Abstract State Processes

Tommaso Bolognesi[1] and Egon Börger[2]

[1] CNR, ISTI, Pisa bolognesi@ei.pi.cnr.it
[2] Università di Pisa, Dipartimento di Informatica, I-56125 Pisa, Italy
boerger@di.unipi.it

Abstract. Process-algebraic languages offer a rich set of structuring techniques and concurrency patterns which allow one to decompose complex systems into concurrently interacting simpler component processes. They abstract however almost entirely from a notion of system state. The method of Abstract State Machines (ASMs) offers powerful abstraction and refinement techniques for specifying system dynamics based upon a most general notion of structured state. The evolutions of the state are governed however by a fixed and typically unstructured program, called 'rule', which describes a set of abstract updates occurring simultaneously at each step (synchronous parallelism). We propose to incorporate into one machine concept the advantages offered by both structuring techniques, and introduce to this purpose Abstract State Processes (ASPs), i.e. evolving processes (extended ASM programs which are structured and evolve like process-algebraic behaviour expressions) operating on evolving abstract states the way traditional ASM rules do.

1 Introduction

This work has been motivated by the observation that both the Process Algebra approach and the Abstract State Machine approach to system specification—one based on evolving behaviour and the other on evolving state ('evolving algebra')—offer each its own advantages in terms of natural expressiveness and flexible support for design and analysis when applied to the proper aspects of the system under investigation. The prospects for a fruitful coupling of the two approaches seem to be good.

For foundational as well as for historical reasons (see [7] for the details) basic ASMs as they originally appeared in [14] are characterized by a single-agent, synchronous, parallel execution model where the program is fixed and unstructured, namely coming as a set of simultaneously executed rules of form

if *Condition* **then** *Updates*

with a set *Updates* of simultaneously executed function updates $f(s_1, \ldots, s_n) := t$. A large number of successful applications surveyed in [4, 5] worked with this model of computation which, extended by the **forall** construct, is also at the basis of the recently obtained proof for the synchronous parallel case of the ASM thesis from a small number of postulates [2]. Basic ASMs offer besides

their synchronous atomic actions in a global state no other execution control structure, in particular no sequencing, no iteration, no submachine call concept. The same phenomenon can be observed for Abrial's notion of Abstract Machines [1] which come as non-executable pseudo-code without sequencing or loop.

In contrast, process algebra descriptions of systems are focussed on the evolution of structured programs, called behaviour expressions, which are built up from subexpressions in a compositional way to support modular specifications and structured proof methods. A process algebraic specification describes the event patterns an observer can experience at the interface of the described system, as well as the interactions among system components, namely as sequence of behaviour expressions $B_0, B_1, \ldots$ where B_i is transformed into B_{i+1} by the occurrence of an atomic internal or an observable (inter-)action. The variety of expressive means for the decomposition of systems into simpler component processes which concurrently communicate and interact contrasts with the absence of a notion of structured state (except for instantiations of process parameters).

We explore here the idea of specifying concurrent systems by Abstract State Processes (ASPs), i.e. evolving processes whose 'steps' make abstract states evolve the way basic ASM rules do. Since stepping through the program of an ASP becomes part of the overall system evolution, the single steps of a sequential execution, of an iteration and of a submachine execution become visible. This white-box view differs from the black-box view of these concepts which has been realized by turbo ASMs in [9] for an integration of standard programming constructs, including recursion [8], into the synchronous parallelism of basic ASMs, and has been successfully applied in [21] for modelling and analysing the semantics of Java and its implementation on the Java Virtual Machine. The white-box view we define here naturally leads to equip ASPs with two more constructs:

- *interleaving*, which allows parallel processes to act independently from each other, one at a time,
- *selective synchronization* (see [3]), which allows interleaved processes to act in parallel if and only if they all 'share a selected event' (read: contribute to a selected update).

Our definitions are tailored not to involve asynchronous ASMs[1]. With the synchronous form of parallelism which is characteristic for basic or turbo ASMs, the difference betweeen mono-agent and multi-agent systems does not constitute a semantical difference in the underlying computation model; although it matters if agents are used to define disjoint name spaces for a separation of state components. From that point of view interleaving represents a scheduling principle which allows one to view multiple-agent processes as executed by one agent. Therefore the Abstract State Processes we are going to define naturally lead to mono-agent machines. We hope that the integration of interleaving and selective synchronization into the synchronous parallelism of ASMs will provide a useful guideline for extending current implementations of turbo ASMs to an implementation of asynchronous multi-agent ASMs.

[1] Passing from multi-agent synchronous to asynchronous ASMs adds considerable complexity to the notion of runs, defined at the end of [15] under the name of

The program evolution of an ASP could be described as a walk through the abstract syntax tree of the behaviour expression, following the scheme used for modeling the execution of Java programs in [21]. Instead we adopt the structured programming approach to make the process-algebraic constructs for communication and synchronization of processes stand out explicitly. In doing so we are not primarily concerned with the minimality and orthogonality requirements which have guided the definition of many process-algebraic systems as well as of basic ASMs. Our choice of constructs is mostly led by the pragmatic needs of system and software engineering applications, which require a balance between conceptual simplicity and manipulative ease.

In this paper we provide only the basic definitions. A further analysis has to be postponed to a future occasion. In the last section we formulate some questions we consider worth to be investigated. We suppose the reader to have at least some elementary knowledge of the fundamental intuitions of process algebra (see [16, 18–20]) and of ASMs (see [14]). For a succinct definition of the extension of basic ASMs to ASPs we use the notational framework of the ASM book [10].

2 Semantics of Standard Abstract State Processes

We define here the extension of basic ASMs by process-algebraic concepts of evolving programs (usually called behaviour expressions) providing, in addition to basic ASM features, white-box sequencing and submachine execution. In the next section we add to this interleaving and selective synchronization.

We preserve the state transformation mechanism of basic ASMs whereby to define the semantics of a construct it suffices to define the update set U it yields in an arbitrary state $\mathfrak{A}$. To this we add the definition of the remaining (called the *residual*) program P the given construct leaves for further execution. Thus we describe for each ASP construct P which pair (P', U) of residual program P' and update set U it yields in an arbitrary state $\mathfrak{A}$. We do this following the inductive definition of the syntax of ASPs, providing for each ASP expression P one or two inference rules which define how to derive statements of form $\text{yields}(P, \mathfrak{A}, P', U)$. We concentrate our attention here upon the definition of the semantics of ASPs, given that the extension of the syntax of ASMs to that of ASPs is a routine matter.

For a better overview we put the definition into two tables. Table 1 is adapted from [9, 10] for the constructs defining turbo ASMs, namely **skip**, assignment, conditional expressions, **let**, **par**, **choose**, **forall**, sequencing and parameterized submachine call. In adapting the definition of turbo ASMs to ASPs, besides integrating the program evolution we replace the black-box view of sequential

'distributed' runs. On the theoretical side, no natural postulates are known to derive for asynchronous ASMs an analogon to the synchronous parallel ASM thesis of [2]. We therefore advice to use asynchronous ASMs only where really appropriate. See in this context also [8].

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

$$\frac{}{\text{yields}(f(s_1, \ldots, s_n) := t, \mathfrak{A}, \mathbf{nil}, \{((f, (\llbracket s_1 \rrbracket^{\mathfrak{A}}, \ldots, \llbracket s_n \rrbracket^{\mathfrak{A}})), \llbracket t \rrbracket^{\mathfrak{A}})\})}$$

$$\frac{\text{yields}(P, \mathfrak{A}, P', U)}{\text{yields}(\mathbf{if}\ \varphi\ \mathbf{then}\ P\ \mathbf{else}\ Q, \mathfrak{A}, P', U)} \quad \text{if } \llbracket \varphi \rrbracket^{\mathfrak{A}} = \mathit{true}$$

$$\frac{\text{yields}(Q, \mathfrak{A}, Q', V)}{\text{yields}(\mathbf{if}\ \varphi\ \mathbf{then}\ P\ \mathbf{else}\ Q, \mathfrak{A}, Q', V)} \quad \text{if } \llbracket \varphi \rrbracket^{\mathfrak{A}} = \mathit{false}$$

$$\frac{\text{yields}(P\frac{x'}{x}, \mathfrak{A}[x' \mapsto a], P', U)}{\text{yields}(\mathbf{let}\ x = t\ \mathbf{in}\ P, \mathfrak{A}, P', U \cup \{(x', a)\})} \quad \begin{array}{l} a = \llbracket t \rrbracket^{\mathfrak{A}} \\ x' \text{ fresh} \end{array}$$

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

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

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

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

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

Table 1. Inductive definition of the semantics of standard ASP rules

execution and submachine calls defined in [9] by the corresponding white-box ASP constructs. For white-box sequencing we write **then** instead of **seq**.

The execution of **skip** and assignment processes yields the empty residual process **nil**. A conditional process as defined in Table 1 blocks other processes with which it may be synchronized if in case the condition is true its subprocess P, or otherwise Q, cannot proceed. Therefore one has to distinguish two interpretations of **if** *Cond* **then** R. The *persistent* one is defined as **if** *Cond* **then** R **else nil**; it has a blocking effect in case *Cond* is false since no inference rule is provided to execute the empty program **nil**. The *transient* version is 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. One may compare this with the blocking evaluation of guards e.g. in the high-level design language COLD [12], whereas in ASMs the rule of a 'blocked' process does not prevent other rules from being executed in parallel.

For the formulation of processes which involve the manipulation of logical variables it is notationally convenient to view states $\mathfrak{A}$—structures with finitely many domains, functions and relations—as sets of pairs (l, v) of locations l and their values v. Locations are pairs (f, a) of a function name f (say of arity n) and a sequence $a = (a_1, \ldots, a_n)$ of arguments in the domain of f. We identify

free variables with 0-ary function symbols (parameterless locations) so that their interpretation is incorporated into $\mathfrak{A}$[2]. Thus we write $\mathfrak{A}[x' \mapsto a]$ for the extension of $\mathfrak{A}$ by the new location x' with value a.

In the **let**-construct local variables appear to which precomputed values are assigned. In basic or turbo ASMs this preliminary computation step remains implicit as part of the unique computation step associated to the machine **let** $x = t$ **in** P, whose execution implies a form of sequentialization which is typical for the call-by-value discipline, namely to first compute t in the given state and then to execute P with the computed value recorded in the local variable x. These variables are called logical because they are not updated and their binding to the current value of t holds only for the atomic execution of P. Since in ASPs such a program P is not executed atomically, but may lead after one step to a residual program P' which involves further steps, the incarnation x' of the variable x with its interpreation by the value t has to survive until the residual process has become empty (reduced to **nil**). This holds analogously also for the other ASP constructors.

We therefore 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. Formally this makes the signature of ASPs dynamic, though the dynamics is restricted to creating new incarnations of local variables, whereas in traditional ASMs the signature is static. By imposing the condition of x' being fresh (meaning by this that it is sufficiently new not to be mixed up with any other variable[3], namely that is not used before and not simultaneously anywhere else) we guarantee that the currently determined value for x will not collide with any other value determined in a different process or in a different step for the same parameter x ('same' in the syntactic sense). For brevity we write x, although we allow it to denote a tuple of parameters.

For notational uniformity we write **par** $\{P(x) \mid \varphi(x)\}$ instead of **forall** x **with** φ **do** P. In case φ evaluates to finitely many elements, **par** $\{P_1, \ldots, P_n\}$ stands for **par** $\{P(x) \mid \varphi(x)\}$ where $\{P(x) \mid \varphi(x)\} = \{P_1, \ldots, P_n\}$. This avoids having to fuss with fixed parameter instances. Similarly for **choose** $\{P(x) \mid \varphi(x)\}$ and for the synchronization and interleaving operators **sync** (t), **intlea** defined in the next section. See Table 3 for a summary of some alternative notations borrowed from the process-algebraic literature. When applying ASP

[2] In [9, 10] this interpretation is indicated by a separate environment function ζ, following a widespread notational practice of mathematical logic.

[3] See [10, Ch.2] for a simple ASM characterization of this use of the **import**-construct applied to the reserve set. That characterization also covers our usage of fresh local variables x', avoiding to introduce a new inference rule into the semantical description of ASPs. One can use a similar expedient to the one introduced in [8] to separate the 'logical' (read-only) variables from the variables for locations (write-variables). It suffices to write $varIncar(x')$ instead of x' with a monadic function *varIncar* which takes variable incarnations as arguments; to guarantee that different incarnations of x are stored in different locations it suffices to pass to *varIncar* different arguments x', x''.

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

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

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

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

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

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

Table 2. Semantics of interleaving and selective synchronization

operators to sets of processes, it is notationally convenient to assume the following implicit transformation which will not be mentioned furthermore: whenever the application of an inference rule leads to a residual program $\mathit{oper}\{P\}$, 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. Similarly, $\mathit{oper}\{\}$ is rewritten into **nil**. By $\varphi\frac{t}{x}$ we denote the result of replacing all free occurrences of the variable x in φ by the term t. By $\mathit{range}(x, \varphi, \mathfrak{A})$ of a formula φ with distinguished variable x in a state $\mathfrak{A}$ we denote the set of all elements a of $\mathfrak{A}$ that make the formula true if x is interpreted by a.

choose-processes as defined in Table 1 are blocking in case there is nothing to choose from, namely when the choice set is empty, since for this case we provide no inference rule. This is in contrast to the ASM **choose**-construct which in case of an empty choice set is usually treated as equivalent to **skip**.

The submachine call defined in Table 1 is by reference. The call-by-value version can be defined using the **let**-construct.

3 ASPs for Interleaving and Selective Synchronization

We add here to standard ASPs the interleaving operator **intlea** and the related selective synchronization operator **sync** (t). Their semantics is defined in Table 2. Interleaving is a choice among processes to perform the next step where however the programs of the not-chosen processes remain in force for subsequent choices. To exhibit the analogy of interleaving to synchronous parallelism, which involves a form of universal quantification, we formulate the **intlea**-rule for an arbitrary set of processes determined by a property $\varphi(x)$. As a consequence, to determine the set of the processes P_b which have not been chosen but are put into interleaving with the residual process P_a of the one process chosen for execution, one needs to keep track of the instantiations of $P(x)$ by the elements b

choose x **with** φ **do** P $\mid \{P(x) \mid \varphi(x)\}$	**choose** $\{P(x) \mid \varphi(x)\}$
forall x **with** φ **do** P $\mid\mid \{P(x) \mid \varphi(x)\}$	**par** $\{P(x) \mid \varphi(x)\}$
$\mid t \mid \{P(x) \mid \varphi(x)\}$	**sync** $(t)\{P(x) \mid \varphi(x)\}$
$\mid\mid\mid \{P(x) \mid \varphi(x)\}$	**intlea** $\{P(x) \mid \varphi(x)\}$
operator$\{P_1, \ldots, P_n\}$ (**operator** $=\mid, \mid\mid, \mid t \mid, \mid\mid\mid$)	**operator**$\{P(x) \mid \varphi(x)\}$ **where** $\{P(x) \mid \varphi(x)\} = \{P_1, \ldots, P_n\}$

Table 3. Syntactic variations of some ASP constructs.

which satisfy the condition φ. Formally this comes up to bind pairwise different new incarnations y_b of the parameter x to b. In the case of an explicitly given set $P_1, \ldots, P_n$ of processes where the parameter instances are fixed, our notational convention eliminates the need to reinstantiate these fixed parameter indeces $1 \leq i \leq n$.

The synchronization operator **sync** (t) allows one to prevent the occurrence of actions which do not involve an update of (the location determined by) t by all the synchronized processes, similar to the *restriction* operation in SCCS. We write for the set of locations appearing in an update set $Loc(U) = \{loc \mid (loc, v) \in U \text{ for some } v\}$. The set of locations determined by a set T of terms in a state is denoted by $Loc(T)^{\mathfrak{A}} = \{(f, ([\![t_1]\!]^{\mathfrak{A}}, \ldots, [\![t_n]\!]^{\mathfrak{A}})) \mid f(t_1, \ldots, t_n) \in T\}$. We write $Loc(t)^{\mathfrak{A}}$ for $Loc(\{t\})^{\mathfrak{A}}$.

Whereas every ASM program yields in every state an update set (though it may be inconsistent in which case the computation is abrupted), when executing an ASP it may happen that the current program in the current state has no yield because no axiom or inference rule can be applied. This produces a form of ASP termination which does not exist for ASMs, that of a (static) deadlock. If it occurrs it might represent a bug in the modelled system. It can arise in the following cases for the residual program:

- A conditional expression with blocked argument process.
- A sequential expression with blocked first argument.
- A choice expression where all alternatives are blocked.
- A parallel expression with at least one blocked process.
- An interleaving expression where all component processes are blocked.

4 Two examples

We illustrate by two simple examples how the operators **intlea**, **sync** (t) allow one to make standard scheduling and handshaking disciplines transparent.

Local sequential scheduling of subprocesses. Using basic ASMs one can express a general modular scheme for scheduling the execution of rules belonging to a set $\mathfrak{R}$ which may even be dynamic. It suffices to update by a *scheduler* a function *select* which chooses the rule $R(select)$ to be executed next.

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

An instance of the scheme is to require the *scheduler* to make *select* behave as interleaving. Another example is the Round Robin principle which is defined by the equation $scheduler(select) = select + 1 \bmod n$ for any number n of processes. The use of such a *scheduler* function, which may also be dynamic, allows one to restrict the mere interleaving by conditions which may still leave some freedom to choose the next rule to be executed, maybe also in dependence of the state where *scheduler* is used.

However, to describe with basic ASMs the combination of interleaving with white-box sequential execution would lead to a programming solution which seems to be less natural than using the ASP operators **intlea** and **then**. Imagine for example the set $\mathfrak{R}$ consists of processes R of form

R = $First_R$ **then** $Second_R$

which are 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 a) locally record by a copy of the current *ticket* value 'when' **self** has been called to start the execution of its first subprocess, and b) to locally advance *ticket* to the next free position in the dynamic ordering of calls of subprocesses $First_R$. This means to place two updates to get and advance the current ticket in parallel with $First_{\mathbf{self}}$: **self** .*ticket* := *ticket* and *ticket* := *ticket* + 1. Then $Second_{\mathbf{self}}$ can be scheduled when its ticket is say 'displayed', i.e. when the guard **self** .*ticket* = *display* is true, adding to $Second_{\mathbf{self}}$ an update to advance the *display*. This yields the following transformation of **intlea** ($\mathfrak{R}$) into **intlea** ($\textsc{LocalSeqSched}(\mathfrak{R})$).

$\textsc{LocalSeqSchedule}(R) =$ $First'_R$ **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 can be viewed as a special case of shared memory communication, namely via gate locations g shared for reading and/or writing. The general scheme for two processes P, Q is as follows, where we use predicates φ, ψ to determine the choice the processes may have for agreeing upon a consistent update of gate g with a value determined by terms s, t:

$$\begin{aligned}&\textsc{Handshaking}(P, \varphi, s, Q, \psi, t) = \textbf{sync } (g)\{R, S\} \textbf{ where}\\ &\quad R = \textbf{choose } x \textbf{ with } \varphi(x) \textbf{ in } (g := s(x) \textbf{ then } P(x))\\ &\quad S = \textbf{choose } y \textbf{ with } \psi(y) \textbf{ in } (g := t(y) \textbf{ then } Q(y))\end{aligned}$$

This ASP formulation of handshaking realizes the 'agreement on values offered at a gate' by the consistency condition for the gate updates. If one wants to faithfully reflect also that in the process-algebraic view, gates are only virtually updated, serving only as communication medium, one can declare the updates of g to be transient, i.e. not relevant for the resulting state transformation. This can be done by simply not considering these transient updates in the definition of the next-state function, which associates new states to given states and update sets. A similar expedient has been used already in [9] to restrict update sets to updates of not-local functions and for defining the error handling construct. The technique can be exploited further for defining practical hiding disciplines for ASMs and ASPs.

5 Related and further work

The relation of our work to the rich process algebra and ASM literature can easily be traced through [11] and [6]. The related approaches to combine a state based method, namely Z, Object-Z or B, with behavioural concepts from process algebraic systems, namely CSP and CCS, are extensively discussed in Chapter 18 of [11], and the references to the relevant papers can be found there. The main difference of our approach with respect to previous work is that we allow, as states, arbitrary structures (as in Z) and as behaviour expressions arbitrary machines (not only single operations, executed one per time, as in Z or B), thus gaining from process algebras the richness of behavioural structure without loosing the generality of the state-based ASM method (see the ASM book [10]). As a result, the general abstraction and refinement techniques survive which made complex real-life applications of ASMs successful (see [7] for a survey).

An interesting start for the further analysis of ASPs is to characterize the possible runs of ASPs, similar to the elegant characterization of turbo ASM runs in [13]. Another direction of study is to investigate to what extent one can adapt to ASPs the structured and proof-oriented process-algebra refinement techniques in [11]. Maybe this leads to a fruitful integration of these techniques into the practical instantiations of the general refinement scheme which has been formulated for ASMs in [6]. A third field of interest is to investigate which process-algebraic proof rules can be generalized to support deductive and possibly mechanically

verified reasoning about ASP runs[4]. We would also like to see the impact of ASPs on the various existing implementations of basic or turbo ASMs, in particular for implementing asynchronous ASMs. Last but not least we are looking for interesting real-life practical applications of ASPs.

Acknowledgement. We thank unknown referees for critical remarks upon a draft of this paper.

References

1. J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.
2. A. Blass and Y. Gurevich. Abstract State Machines capture parallel algorithms. *ACM Transactions on Computational Logic*, 3, 2002.
3. T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.
4. E. Börger. Why use evolving algebras for hardware and software engineering? In M. Bartosek, J. Staudek, and J. Wiederman, editors, *Proceedings of SOFSEM'95, 22nd Seminar on Current Trends in Theory and Practice of Informatics*, volume 1012 of *LNCS*, pages 236–271. Springer-Verlag, 1995.
5. E. Börger. High level system design and analysis using abstract state machines. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Current Trends in Applied Formal Methods (FM-Trends 98)*, number 1641 in LNCS, pages 1–43. Springer-Verlag, 1999.
6. E. Börger. The ASM refinement method. *Formal Aspects of Computing*, 14:?–?, 2002.
7. E. Börger. The origins and the development of the ASM method for high level system design and analysis. *J. of Universal Computer Science*, 8(1):2–74, 2002.
8. E. Börger and T. Bolognesi. Remarks on turbo asms for computing functional equations and recursion schemes. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003*, LNCS. Springer-Verlag, 2003.
9. E. Börger and J. Schmid. Composition and submachine concepts for sequential ASMs. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic (Proceedings of CSL 2000)*, volume 1862 of *LNCS*, pages 41–60. Springer-Verlag, 2000.
10. E. Börger and R. Stärk. *Abstract State Machines. A method for high-level system design and analysis*. Springer, 2003.
11. J. Derrick and E. Boiten. *Refinement in Z and Object-Z*. Springer, 2001.
12. L. M. G. Feijs and H. B. M. Jonkers. *Formal Specification and Design*. Cambridge University Press, 1992.
13. N. G. Fruja and R. F. Stärk. The hidden computation steps of turbo Abstract State Machines. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003*, LNCS. Springer-Verlag, 2003.
14. Y. Gurevich. Evolving algebras. a tutorial introduction. *Bulletin of EATCS*, 43:264–284, 1991.

[4] In reaction to the announcement of T. Bolognesi's talk on this work at MSR in Redmond on November 15, 2003, Leslie Lamport sent us an unpublished manuscript [17] where he develops a notation for TLA actions and predicates which reflect process-algebraic concepts, aiming at exploiting process-algebraic rules for reasoning about programs in a programming language defined by operators for sequential composition, choice, interleaving and iteration with a break statement.

15. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
16. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.
17. L. Lamport. Adding "Process Algebra" to TLA. January 1995.
18. R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.
19. A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall, 1998.
20. S. Schneider. *Concurrent and Real-Time Systems: The CSP Approach.* Wiley, 2000.
21. R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation.* Springer-Verlag, 2001. .