Egon Börger (Pisa) & Alexander Raschke (Ulm)

Definition of ASMs

Syntax and Semantics

Università di Pisa, Dipartimento di Informatica, boerger@di.unipi.it
Universität Ulm, Abteilung Informatik, alexander.raschke@uni-ulm.de

See Ch. 7 of Modeling Companion
http://modelingbook.informatik.uni-ulm.de

Syntax: ASM program/rule (over given signature Σ)

Update rule: $f(t_1, \ldots, t_n) := t$ is an ASM program

- for every n-ary function symbol $f \in \Sigma$ where $n \geq 0$ and t_i, t are expressions (terms) over Σ
- meaning: evaluate $(t_1, \ldots, t_n, t)$, use the result $(a_1, \ldots, a_n, a)$ to update the interpretation of f at argument $(a_1, \ldots, a_n)$ to value a.

Conditional rule: **if** $Condition$ **then** P **else** Q is an ASM program

- for each Boolean expression $Condition$ and ASM programs P, Q
- meaning: if $Condition$ evaluates to $true$ execute P, otherwise Q.

Block (Par) rule: P **par** Q is an ASM program

- for any ASM programs P, Q
- meaning: execute P and Q in parallel, simultaneously in the given state.

Syntax: let and Macro programs

Let rule: **let** $x = t$ **in** P is an ASM program

- for each expression (term) t and ASM program P
- meaning : evaluate t, assign the computed value to x and then execute P with this value for x ('call by value').

NB. The scope of x is P.

Call (Macro) rule: $Q(t_1, \ldots, t_n)$ is an ASM program

- for every *rule declaration* $Q(x_1, \ldots, x_n) = P$ where
  - P is an ASM program
  - t_i are expressions
  - all free variables in P are among $x_1, \ldots, x_n$
- meaning: execute Q with parameters $(t_1, \ldots, t_n)$ ('call by name').

Syntax: choose and forall programs

Forall rule: **forall** x **with** $Property$ **do** P is an ASM program

- for each Boolean-valued expression $Property$ and ASM program P
- meaning: execute simultaneously every $P(x)$ where x satisfies the $Property(x)$ (in the given state).

NB. The scope of x ranges over $Property$ and P.

Choose rule: **choose** x **with** $Property$ **do** P is an ASM program

- for each Boolean-valued expression $Property$ and ASM program P
- meaning: choose an x satisfying $Property(x)$ and execute with it $P(x)$.

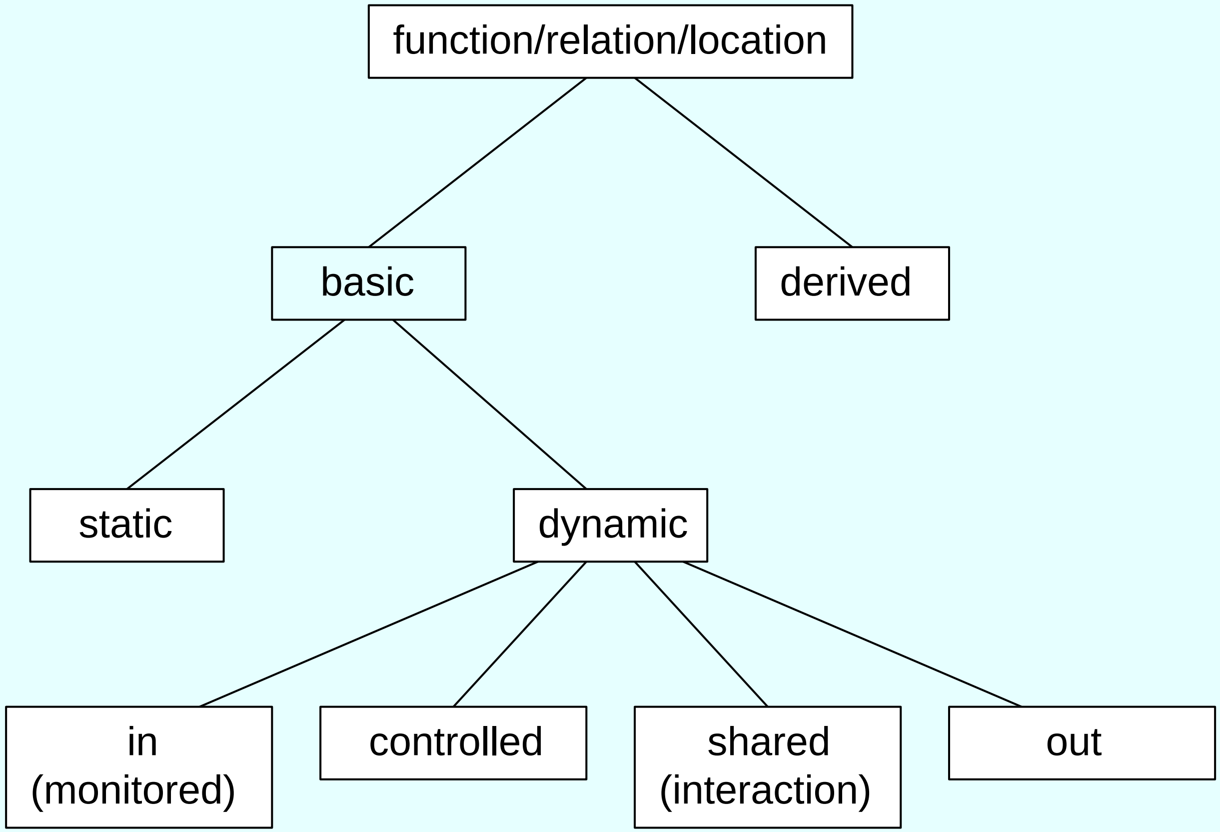
NB. The scope of x ranges over $Property$ and P.

Syntax: signature, ASM

- *Signature* (vocabulary) Σ (of an ASM program) is a set of function symbols f^n of arity $n \geq 0$ (comprising all those which occur in the expressions of the ASM program)
  - including 0-ary functions (*constants*) $true, false, \mathbf{undef}$ (static)
  - possibly including a 0-ary (dynamic) function **self**
  - possibly including a (dynamic) unary function $\mathbf{new}$
- Predicates/Relations are treated as characteristic functions (with values in $\{true, false, \mathbf{undef}\}$)
- Sometimes **skip** is used as ASM program which does nothing

An ASM is defined over a signature by a *main* program (with name of arity 0), a set of rule declarations and a set of initial states.

Classification of functions/locations (of an ASM program)



A 'derived' f has a fixed definition for each $f(x)$. For 'controlled' f, each $f(x)$ can be read and written by and only by the given ASM program P. For 'monitored' resp. 'out' f, $f(x)$ is read-only resp. write-only for P.[1]

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

States (signature interpretation)

- A domain (*superuniverse*) D together with an interpretation of each function symbol f^n in Σ as a function $f_S : D^n \to D$ is called a state S (of the given ASM program)
  – with $true, false, \mathbf{undef}$ interpreted by pairwise distinct elements.

- Expressions t are evaluated in state S in the usual way, denoted by $eval(t, S, env)$.
  – The environment is an interpretation of all free variables (in the given ASM program) by elements of the superuniverse.
  – S and/or env are omitted if they are clear from the context.

- Elements of the superuniverse are also called elements of a state.
- States (the function interpretation) may change, but the superuniverse does not change (see Reserve set below)

State changes by sets of function updates

- a location (in S) is a pair $(f^n, (v_1, \ldots, v_n))$ (*memory unit*)
  - with $f^n \in \Sigma$ and elements v_i (of S)
- $f^n_S(a_1, \ldots, a_n)$ is called *content* of l in S, denoted $S(l)$
  - f is called the function symbol of l, denoted $fctSymbol(l)$
- an update (in S) is a location/value pair (l, v) where
  - $l = (f^n, (v_1, \ldots, v_n))$ is a location (of S)
  - v is an element (in S), -- used as value to update the content of l
- an *update set* is a set of updates
- an update set is *consistent* if it does not contain two updates for the same location (i.e. with different values)
- *firing an update set* U in state S yields the sequel $S + U$ of S whose content of any location l is defined by:

$$S + U(l) = \begin{cases} v & \textbf{if thereissome } (l, v) \in U \\ S(l) & \textbf{if thereisno } (l, v) \in U \end{cases}$$

Structural and elementwise state/memory view

The current state S of a given ASM program is denoted by $currstate$.

- $currstate$ is viewed as a derived function.
- $currstate$ is implicitly parameterized by an ASM program or a program executing agent.

The values of $currstate$ can be seen in two ways:

- structurally: as a family of function tables over D: $(D, (f_S)_{f \in \Sigma})$ – what in logic is called an *algebra* (or Tarski structure)
- elementwise: as the set of all memory units

  $$((f, (v_1, \ldots, v_n)), f_S(v_1, \ldots, v_n))$$

  over D and Σ with their value, i.e. pairs of locations with their content

ASM program semantics: computed update sets

An ASM Program $Yields$ in a State with a given environment (interpretation of its free variables) an Update set recursively:

$Yields(\textbf{skip}, S, env, \emptyset)$ -- **skip** does nothing

$Yields(f(t_1, \ldots, t_n) := t, S, env, \{((f, (v_1, \ldots, v_n)), v)\})$ -- assign
where $v_i = eval(t_i, S, env)$ **and** $v = eval(t, S, env)$

$Yields(\textbf{if } Cond \textbf{ then } P \textbf{ else } Q, S, env, U)$ **if** -- conditional
$Yields(P, S, env, U)$ **and** $eval(Cond, S, env) = true$
or $Yields(Q, S, env, U)$ **and** $eval(Cond, S, env) = false$

$Yields(P \textbf{ par } Q, S, env, U \cup V)$ **if** -- **par** block
$Upd(P, S, env, U)$ **and** $Upd(Q, S, env, V)$

ASM program semantics: let, forall, Call

$Yields($**let** $x = t$ **in** $P, S, env, U)$ **if** -- call by value

$Yields(P, S, env[x \mapsto eval(t, S, env)], U)$

$Yields($**forall** x **with** $Prop$ **do** $P, S, env, \bigcup_{a \in I} U_a)$ **if** -- **forall**

forall $a \in I \;\; Yields(P, S, env[x \mapsto a], U_a)$

where $I = \{a \mid eval(Prop(a), S, env[x \mapsto a]) = true\}$

$Yields(Q(t_1, \ldots, t_n), S, env, U)$ **if**

$Yields(P(x_1/t_1, \ldots, x_n/t_n), S, env, U)$

where $Q(x_1, \ldots, x_n) = P$ is a rule declaration -- call by name

NB. Up to here, U is even a function of P, S, env. There is no non-determinism. So one can write $U = Upd(P, S, env)$ instead of $Yields(P, S, env, U)$.

ASM program semantics: choose (non functional $Yields$)

$Yields($**choose** x **with** $Prop$ **do** $P, S, env, U)$ -- choose

if forsome a **with** $eval(Prop(a), S, env[x \mapsto a]) = true$

$Yields(P, S, env[x \mapsto a], U)$

$Yields($**choose** x **with** $Prop$ **do** $P, S, env, \emptyset)$ -- if no choice do nothing

if forall a $eval(Prop(a), S, env[x \mapsto a]) = false$

ASM Semantics: single-agent ASM run

An ASM with main rule P can make a move (or *step*) from state S (with given env) to the sequel state $S' = S + U$, written $S \Rightarrow_P S'$, if $Yields(P, S, env, U)$ for a consistent set U of updates.

The updates in U are called *internal* to distinguish them from updates of monitored or shared locations; the sequel is called the next internal state.

A run or *execution* of P is a finite or infinite sequence $S_0, S_1, \ldots$ of states (of the signature of P) such that

- S_0 is an initial state,
- for each n
  - either $S_n \Rightarrow_P S'_n$ and $S_{n+1} = S'_n + U$ with a consistent update set U produced by the environment for monitored or shared locations
  - or P cannot make a move in state S_n (i. e. produces an inconsistent update set). In this case S_n is called the last state in the run.

$Reserve$ set and the function $\mathbf{new}$

- $\mathbf{new}\ (X)$ provides a 'fresh' element and makes it an element of X
- 'Fresh' elements come from a (dynamic) $Reserve$ set which contains elements of a state that are not in the domain or range of any basic function of the state.
- Parallel calls of $\mathbf{new}$ are assumed to provide different elements.

The effect of **let** x =$\mathbf{new}\ (X)$ **in** P is often described by:

import x **do**

$X(x) := true$ -- place x into the set X

P -- Execute P with x

with corresponding **import** rules. See AsmBook for details.

Multi-Agent ASM: Syntax

A multi-agent ASM $\mathcal{M}$ is a family

$$(ag(p), pgm(p))_{p \in Process}$$

of single-agent ASMs consisting of a set of $Process$es p viewed as

- agents $ag(p)$ which execute step by step (‘sequentially’)
- each its ASM program $pgm(p)$ (of signature Σ_p)
- interacting with each other via reading/writing in designated (shared or input/output) locations.

$ag : Process \rightarrow Agent$, $pgm : Process \rightarrow AsmRule$ may be dynamic.

Atomic reads/writes in concurrent ASM runs

- A single agent ASM
  - performs in each state S_n of a run both, reads and writes, as one read&write step (one atomic action) resulting in the sequel state S'_n
  - is synchronized with its environment which (in one atomic step) updates S'_n to the next state S_{n+1} in the run.
- In concurrent ASM runs, different agents
  - may perform their read/write actions asynchronously, reading in one state and writing to another state, each agent at its own speed,
  - interact via reads/writes of *interaction* (i.e. in/shared/out) locations.
- Thus we emulate an atomic read&write step of $pgm(p)$ by a program $\textsc{ConcurStep}(pgm(p))$ to perform either directly this atomic read&write step of $pgm(p)$ or three consecutive atomic actions:

  read&SaveGlobalData, LocalWriteStep, WriteBack

  which in a concurrent run may happen asynchronously, in different states (at different moments of time).

Multi-Agent ASM: Semantics

A concurrent run of a multi-agent ASM $\mathcal{M}$ is

- a sequence $(S_0, P_0), (S_1, P_1), \ldots$ of states S_n, subsets $P_n \subseteq Process$
- such that each state S_{n+1} is obtained from S_n by applying to it all the updates computed by any process $p \in P_n$
  - formally $S_{n+1} = S_n + \bigcup_{p \in P_n} U_p$ where for given *env*ironment $Yields(\textsc{ConcurStep}(pgm(p)), S_n, env, U_p)$ holds.

The run terminates in state S_n if the updates computed by the agents in P_n are inconsistent.

NB. We define $\textsc{ConcurStep}(pgm(p))$ such that each of its possible substeps (which together emulate one read&write step of $pgm(p)$), when executed by $p \in P_n$, is an atomic single-agent read&write step in S_n.

NB. The signature of states is the union of Σ_p for all $p \in Process$.

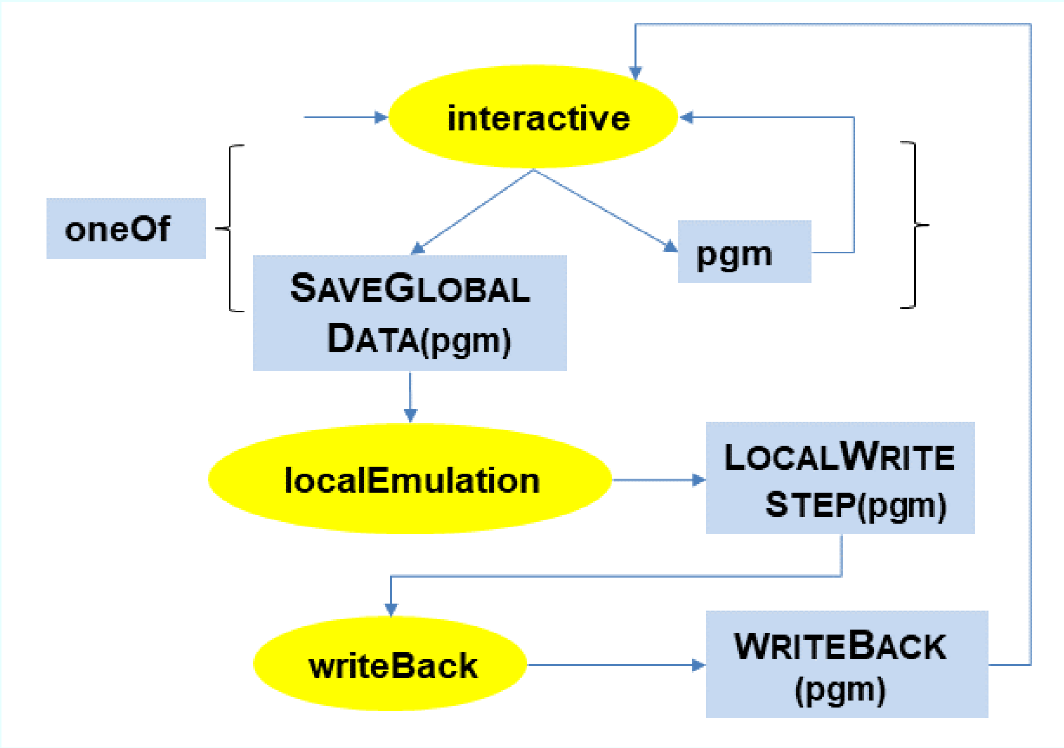
Interactive and local states in concurrent ASM runs

When $p \in P_n$ contributes to build S_{n+1} by executing one of the $\textsc{ConcurStep}$s of $pgm(p)$, it is in one of three (resp.two) modes:

- in *interactive* mode p reads in state S_n the data needed to perform the step described by the given $pgm(p)$. To build S_{n+1} out of S_n:
  - either p directly computes its update set U_p, in S_n, and applies it to S_n, possibly updating some interaction locations
  - or p does $\textsc{SaveGlobalData}$ locally and switches to locally compute U_p, updating only local locations
- in *localEmulation* mode p computes a local copy of U_p, using the previously saved global data, and switches to $\textsc{WriteBack}$ to interaction locations, updating only local locations
- in *writeBack* mode p will $\textsc{WriteBack}$ to those (globally visible) interaction locations whose values it has updated locally (by executing an assignment $f_p(s) := t$ in its preceding $mode = localEmulation$).

NB. In the 2-step version writeBack mode is suppressed.

Three-step version of $\textsc{ConcurStep}(pgm)$



$\textsc{LocalWriteStep}(pgm)$ results from replacing in pgm

- every in/shared/out function symbol f by a new local function symbol f_p, where $p = ag(pgm)$ (used to locally $\textsc{SaveGobalData}$)
- adding $\textsc{Insert}(updData(f_p, s, t), GlobalUpd)$ in parallel to each $f_p(s) := t$ (used to define $\textsc{WriteBack}$ to shared/out locations)

Two-step version of $\textsc{ConcurStep}(pgm)$

choose $M \in \{\textsc{Read\&WriteStep}(pgm), \textsc{ReadStep}(pgm)\}$

M

if $mode = localEmulation$ **then**

$\textsc{LocalEmulation}(pgm) \quad mode := interactive$

where

$\textsc{Read\&WriteStep}(pgm) =$ -- NB. $mode$ does not change

if $mode = interactive$ **then** pgm

$\textsc{ReadStep}(pgm) =$ -- NB. $mode$ change leads to write step

if $mode = interactive$ **then**

$\textsc{SaveGlobalData}(pgm) \quad mode := localEmulation$

$\textsc{LocalEmulation}(pgm) =$ -- emulation of write step

$\textsc{LocalWriteStep}(pgm)$ **seq** $\textsc{WriteBack}(pgm)$

NB. Turbo ASM operator **seq** guarantees atomic 1-step execution.

Submachines of $\textsc{ConcurStep}(p)$

$\textsc{SaveGobalData}(pgm)$ and $\textsc{WriteBack}(pgm)$ transfer values between the globally visible interaction functions and their local copies.

- $\textsc{SaveGobalData}(pgm)$ copies the current values of monitored and shared function terms $f(t)$ of pgm into a local copy $f_p(t)$ which is controlled by p ($p = \textbf{self} = ag(pgm)$):

  $\textsc{SaveGobalData}(pgm) = \textbf{forall } f \in Monitored \cup Shared \; f_p := f$

  NB. $f := g$ abbreviates **forall** $args \; f(args) := g(args)$
- $\textsc{WriteBack}(pgm)$ is the inverse copying, of the just updated local values for output and shared function terms, back to the 'global' terms in pgm (NB. $GlobalUpd$ is local, controlled by $p = \textbf{self}$):

  $\textsc{WriteBack}(pgm) =$

  $\quad \textbf{forall } updData(f_p, s, t) = ((f_p, args), val) \in GlobalUpd$

  $\quad\quad f(args) := val$

  $\quad GlobalUpd := \emptyset$ -- $GlobalUpd$ assumed to be initially empty

Ambient ASMs

- Syntax extension: **amb** exp **in** P is an ASM program
  – for each expression and ASM program P
- Function classification extension:
  $AmbDependent(f)$ **iff**
  forsome e, e' **with** $e \neq e'$ **forsome** x $f(e, x) \neq f(e', x)$
  Otherwise f is called $AmbIndependent$.
- $eval$ extension by ambient parameter: see below
- Semantics extension:
  to avoid a signature blow up by dynamic ambient nesting, we treat amb as a stack where new ambient expressions are $push$ed (passed by value):
  $Yields($**amb** exp **in** $P, S, env, amb, U)$ **if**
  $Yields(P, S, env, \text{PUSH}(eval(exp, S, env, amb), amb), U)$

Ambient $eval$uation function

$eval$uation function extension by ambient parameter:

Case AmbDependent(f):

$$eval(f(t_1, \ldots, t_n), S, env, amb) =$$
$$f_S(amb, eval(t_1, S, env, amb), \ldots, eval(t_n, S, env, amb))$$

Case AmbIndependent(f):

$$eval(f(t_1, \ldots, t_n), S, env, amb) =$$
$$f_S(eval(t_1, S, env, amb), \ldots, eval(t_n, S, env, amb))$$

NB. Since often the interpreation env of free variables is omitted, the ambient is called the environment.

References

- E. Börger and A. Raschke: Modeling Companion for Software Practitioners. Springer 2018
  `http://modelingbook.informatik.uni-ulm.de`
- E. Börger and K.-D.Schewe: Concurrent Abstract State Machines. *Acta Informatica* 2016, 53 (5) 469–492. Listed as Notable Article in:
  ACM 21th Annual BEST OF COMPUTING
  `www.computingreviews.com/recommend/bestof/notableitems.cfm?bestYear=2016`
- E. Börger and R. Stärk: Abstract State Machines. Springer 2003.

Copyright Notice