Egon Börger: From ASMs to Abstract State Processes

or: How to marry different granularities of *computation step*

Starting Point: Benefits of ASM Method

- Provides accurate yet practical industrially viable process oriented semantics for pseudocode on arbitrary data structures, namely by clear notion of
  - (global) state
  - state transition (next-step through atomic actions)

  tunable to desired level of abstraction, with well defined interfaces (rigor without formal overkill)
- The parallel ASM execution model
  - easens specification of macro steps (refinement and modularization)
  - avoids unnecessary sequentialization of independent actions
  - easens parallel/distributed implementations

How to marry parallel execution with non-atomic substeps

The Problem: How to incorporate non-atomic structuring concepts

- **seq** (sequentialization) and finite iteration
- submachines
  with
  - return values,
  - exception handling,
  - local values, etc.

as standard refinements into synchronous parallel ASMs.

First idea: Black-Boxing of Submachines

Turn white box view "first P then Q" (and in general of submachines) into black box view so that P **seq** Q (in general the submachines) become executable in parallel with other atomic actions.

Idea: Use that the effect of a single ASM step is described by a set of updates which are executed in parallel.

Solution: Collect the cumulative effect of updates of multiple successive steps into ONE update set (with possible overwriting at some locations in case of sequential steps).

See E. Börger and J. Schmid: *Composition and Submachine Concepts for Sequential ASMs*. Proc. CSL'2000, Springer LNCS 1862

Ch.4.1 of *AsmBook* (E. Börger and R. Stärk: Abstract State Machines. A Method for High-Level System Design and Analysis. Springer 2003)

Second idea: White-Boxing of Submachines via Interleaving

Approach: Combining most general *states* of ASMs with process-algebraic *structuring techniques* and *concurrency patterns* to decompose complex systems into concurrently interacting simpler component processes.

Two main concepts:

- *interleaving*: scheduling principle which allows one to view multiple-agent processes as executed by one agent at a time
- *selective synchronization* which allows interleaved processes to act simultaneously (in parallel) if and only if they all 'share a selected event' (read: contribute to a selected update)

ASP state transformation

We combine the traditional ASM state transformation—via firing of update sets—with the structured programming approach: *program* evolution made explicit as *part of the dynamic state* (instead of describing it as a walk through the abstract syntax tree).

Therefore we describe for each ASP construct P which pair (P', U) of

- update set U
- residual program P'

it yields in an arbitrary state $\mathfrak{A}$ to form—by firing U—the next state in which the residual program P' has to be executed.

Realization: we provide inductively for each ASP expression P one or two inference rules which define how to derive statements of form yields$(P, \mathfrak{A}, P', U)$.

Long living variable incarnations

ASM: call-by-value discipline for **let** $x = t$ **in** P (atomic execution).

ASP: program P in **let** $x = t$ **in** P may lead in one step to a residual program P' which involves further steps. Thus the incarnation x' of the variable x with its interpreation by the value t has to survive until the residual process has become empty.

Therefore we use for each execution of a constructor $c(x)\ P$ in a given state a fresh instance of x, say x', standing for a new 0-ary function which records for the entire execution of the constructor body P the value assigned in the given state to the parameter x.

View states $\mathfrak{A}$ as sets of pairs (l, v) of locations l and their values v. This allows one to identify free variables with 0-ary function symbols (parameterless locations) so that their interpretation is incorporated into state $\mathfrak{A}$.

Semantics of standard ASP rules

$$\frac{}{\mathsf{yields}(\textbf{skip}, \mathfrak{A}, \mathbf{nil}, \emptyset)}$$

$$\frac{}{\mathsf{yields}(f(s_1, \ldots, s_n) := t, \mathfrak{A}, \mathbf{nil}, \{((f, ([\![s_1]\!]^{\mathfrak{A}}, \ldots, [\![s_n]\!]^{\mathfrak{A}})), [\![t]\!]^{\mathfrak{A}})\})}$$

$$\frac{\mathsf{yields}(P, \mathfrak{A}, P', U)}{\mathsf{yields}(\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q, \mathfrak{A}, P', U)} \quad \text{if } [\![\varphi]\!]^{\mathfrak{A}} = \mathit{true}$$

$$\frac{\mathsf{yields}(Q, \mathfrak{A}, Q', V)}{\mathsf{yields}(\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q, \mathfrak{A}, Q', V)} \quad \text{if } [\![\varphi]\!]^{\mathfrak{A}} = \mathit{false}$$

$$\frac{\mathsf{yields}(P\frac{x'}{x}, \mathfrak{A}[x' \mapsto a], P', U)}{\mathsf{yields}(\textbf{let } x = t \textbf{ in } P, \mathfrak{A}, P', U \cup \{(x', a)\})} \quad a = [\![t]\!]^{\mathfrak{A}},\ x' \text{ fresh}$$

Semantics of standard ASP rules (Cont'd)

$$\frac{\text{yields}(P\frac{y_a}{x}, \mathfrak{A}[y_a \mapsto a], P_a, U_a) \quad \text{for each } a \in I = range(x, \varphi, \mathfrak{A})}{\text{yields}(\textbf{par}\ \{P(x) \mid \varphi(x)\}, \mathfrak{A}, \textbf{par}\ \{P_a \mid a \in I\}, \bigcup_{a\in I} U_a \cup \{(y_a, a)\})} \quad y_a \text{ fresh}$$

$$\frac{\text{yields}(P\frac{x'}{x}, \mathfrak{A}[x' \mapsto a], P', U) \quad \text{for some } a \in I = range(x, \varphi, \mathfrak{A})}{\text{yields}(\textbf{choose}\ \{P(x) \mid \varphi(x)\}, \mathfrak{A}, P', U \cup \{(x', a)\})} \quad x' \text{ fresh}$$

$$\frac{\text{yields}(P, \mathfrak{A}, P', U)}{\text{yields}(P\ \textbf{then}\ Q, \mathfrak{A}, P'\ \textbf{then}\ Q, U)} \quad \text{if } P' \neq \textbf{nil}$$

$$\frac{\text{yields}(P, \mathfrak{A}, \textbf{nil}, U)}{\text{yields}(P\ \textbf{then}\ Q, \mathfrak{A}, Q, U)}$$

$$\frac{\text{yields}(P\frac{t_1\cdots t_n}{x_1\cdots x_n}, \mathfrak{A}, P', U)}{\text{yields}(r(t_1, \ldots, t_n), \mathfrak{A}, P', U)} \quad r(x_1, \ldots, x_n) = P \text{ rule declaration}$$

Semantics of if $Cond$ then R

Distinguish two interpretations of **if** $Cond$ **then** R

- *persistent* version: defined as **if** $Cond$ **then** R **else** $\mathbf{nil}$. It has a blocking effect in case $Cond$ is false since no inference rule is provided to execute the empty program $\mathbf{nil}$.
- *transient* version: defined as **if** $Cond$ **then** R **else skip** which in case $Cond$ is false uses the inference rule for **skip**.

We use the persistent version as the default.

See the blocking evaluation of guards e.g. in the high-level design language COLD, whereas in ASMs the rule of a 'blocked' process does not prevent other rules from being executed in parallel.

Semantics of interleaving and selective synchronization

$$\frac{\text{yields}(P\frac{y_a}{x}, \mathfrak{A}[y_a \mapsto a], P_a, U_a) \quad \text{for some } a \in I = range(x, \varphi, \mathfrak{A})}{\text{yields}(\mathbf{intlea}\ \{P(x) \mid \varphi(x)\}, \mathfrak{A}, \mathbf{intlea}\ \{P_b \mid b \in I\}, U_a \cup \bigcup_{b \in I}\{(y_b, b)\})}$$

where y_c fresh for all $c \in I$, $P_b = P\frac{y_b}{x}$ for $b \in I \setminus \{a\}$

$$\frac{\text{yields}(P\frac{y_a}{x}, \mathfrak{A}[y_a \mapsto a], P_a, U_a) \quad Loc(t)^{\mathfrak{A}} \in Loc(U_a) \text{ for each } a \in I}{\text{yields}(\mathbf{sync}\ (t)\{P(x) \mid \varphi(x)\}, \mathfrak{A}, \mathbf{sync}\ (t)\{P_a \mid a \in I\}, \bigcup_{a \in I} U_a \cup \{(y_a, a)\})}$$

where I$=range(x, \varphi, \mathfrak{A})$, y_a fresh for all $a \in I$

$$\frac{\text{yields}(P\frac{y_a}{x}, \mathfrak{A}[y_a \mapsto a], P_a, U_a) \quad Loc(t)^{\mathfrak{A}} \notin Loc(U_a) \text{ for some } a \in I}{\text{yields}(\mathbf{sync}\ (t)\{P(x) \mid \varphi(x)\}, \mathfrak{A}, \mathbf{sync}\ (t)\{P_b \mid b \in I\}, U_a \cup \bigcup_{b \in I}\{(y_b, b)\})}$$

where $I = range(x, \varphi, \mathfrak{A})$, y_c fresh for all $c \in I$, $P_b = P\frac{y_b}{x}$ for $b \in I \setminus \{a\}$

Notational Conventions and Remarks

Whenever the application of an inference rule leads to a residual program $oper\{P\}$, where $oper$ is any of the operators and $\{P\}$ is a singleton set of processes, the residual program is considered as automatically rewritten into P.

$oper\{\}$ is rewritten into $\mathbf{nil}$.

$range(x, \varphi, \mathfrak{A})$ of a formula φ with distinguished variable x in a state $\mathfrak{A}$ denotes the set of all elements a of $\mathfrak{A}$ that make the formula true if x is interpreted by a.

We write **par** $\{P(x) \mid \varphi(x)\}$ instead of **forall** x **with** φ **do** P.

The submachine call defined above is by reference. The call-by-value version can be defined using the **let**-construct.

Local sequential scheduling of subprocesses

Modular scheme for scheduling the execution of rules belonging to a set $\mathfrak{R}$ which may even be dynamic.

$\textsc{Scheduling}(\mathfrak{R}, scheduler) =$

par $\{R(select), select := scheduler(\mathfrak{R}, select)\}$

Example 1. Make $select$ behave as interleaving.

Example 2. Round Robin: defined for any number n of processes by

$$scheduler(select) = select + 1 \bmod n$$

Combining interleaving with white-box SEQ

Let $\mathfrak{R}$ consist of processes R of form

$\mathrm{R} = First_R$ **then** $Second_R$

constrained to be interleaved such that the order of execution of the subprocesses $Second_R$ is determined by the order of execution of the subprocesses $First_R$.

For a local realization of such a scheduling we equip each process **self** with its instance **self**.$ticket$ of a location $ticket$ which keeps track of the order in which the interleaving operator chooses the subprocesses $First_R$ for execution.

This allows one to

- locally record by a copy of the current $ticket$ value 'when' **self** has been called to start the execution of its first subprocess,
- locally advance $ticket$ to the next free position in the dynamic ordering of calls of subprocesses $First_R$.

ASP scheme combining interleaving with white-box SEQ

$\textsc{LocalSeqSchedule}(R) = \ First'_R$ **then** $Second'_R$ **where**

$First'_R =$**par** $\{First_R, \textsc{GetAndAdvanceOrderPos}(R)\}$

$\textsc{GetAndAdvanceOrderPos}(R) =$

par $\{R.ticket := ticket, ticket := ticket + 1\}$

$Second'_R =$ **if** $displayed(R.ticket)$ **then**

par $\{Second_R, \textsc{AdvanceDisplay}(R)\}$

$displayed(Ticket) \ = \ (Ticket = display)$

$\textsc{AdvanceDisplay}(R) = (display := display + 1)$

Handshaking

Process communication via rendez-vous (handshaking) is often specified via so-called *gates* at which the participating agents have to 'agree on offered values'.

This is really a special case of shared memory communication, namely via gate locations g shared for reading and/or writing.

φ, ψ determine the choice the processes may have for agreeing upon a consistent update of gate g with a value determined by terms s, t:

$\textsc{Handshaking}(P, \varphi, s, Q, \psi, t) = \mathbf{sync}\ (g)\{R, S\}$ **where**

$R =$ **choose** x **with** $\varphi(x)$ **in** $(g := s(x)$ **then** $P(x))$

$S =$ **choose** y **with** $\psi(y)$ **in** $(g := t(y)$ **then** $Q(y))$

References

T. Bolognesi and E. Börger, Abstract State Processes.

In: E. Börger and A. Gargantini and E. Riccobene (Eds): Abstract State Machines 2003–Advances in Theory and Applications. Springer LNCS 2589, 2003

Ch.4.2 of *AsmBook* (E. Börger and R. Stärk: Abstract State Machines. A Method for High-Level System Design and Analysis. Springer 2003)

