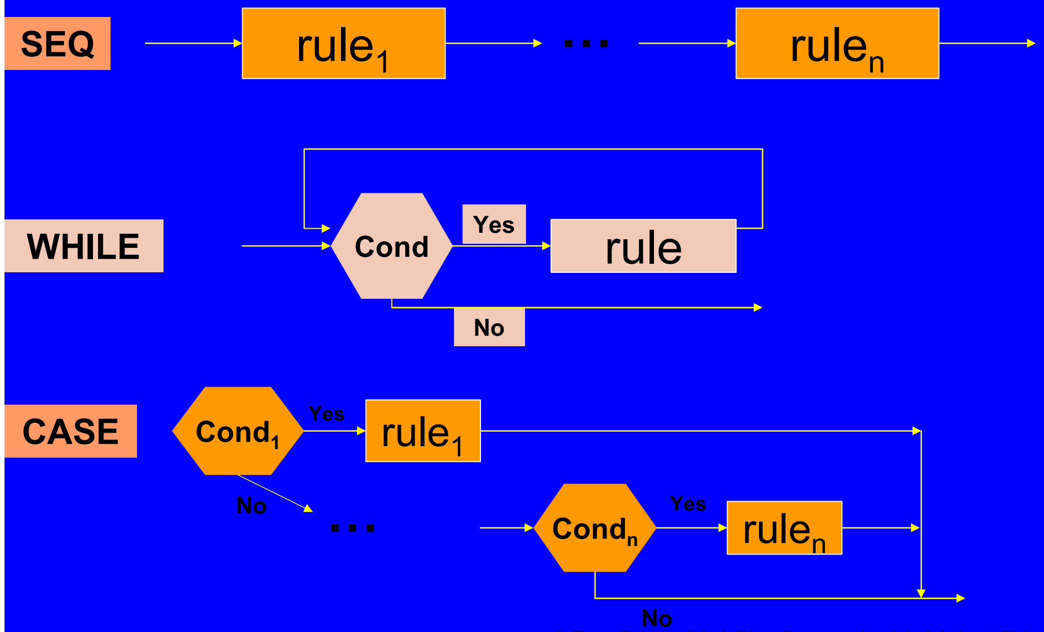
Simultaneous substitution: E.g. Conway's game of life

suspend(c) ≡ alive(c) := false
resume(c) ≡ alive(c) := true

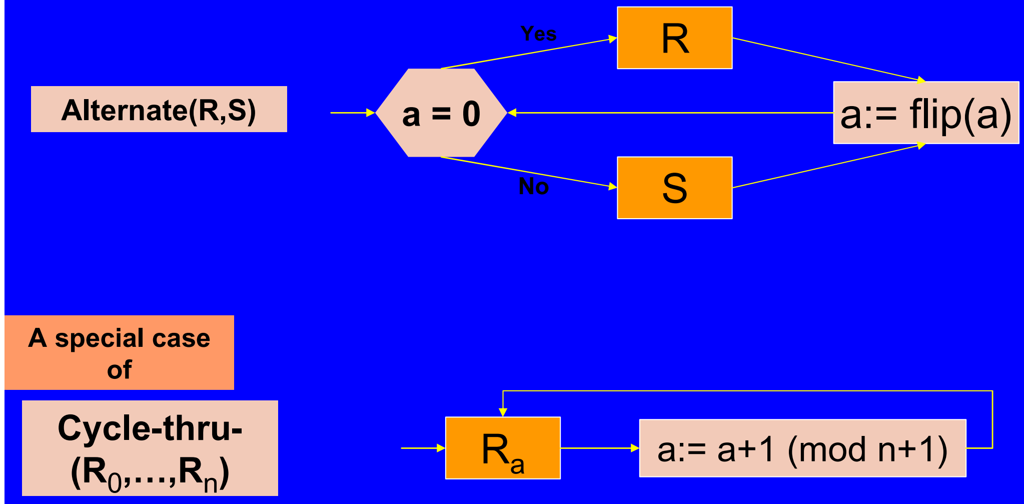


Pattern: Fire simultaneously in "neighbouring places" a rule
If Cond(Neighb(p)) then SubstitutionRule(p)

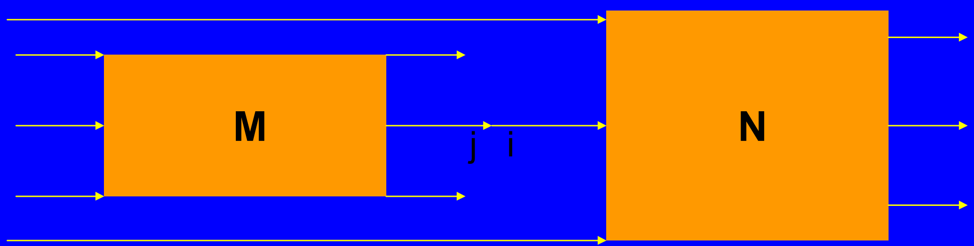
Control State ASMs for standard sequencing constructs (white box view)



Control State ASMs for standard iteration constructs (white box view)



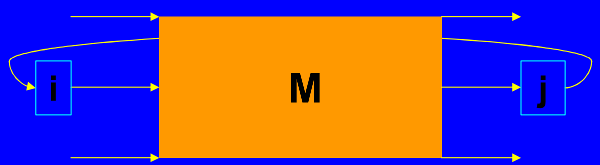
Networks of Mealy ASMs (seq & par composition)



i.e. adding to M rules: if out = j then in :=i

hiding the two input/output channels by this internal connection

feedback operator



deleting i/j from input/output lines (white box view)

+ parallel composition M N

For normal forms based upon 2 automata K, E see D. Rödding LNCS 185 (1983)

Structured Programming: **Computing Recursive Functions**

Böhm-Jacopini-ASMs defined recursively

- from sequential ASMs using seq and iterate
- the only static functions: the initial functions
  - projection, const, + 1, = 0
- only one monitored function per machine, 0-ary, say in for inputting the sequence of args, which does not change its value during a computation
- only one output fct per machine, say out : N
- no shared functions

Black Box View of seq, iterate **encapsulating finitely many steps into one atomic action ("accumulated set of updates") as defined in** "Composition and Submachine Concepts for Sequential ASMs"

Börger/Schmid CSL'2000, LNCS 1862

Structured Programming Theorem Comm. ACM 1966

Every partial recursive function can be computed by a Böhm- Jacopini- ASM.

- Proof by induction on partial recursive functions.
- Each initial function f is computed by the following machine F
  - consisting of only one function update, reflecting the (operational?!) "application" of the defining equation of f to determine the value of f for the given arguments

$$F \equiv out_F := f(in_F)$$

Computing Primitive Recursion

Let $f(x,0) = g(x)$, $f(x,y+1) = h(x,y,f(x,y))$

Let g, h be computed by G , H

Then f is computed by

$F \equiv$ **let** $(x, y) = in_F$ **in**

  {ival : = G (x) , rec : = 0 }

  seq (while (rec < y)

    { ival : = H(x, rec, ival), rec : = rec + 1})

  seq out_F : = ival

Computing Simultaneous Substitution

- Let $f(x) = g(h_1(x), \dots, h_m(x))$
- Let g, $h_1, \dots, h_m$ be computed by G , $H_1, \dots, H_m$
- Then f is computed by

$$F \equiv \{H_1(in_F), \dots, H_m(in_F)\}$$
$$\textbf{seq}\ out_F := G(out_{H1}, \dots, out_{Hm})$$

- using {…} for par (simultaneous execution)
  - reflecting independence of g-arguments from their evaluation order
- macros for connecting H to input in and output out
  - reflect sequential order for reading arguments and providing values
  - H (in) ≡ in_H : = in seq H first, arguments are given as input
  - out : = H (in) ≡ at the end, values are given as result

    in_H : = in seq H seq out : = out_H

Computing μ-Operator

- Let $f(x) = \mu y\,(g(x,y) = 0)$
- Let g be computed by G
- Then f is computed by

$F \equiv$ {G (in_F ,0) , rec : = 0 }

  seq (while ($out_G \neq 0$)

    {G (in_F , rec + 1) , rec : = rec + 1 })

  seq out_F : = rec

NB. The preceding ASMs unfold the underlying mechanism for the evaluation of terms, which is partly sequential, partly parallel, hardwired in our brains & taken for granted in the functional interpretation of the defining Gödel-Herbrand equations

Backtracking Machine (for Tree Computations)

- **If mode = ramify then**

  Let k = |alternatives (Params)|

  Let $o_1, \ldots, o_k$ =new (NODE)

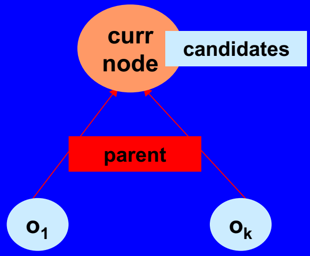
  candidates (currnode) := { $o_1, \ldots, o_k$ }

  forall $1 \le i \le k$ do

  parent (o_i) := currnode

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

  mode := select



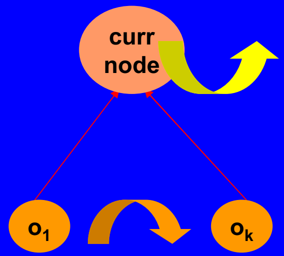
- **If mode = select then**

  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 progg 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

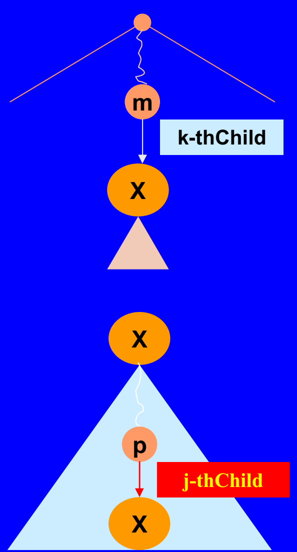
  backtrack ≡ if parent (currnode) = root then mode := Stop
  else currnode := parent (currnode)
  $X.a := f(Y_1.a_1, ..., Y_k.a_k)$
    - where X = env(parent(currnode)), Y_i =env(o_i) for children nodes
- Inheriting attribute from parent and siblings
  - included in update of env (e.g. upon node creation) 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

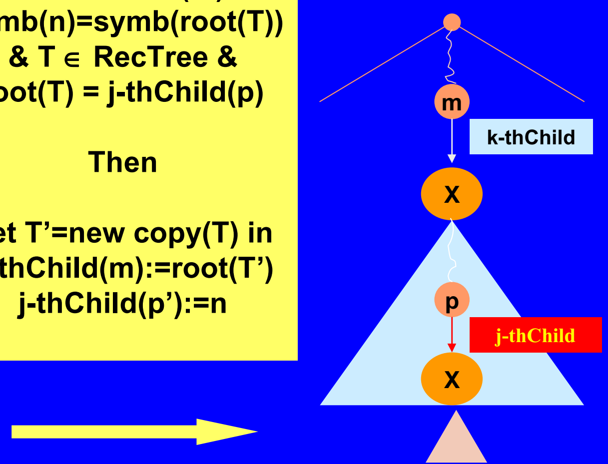
Generalizing Parikh's analysis of context free languages by pumping of cf trees from basis trees (with terminal yield) and recursion trees (with terminal yield except for the root variable)



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

Then

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



Specification & Computation Models for System Design

- Executable high-level design languages
  - UNITY
  - COLD
- State-based specification languages
  - distributed: Petri Nets
  - sequential: SCR (Parnas Tables), Z/B, VDM
- Virtual Machines
  - Dijkstra's Abstract Machine Concept
  - Active Db
  - Data Flow (Neural) Machines
- Stateless Modeling Systems
  - Process Algebras (CSP, LOTOS, etc.)
  - Logic Based Systems (denotational, algebraic, axiomatic)

UNITY vs ASMs: similarities

“Parallel Program Design. A Foundation”

by K. Mani Chandy and Jayadev Misra, Addison Wesley, 1988

- Formal, design oriented, state based, high-level description of systems
- Absence of control flow
- Computations as sequences of state transitions
- Parallelism of simultaneous multiple conditional assignments
- Sharing of “data” via their names

UNITY vs ASMs : differences

- Time: global synchronous UNITY system time, one clock to schedule the statements of every program in the system; in distributed ASMs each agent can have its own clock, for every sequential ASM all rules are executed simultaneously
- Interleaving and Fairness Condition on Runs
- Specialized Refinement/Composition concept
- UNITY is linked to a particular proof system geared to extract proofs from pgm text
- UNITY has no Function Classification
- non-determinism restricted to choosing rules

UNITY statements as ASMs rules

UNITY	ASMs
Multiple assignment	
x,y,z:=0,1,2	x,y,z:=0,1,2
Conditional assignment	
x:=-1 if y<0 0 if y=0 1 if y>0	if y<0 then x:=-1 elseif y=0 then x:=0 elseif y>0 then x:=1
Quantified assignment	
<|| i : 0≤i<N : A[i]:=B[i]>	forall i in [0,…,N] A[i]:=B[i]

UNITY_ASM

UNITY_ASM ua — ASM name, a string

RULES — ASM rule declarations

r_1=…

…

r_n=…

ua.rules={r_1,…,r_n} — Rule universe

BODY

choose r ∈ ua.rules — Scheduling at the rule level

r — Execution of the scheduled rule

endchoose

UNITY_SYSTEM_ASM

UNITY_SYSTEM_ASM usa
COMPONENTS
UNITY_ASM ua_1
…
UNITY_ASM ua_n
components={ua_1,…,ua_n}
BODY
choose c ∈ components
c
endchoose

ASM name, a string

UNITY_ASM declarations

Component universe

Scheduling at the components level

Execution of the BODY of the scheduled component

Unity slides courtesy of Simone Semprini

COLD vs ASMs : similarities

"Formal Specification and Design"
by L.M.G. Feijs and H.B.M. Jonkers, Cambridge Univ. Press 1992

- Common OO Lg for Design combining abstract data types (VDM,Z) with states for system descriptions ranging from high-level to implementation ("wide-spectrum")
- Kernel language
  – with user- and application-oriented extensions
- States as structures
- Computations as sequences of state transitions
- Parallelism of simultaneous multiple conditional assignments
- Basic constructs
  – skip, choose (for rules and variable assignments), let

COLD vs ASMs : differences

- Purely sequential :
  – State transitions viewed as sequential execution of procedure calls, built from stms viewed as expressions with side effect
- No Function Classification, no explicit "forall" construct
- Object Oriented Programming Language constructs:
  – a class (with a set of states, one initial state, and a set of transition relations) corresponds to an ASM, but
  – different states of a same class may have different signature
- Sequencing and iteration constructs (black box view)
- COLD linked to a dynamic logic proof system supporting ADT
  – geared to provide proofs for algebraic specifications of states and their dynamics (a la Z, VDM)
- separate guard stm for Blocking Evaluation of Guards
  – (i.e. identity state transition only if the guard is true)

COLD statements as ASMs rules

COLD	ASM
Multiple non-deterministic assignment	
MOD V END (arbitrary modification of some variables)	choose n∈N, x_1...x_n ∈ V choose v_1...v_n ∈ Value forall 1≤i≤n x_i:=v_i
Non-deterministic sequential procedure invocation	
USE P END (arbitrary sequence of procedure invocations)	choose n∈N, p_1...p_n ∈ P p_1 seq...seq p_n

Specification & Computation Models for System Design

- Executable high-level design languages
  - UNITY
  - COLD
- State-based specification languages
  - distributed: Petri Nets
  - sequential: SCR (Parnas Tables), Z/B, VDM
- Virtual Machines
  - Dijkstra's Abstract Machine Concept
  - Active Db
  - Data Flow (Neural) Machines
- Stateless Modeling Systems
  - Process Algebras (CSP, LOTOS, etc.)
  - Logic Based Systems (denotational, algebraic, axiomatic)

Modeling Petri Nets as asynchronous multi-agent ASMs

General view of Petri nets as distributed transition systems transforming objects under given conditions

- Classical instance (Petri):
  - objects are **marks** on places
    - places, denoted by circles, are passive net components to store objects ("locations")
  - **transitions** modify objects by adding and deleting marks on places
    - transitions are active net components, denoted by boxes ("rules")
- Modern instances (predicate/transition nets):
  - **places** are locations for objects belonging to abstract data types, i.e. variables taking values of given type (marking = variable interpretation)
  - **transitions** update vars and extend domains under conds
  - **conditions** are arbitrary first-order formulae

Modeling Petri Nets as asynchronous multi-agent ASMs

The numerous extensions of classical Petri nets all are forms of the following class of asynchronous multi-agent ASMs:

- State
  - P set of " **places** " ("passive" net components)
  - A set of "**agents**" (which execute transitions)
  - F class of "**value assigning**" (state changing) fcts
- Rules (one agent for each "transition") of the following form, where pre/post-places are sequences/sets of places, participating in the "information flow relation" (local state change):

If cond(pre-places)

then updates(post-places)

where **updates(post-places)**

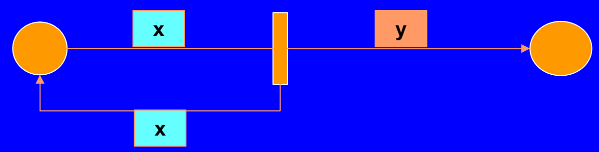
("active net components") are sets of f(p) := t

Includes view of states as logical predicates, associated to places & transformed by actions

Avoiding Frame Problem in Petri Nets

The ASM-like view of "states as logical predicates", associated to places and transformed by actions, helps to avoid a form of frame problem traditional Petri nets come with:

- namely when in a transition some "marks" are deleted from pre-places to be put back again by the transition



Comparing ASMs and Parnas Tables (SCR) Common Goals

- provide documentation for understanding by humans
- use functions & variables, functions are monitored or controlled
- standard mathematical language
- functions dynamic via time
- structure of buildings blocks and decomposition traces

- … through ground models and hierarchy of refinements
- … functions of arbitrary arity, arbitrarily complex locs, also static, derived, shared fcts
- … and algorithmic (executable) process notation
- … and possibly distributed coming with different times
- …common programming structures

Börger Dagstuhl Seminar Report 149 (1996)

Comparing ASMs and Parnas Tables (SCR) Differences

- Parnas tables come with
  - frame problem (declarative x/x'-notation yields NC/No Change clauses)
  - difficult semantics (see Parnas-Madey in SCP 25,1995)
    - complex classification of tables
    - no semantical foundation for use of auxiliary functions
  - restriction to sequential systems of finitely many state variables (functions of time, either monitored or controlled)
  - special matrix notation (2-dimensional layout of CASE OF)
  - **hard to extend** to cope with practical needs like relations (in particular non-determinism), composition, sequencing, stepwise refinement, typing (see SCR paper in NASA LFM'2000)
- Parnas tables are special forms of ASMs

Normal Parnas Tables

Assign value $t_{i,j}$ to f(x,y) under i-th row & j-th column condition

N(f)	c_1	…	c_j	…	c_n
r_1	$t_{1,1}$		…		$t_{1,n}$
⋮					
r_i			$t_{i,j}$		
⋮					
r_m	$t_{m,1}$		…		$t_{m,n}$

ASM notation
forall i≤n, j≤m
if r_i and c_j
then $f(x,y) := t_{i,j}$

Functional notation **f(x,y) := case exp of**
r_i & c_j: $t_{i,j}$

Inverted Parnas Tables

Assign a value t_j to f(x,y) under a leading/side condition

I(f)	t_1	…	t_j	…	t_n
r_1	$c_{1,1}$		…		$c_{1,n}$
⋮					
r_i			$c_{i,j}$		
⋮					
r_m	$c_{m,1}$		…		$c_{m,n}$

If $r_i(x,y)$ then
If $c_{i,j}(x,y)$ then $f(x,y) := t_j$

Parnas Decision Tables

Trigger column action t_j under column condition

D(f)	t_1 …	t_j …	t_n
s_1	$r_{1,1}$ …	$r_{1,j}$ …	$r_{1,n}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
s_m	$r_{m,1}$ …	$r_{m,j}$ …	$r_{m,n}$

ASM notation : forall $j \leq n$ if for all $i \leq m$ $r_{i,j}(s_i)$ then trigger t_j

How to distinguish with table notation if instead of forall $j \leq n$ one means for one $j \leq n$?

Comparing ASM and Z/B

- Z specs difficult to make executable Anthony Hall in ZUM'97,LNCS 1212
- B machines/refinements (B-Book 1996) are based upon
  - pocket calculator model (**one operation/event "per time unit"**)
  - finite sets/functions and states of finitely many variables
- B has axiomatic foundation by wp theory, using syntactic global concept of substitution **(used to define local assignment x := t & parallel composition)**, interpreted by set-theoretic models
- B fixed link between design & proofs (relating syntactical pgm constructs & proof rules) **restricting design space (e.g. including M allowed to call only one operation of included M')**
- B tailored for termination proofs, using restricted refinement notions, **of single operations/events (with "unchanged" properties)**
- B geared to obtain executable programs **from logical descrps**
- B has industrial tool kits (B toolkit, Atelier B), ASM has public domain tools Workbench, ASMGofer, XASM and the MSR tool AsmL

Comparing the Computation Model of B Machines & ASMs

- **"Pocket calculator model"**
  set of operations (which are callable by the user) or of events (which may happen)
  - one at a time ("no simultaneity bw the exec of two events")
  - hiding the machine state (giving the user "the ability to activate the operations" - to "modify the state within the limits of the invariant" - "not to access its state directly", pg.230)

  Structured ASMs provide atomic (zero-time) synchronous parallel execution of entire (sub)machines whose computations, analysed in isolation, may have duration & may access the needed state portion (interface). Turbo ASMs combine atomic black box & durative white box view Börger/Schmid (LNCS 1862)
- B has to define a "multiple generalized substitution" to define the parallel composition of two machines, which is a basic concept in ASMs.

Comparing the Refinement Notions for B Machines & ASMs

- B-refinement only of single operations with unchanged signature, tailored to provide "unchanged" properties
- ASMs provide refinement notions which allow **change of signature (data refinement) & of operation** sequences

$$\begin{tikzcd}
\textbf{State} \arrow[rr, "\tau_1 \ldots \tau_m"] \arrow[d, leftrightarrow, "\text{ref}"', "\text{abs}"] & \boxed{\equiv} & \textbf{State'} \arrow[d, leftrightarrow, "\text{ref}"', "\text{abs}"] \\
\textbf{RefState} \arrow[rr, "\sigma_1 \ldots \sigma_n"] & & \textbf{RefState'}
\end{tikzcd}$$

with equivalence ≡ definable to relate the locations of interest in states of interest, which result from comp segments of interest. Properties can be "preserved" modulo the ref/abs relations

Comparing Links bw Design and Proofs in B Machines & ASMs

- B links design & proofs by relating syntactical program constructs & proof principles, **at the price of restricting the design space**
  - **Exl. 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." (B-Book pg.317)**
  - **Exl. Let M' have the following operations, satisfying the invariant** $v \le w$:
    - **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 ≤ w is broken by M for w = v+1.**
- "…formal reasoning involving events…It would be quite complicated to envisage that two (or more) events could happen simultaneously" (Abrial/Mussat 1996)

Specification & Computation Models for System Design

- Executable high-level design languages
  - UNITY
  - COLD
- State-based specification languages
  - distributed: Petri Nets
  - sequential: SCR (Parnas Tables), Z/B, VDM
- Virtual Machines
  - Dijkstra's Abstract Machine Concept
  - Active Db
  - Data Flow (Neural) Machines
  - JVM (platform independent machine for programming lg interpretation)
- Stateless Modeling Systems
  - Process Algebras (CSP, LOTOS, etc.)
  - Logic Based Systems (denotational, algebraic, axiomatic)

Comparing ASM and VDM

- VDM restricted to sequential runs
- Abstraction level of VDM fixed
  - for sets by VDM-SL types
    - to be built from basic types by constructors
  - for functions by explicit and implicit definitions
  - for operations by procedures (with side effects)
  - for states by records of read/write variables
- Biased to functional modeling
- VDM-SL has ISO standard & tool support developed by IFAD (Reference: J. Fitzgerald, P. Gorm Larsen: Modelling Systems, Cambridge UP 1998)

Dijkstra's Concept of Abstract Machines

- In 1968, when formulating the T.H.E. operating system, Dijkstra coined the term Abstract Machines with abstract instructions providing local modifications
- The notion of Abstract Machines was preceded and followed by a large number of concrete definitions of such machines
  - Dahl's Simula67 classes, Landin's SECD, Warren's WAM, Java VM
  - IBM's Virtual Machine concept for high-level OS view, hierarchical systems, layered architectures, data spaces
  - VDM, B machines, etc.
- The definition of ASMs conceptually clarifies the underlying general meaning of "abstract instruction" for such machines
  - see sect. 3.1 in E. Börger: High Level System Design and Analysis using Abstract State Machines. Springer LNCS 1641 (1999) 1-43
- All those abstract or virtual machines can be naturally defined as particular ASMs (see some example below)

Active Database Machines

- Rules of form

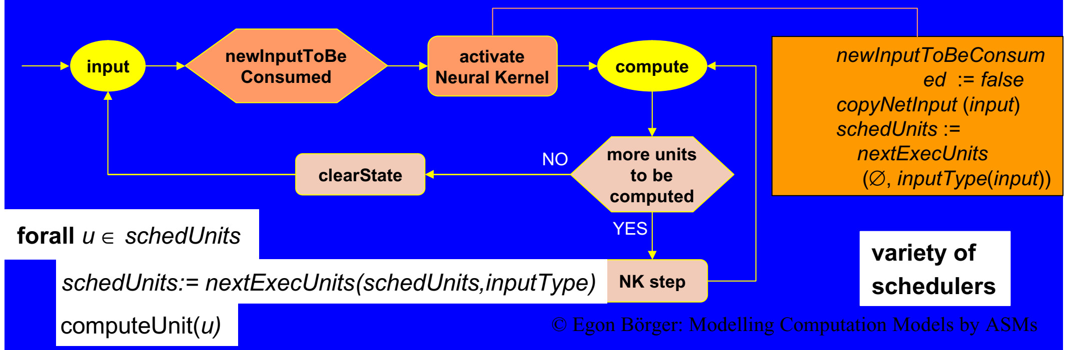
If event & condition Then action

- event : the trigger which may result in firing the rule
- condition : the relevant part of "state" (context) in which an event occurs, must be additionally satisfied for rule execution
- action : the task to be carried out by the database rule
- Different active databases result from varying
  - the underlying notion of state, as constituted by syntax and semantics of events, conditions and actions, and of their relation to the underlying database states
  - the scheduling of the evaluation of condition and action components relative to the occurrence of events (coupling modes, priority declarations, etc.)
  - the rule ordering (if any), etc.

Data Flow Machines: Neural Nets

Börger/Sona JUCS 2001

- A **Neural Net** is usually seen as a black-box yielding output to the env, as result of an internal computation which is triggered by an input taken from the env. The internal computation consists of a finite sequence of atomic actions performed by the basic computing elements (nodes of a directed data-flow graph)
  - In forward propagation mode, the network input is transmitted by the input units to the internal units which propagate their results through the graph until the output units are reached



Data Flow Unit Computation in Neural Nets

computeUnit (u) ≡ **if** *inputType = forward* **then**
  let *result = forwardValue(u)* **in**
    propagateForward (u, result)
    updateLocalStateForward(u, result)
if *inputType = backward* **then**
  let *result = backwardValue(u)* **in**
    propagateBackward(u, result)
    updateLocalStateBackward(u, result)

propagateForward (u, dataToPropagate) ≡
  forall *d* ∈ *dest (u)*
    inForward ^{int} *(d, u) := intValueForw (d, u, dataToPropagate)*
  if *u* ∈ *outputUnits* **then**
    output (u) := extValueForw (u, dataToPropagate)

propagateBackward (u, dataToPropagate) ≡
  forall *s* ∈ *source (u)*
    inBackward ^{int} *(s, u) := intValueBack (s, u, dataToPropagate)*
  if *u* ∈ *inputUnits* **then**
    outputBack(u) := extValueBack (u, dataToPropagate)

Specification & Computation Models for System Design

- Executable high-level design languages
  - UNITY
  - COLD
- State-based specification languages
  - distributed: Petri Nets
  - sequential: SCR (Parnas Tables), Z/B, VDM
- Virtual Machines
  - Dijkstra's Abstract Machine Concept
  - Active Db
  - Data Flow (Neural) Machines
- Stateless Modeling Systems
  - Functional programming paradigm
  - Process Algebras (CSP, LOTOS, etc.)
  - Logic Based Systems (denotational, algebraic, axiomatic)

ASM Model for Functional Programming Features

Theoretical basis: value returning **Turbo ASMs** containing possibly seq, iterate

- Let R(x)=body be a rule definition, actual params a

  $[[\ R(a)\ ]]^A = [[\ body(a/x)\ ]]^A$ Börger/Schmid 2000
- $[[\ l \leftarrow R\ ]]^A = [[\ body(l/result)\ ]]^A$
- Let $y_1=R_1\ (a_1), \ldots, y_n=R_n\ (a_n)$ in S defined as

  Let $l_1,\ldots,l_n$ = new(LOC) in

  forall $1 \le i \le n$ do $l_i \leftarrow R_i\ (a_i)$ seq

  let $y_1=l_1,\ldots,y_n=l_n$ in S

Definition allows to use arbitrary functional equations x=R(a) for value returning subcomputations of R, for parameter a, as standard refinement of an ASM

Example: Turbo ASM Model for Quicksort

Quicksort(L) =

If $|L| \le 1$ then result:=L else

Let

x=Quicksort (tail(L) $_{< head(L)}$)

y= Quicksort(tail(L) $_{\ge head(L)}$)

in

result := concatenate(x,head(L),y)

Example: Turbo ASM Model for Mergesort

Mergesort(L) =

If $|L| \le 1$ then result:=L else

Let x=Mergesort (LeftHalf(L))

y= Mergesort (RightHalf(L))

in result := Merge(x,y)

Merge(L,L') =

If L=[] or L'=[] then result:= (the unique l s.t. ($l \in$ {L,L'} and $l \ne$ []))

elseif head(L) $\le$ head(L') then

let x=Merge(tail(L),L') in result:= concatenate(head(L),x)

elseif head(L') $\le$ head(L) then

let x= Merge(L,tail(L')) in result := concatenate(head(L'),x)

Modeling Process Algebras by ASMs

- Each CSP is a particular multi-agent ASM with
  - agents reacting to events
  - communication
  - non-deterministic choice
- The Occam and Transputer realization of CSP have been modeled by particular ASMs:
  - Succinct ASM model for the realization of CSP by OCCAM

    Börger/Durdanovic/Rosenzweig PROCOMET'94
  - The ASM model for OCCAM has been refined to a proven to be correct ASM model for the compilation of Occam programs to TRANSPUTER code Börger/Durdanovic Computer J.1996
- A general model for process-algebraic concepts within the ASM framework has been given in terms of Abstract State Processes (ASPs) Bolognesi/Börger 2002

UML Activity Diagram for semantics of Occam

Börger/Cavarra/Riccobene LNCS 1816

By ASM model for act dgms only the atomic actions need to be instantiated

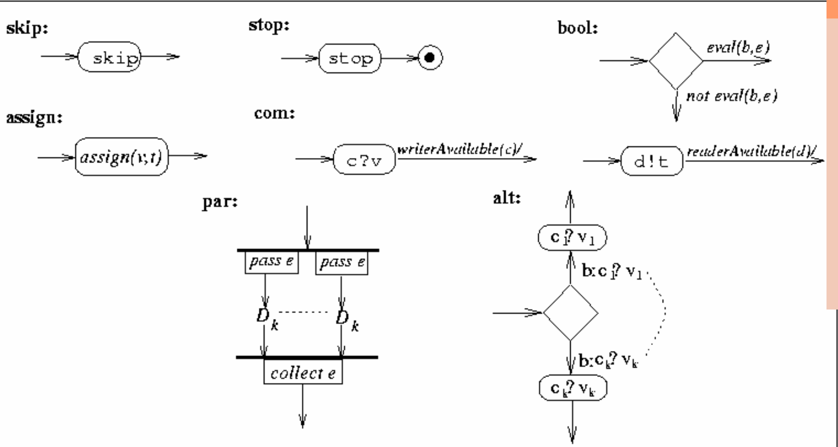
writerAvailable(c) ≡ ∃writer∈Agent ∃n ∈Node:
active(writer) = in(n) & action(n) = d!t
& eval(d,e(writer)) = eval(c,e(self))

readerAvailable(c) ≡ ∃reader∈Agent ∃n ∈Node:
active(reader) = in(n) & action(n) = c?v
& eval(d,e(self)) = eval(c,e(reader))

c?v ≡ e(self) := e(self)[v/ eval(term(writer_c),e(writer_c))]

d!t ≡ skip

b: c?v ≡ eval(b,e(self)) & writerAvailable(c)



assign(v,t) ≡ e:= e[v/ eval(t,e]

pass e ≡ e(self) := e(parent(self))

collect e ≡ e(self):= $\cup_{i<k}$ e(a_i)

ASMs & Logic Based Specification Systems

Every modeling language affects the form of the models (design space), their comprehension, the means for their analysis

ASMs separate design from analysis (Maths: defining ≠ proving)

to avoid premature design decisions ("specify for change", keep design structure open)

ASMs separate validation from verification

- no a priori commitment neither to proof rules nor to specific proof rules
  distinguishing different levels of rigor for system justification
- a posteriori compatibility with any (formal or computerized) proof system
  - PVS verification of ASM based correctness proof (pipelining of DLX , Verifix compiler project)
  - KIV verification of ASM based correctness proof compiling PROLOG programs to WAM code, Java programs in Java Reference Manual, etc
  - Model checking of safety and liveness properties for ASM models (Production Cell, Flash protocol, etc.)
- declarative features can be built into ASMs as assumptions (on state, environment, store, applicability of rules).

Axiomatic/Denotational Specification Methods

- Denotational: program denotation defined by systems of equations (usually inductively, using fixed-point operators)
  - Scott-Strachey, VDM (D. Bjoerner, C. Jones), Monadic Semantics (E. Moggi), Predicate Transformers (E. Dijkstra) & multiple variants (see Action Semantics book and survey by P. Mosses in PSI'01)
- Axiomatic: algebraic (Hoare), dynamic logic (Harel), temporal logic TLA (Lamport), etc.
- Ax/Den approaches mainly tailored for semantics of programming languages, not a general system development method (See survey by P. Mosses Proc. PSI'01)
  - **states** are specialized, namely based upon abstract syntax trees with still to be executed pgm, already computed intermediate values, env, store,… (transition"labels")

Logical Character of Axiomatic/Denotational Spec Methods

Ax/Den approaches follow the pattern of logic: specs typically expressed by systems of axioms and inference rules

- spec perceived as a logical expression or equation
- implementation understood as implication
- composition defined as conjunction

- **Problems:**
  - **frame problem** via declarative nature of axiomatization
  - difficult to control order of rule applications
    - e.g. non-determinism hidden in rule application by user

SOS Specification Methods

- Structural Operational Semantics (Plotkin 1981)
  - tailored for semantics of programming languages (See survey by P. Mosses Proc. CSL'99)
  - transitions specified structurally, with implicit control, by axioms and inference rules (typically of Horn clause like equational, rewriting, tile logic) reflecting the steps of compound phrases in terms of steps of its component phrases
    - "frame rules" expressing that component rules ('small-step') propagate to enclosing structures ('one-hole term contexts', generalized in tile logic to multiple hole contexts)
- Natural Semantics (G. Kahn): inference rules a la Gentzen's sequents calculi for Natural Deduction, involving only initial/final (no intermediate) states ("big-step")
  - Exl: Big-Step Def of Standard ML semantics (Milner et al 1997)

Variants of SOS Specification Methods

- Reduction Semantics: standard term rewriting
  - difficult to control order of traditional reduction steps
    - e.g. by leftmost outermost reduction sequences or by restricting reduction steps to occur with predefined evaluation contexts (Felleisen)
- Rewriting Logic (Meseguer TCS'92): conditional concurrent rewrite rules modulo an equivalence relation over terms
- Modular SOS (Mosses Proc. MFCS'99) with independent transition rules for each language construct
  - relevant state info incorporated into labels α of transition rules $\rightarrow_{\alpha}$ ('semantic entities' treated as 'components of labels', formally as arrows of a category where the labels of adjacent steps are composable)
  - implementation in Maude (executable Rewriting Logic), translating label formulae to equations about the corresponding state, for prototyping AN descriptions of programming languages

Relating Mosses' Action Notation and ASMs

- **AN tailored to support development of programming langs** (not a general purpose sw/hw system design framework, no ground model or refinement notion)
  - enriching denotational with practically useful operational features
    - overcoming pragmatically dissatisfactory aspects of purely denotational approach by directly reflecting (primitive and composed) actions corresponding to programming concepts (semantic mapping of AST to predefined actions)
  - making a compromise between competing language development requirements, corresponding to views of designer, implementer, programmer
- **AN aims at generation of tool env from lang spec**
  - semantics directed generation of interpreters, compilers,…
- Technical comparison: ASM-based Montages spec of AN semantics & implementation of AN in XASM by Anlauff et al '01

Relating Mosses' Action Notation and ASMs

- Actions categorize general ASM function updates and declarations by a classification on the basis of
  - different computational aspects
    - control (seq, par, non-determinism) ("basic facet")
    - data storage
      - transient between actions ("functional facet")
      - stable in cells ("imperative facet")
    - communication describing interactions between distributed agents ("communicative facet")
    - scope information ("declarative facet")
  - types of effect propagation of actions
    - transient (intermediate results), stable (cell data for values of vars), permanent (communication data), scoped (binding tokens to data)
  - types of action performance
    - Execution may complete, escape, fail, diverge
- These features are not directly available (though definable) in ASMs (see Börger/Schmid 2000, Anlauff et al 2001)

Exercise

Describe Schönhage's Storage Modification Machines (SIAM J. Computing 9, 1980) as ASMs using only 0-ary or unary dynamic functions, no static or shared function and only input as monitored function. An SMM has as memory a dynamic graph whose nodes n are named (not necessarily uniquely) by sequences of labels for edges, forming a path from a distinguished center node to n. Besides usual instructions for control (Goto s, If input = i goto s_i (for i=0,1), If n=n' Then s Else s' conditioned by an equality test for node names) and instructions to write output symbols on an output tape, there are two characteristic instructions to create new nodes and to redirect edges between nodes: new (n,e) redirects edge e from (the node named by) n to a new node which is linked (by an edge) to the same nodes n is linked to, set e to n' redirects e to n'.

Every ASM restricted in this way is lock-step equivalent to an SMM (see the article by S. Dexter, P. Doyle, Y. Gurevich in JUCS 3 (4) 1997).

References

- E.Börger and D. Rosenzweig: Mathematical Definition of Full Prolog Science of Computer Programming 24 (1995) 249-286
- E.Börger and R.F.Salamone: CLAM Specification for Provably Correct Compilation of CLP (R) Programs E.Börger (Ed.) Specification and Validation Methods. Oxford University Press, 1995, 97-130 E. Börger, J. Schmid: Composition and Submachine Concepts for Sequential ASMs. Springer LNCS 1862 (2000) 41-60
- 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
- L.M.G. Feijs, H.B.M. Jonkers : Formal Specification and Design Cambridge University Press 1992

References

- M. Anlauff, S. Chakraborty, P.W. Kutter, A. Pierantonio, L. Thiele: Generating an action notation environment from Montages descriptions. Int J. STTT (2001) 3:431-455
- E. Börger: High Level System Design and Analysis using Abstract State Machines. Springer LNCS 1641 (1999) 1-43
- E. Börger: Abstract State Machines: A Unifying View of Models of Computation and of System Design Frameworks Annals of Pure and Applied Logic (2003)
- E.Börger, F.J.Lopez-Fraguas, M.Rodrigues-Artalejo: A Model for Mathematical Analysis of Functional Programs and their Implementations B.Pehrson and I.Simon (Eds.): IFIP 13 World Computer Congress 1994, Vol.I: Technology/Foundations, 410-415

References

- J. Fitzgerald, P. G. Larsen: Modelling Systems Cambridge University Press, 1998
- H.-J. Genrich and K. Lautenbach: System Modeling with High-Level Petri Nets. TCS 13 (1981)
- K. Jensen: Coloured Petri Nets Springer-Verlag 1992
- K. Jensen and G. Rozenberg: High-Level Petri Nets. Theory and Applications. Springer-Verlag 1991
- D. Johnson and L. Moss : Grammar Formalisms Viewed als Evolving Algebras. Linguistics and Philosophy 17 (1994) 537-560
- K. Mani Chandy, Jayadev Misra: Parallel Program Design. A Foundation. Addison Wesley 1988

References

- J. Meseguer: Conditional rewriting logic as a unified model of concurrency. TCS 96 (1) 73-155, 1992
- P. D. Mosses : Action Semantics Cambridge University Press 1992
- P. D. Mosses : Logical Specification of Operational Semantics.
  - BRICS Report Series RS-99-55 (ISSN 0909-0878), Dec'99 and CSL'99 (Springer LNCS 1683), pages 32-49
- P. D. Mosses : The Varieties of Programming Language Semantics And Their Uses. Proc. PSI'01
- W. Reisig: Elements of Distributed Algorithms Springer 1998
- D.Rödding: Modular Decomposition of Automata (Survey). M. Karpinski (Ed): Foundations of Computation Theory. LNCS 158, 1983, 394-412