Research Collection

Object-based abstract state machines

Author(s):
Janneck, Jörn Wilhelm; Kutter, Philipp W.

Publication Date:
1998

Permanent Link:
https://doi.org/10.3929/ethz-a-004289119

Rights / License:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection. For more information please consult the Terms of use.
Abstract. When designing and implementing complex computer systems, powerful description techniques are needed that allow the representation of hardware and software components at various abstraction levels and support refinement during the development process. These techniques should be formal in order to support mathematical reasoning about systems properties while at the same time they should be operational so as to facilitate comprehension and implementation of the system.

In this paper we propose Object-based Abstract State Machines (ObASM), an extension of Abstract State Machines (ASM). ASM are an operational formalism that has been successfully applied to the description of hardware and software systems. ObASM are adding a notion of locality of state and the concept of communication between loosely coupled agents. These extensions are designed to be minimal additions to the traditional ASM formulation, preserving their simplicity while at the same time significantly enhancing their modeling power.

We give a formal semantics for ObASMs and discuss the properties of the extended formalism.

1 Introduction

Today’s software and hardware systems are characterized by a composition of complex subsystems and complex interactions between them. An abstract specification of the whole system is needed to understand the behavior before expensive implementation is started. Research proposed many formalisms that are well suited for analysis or even verification. Unfortunately it turned out, that such descriptions are not always simple enough to be used in industrial system development, and even where the methods have been successfully applied [19], it was often not possible to refine the descriptions down to the implementation [18].

Gurevich’s Abstract State Machines (ASMs) [13][14] are an operational formalism that can be used to model algorithms on different abstraction levels, from an abstract view down to the implementation details. ASMs have proven their applicability and usefulness to system specification and design through numerous extensive case studies in major domains as diverse as semantics and
implementation of languages (C, C++, OCCAM, Oberon, Java, VHDL, SDL), hardware and software architectures (APE 100, ARM, pipelining in RISC architectures, PVM, Transputer), and embedded control systems (steam boiler control, production cell) [5][20].

Although ASMs are easily used to describe complex hard- and software systems, they lack support for combining the specified systems. One of the obstacles is, that the global state is always present, and side-effects of one subsystem may influence another subsystem in an uncontrolled way. Another problem is, that in each of the mentioned case studies, the composition of simple systems to more complex ones is done in an ad hoc way. Distributed ASMs [17][12] have addressed the problem of composing subsystems by introducing a number of independently running agents, that communicate via shared memory. There is a priori no locality of states and message passing can be simulated [16], but is not a primitive concept. Montages [24][2] provide a generic version of the composition mechanisms used in the above mentioned case studies in programming languages semantics. The method concentrates on a visual formalism for initializing control data flow links among the components. Locality is used but due to the underlying ASMs not guaranteed, and message passing is coded as control pointers that move along the links. In this paper, we propose a variant of ASMs accounting for locality and communication. The main idea is to make the object oriented flavor of ASMs more explicit, having both state and behavior attached to objects, that communicate by message passing only. The resulting Object based ASMs (ObASMs) guarantee compositionality in the sense, that algorithms based on the local state of an object cannot be disturbed by other system-components, since only local updates of the state are allowed. On the other hand, ObASMs are general allowing to model the full range from tightly coupled to loosely coupled, distributed systems. The appropriateness of the object based message passing paradigm to the former is evidenced by the growing body of distributed software component models (at least partly) based on this approach, like e.g. CORBA, DCOM, or JavaBeans, while its suitability for the latter is shown in, for instance [8].

In the next section we give an introduction to ASMs, and the definition and semantics of ObASMs. Our definition allows to see the system both operationally using ASMs and denotationally as fix-points of general transition-systems evaluated non-deterministically along a partial ordering following only the causality of messages. The latter, large grain view of our system allows the use of other formal methods. As example we give two class definitions modeling the distributed behavior of places and transitions in Petri-nets.

ObASMs are thought to be integrated with existing hardware and software systems and used not only for specification and analysis but the entire system engineering process including prototyping and testing. The requirements for this integration are given in section 5.
2 Abstract State Machines

The fundamental concept in ASMs is the object, which we consider an atomic entity. We call the set of all objects the universe $\mathcal{U}$ \footnote{In traditional ASM literature this set is called super-universe and sets are called universes.}. The state $\lambda$ of the system is given as the mapping of a number of function symbols, the signature $\Sigma$, to actual functions. For short we write $f_\lambda$ for the function interpreting the symbol $f$ in state $\lambda$. We write $\mathcal{S}(\Sigma)$ for the set of all $\Sigma$-states.

A state transition changes these functions pointwise, by so-called updates. An update is a triple

$$(f, (o_1, \ldots, o_n), o_0)$$

where $f$ is an n-ary function symbol in $\Sigma$, and $o_0, \ldots, o_n \in \mathcal{U}$. Intuitively, firing this update at a state $\lambda$ changes the function associated with $f$ in $\lambda$ at the point $(o_1, \ldots, o_n)$ to the value $o_0$, leaving the rest of the function unchanged. In other words, given an update set $\alpha$ and a state $\lambda$, firing $\alpha$ at $\lambda$ results in the successor state $\lambda' = \alpha(\lambda)$. We then have the following relation between $f_\lambda$ and the new functions $f_{\lambda'}$:

$$f_{\lambda'}(o_1, \ldots, o_n) \mapsto \begin{cases} o_0 & \text{if } (f, (o_1, \ldots, o_n), o_0) \in \alpha \\ f_{\lambda'}(o_1, \ldots, o_n) & \text{otherwise} \end{cases}$$

(1)

The update set for a given state is computed by the transition function. This function is the denotation of a rule, which is basically a syntactical object describing the updates to be performed in a given state. For instance, the rule $f(2, x) := y$ would result in the update $(f, (2, 6), 1)$ if $x$ and $y$ would be function symbols of arity 0 evaluating to the objects 6 and 1, respectively.

The syntactical constructions allowed in rules and their denotations in terms of transition functions is given in the appendix.

3 Object-based Abstract State Machines

An Object-based ASM (ObASM) describes the behavior of a system of objects that can be considered to have concurrent processing threads associated with them. Individual threads communicate via message passing. Computation is always initiated by an incoming message.

3.1 Terms and evaluation

Each object $x \in \mathcal{U}$ has an associated class $\gamma(x)$ from the set $\mathcal{C}$ of all classes in an ObASM. The class $c$ of an object determines its behavior, given by transition functions, and its signature $\Sigma_c$. 

\footnote{In traditional ASM literature this set is called super-universe and sets are called universes.}
The vocabulary of the entire system is given as

\[ \Sigma = \bigcup_{c \in \mathbb{C}} \Sigma_c \]  

(2)

Further an object \( x \) has associated with it its own local state \( \lambda(x) \in \mathcal{G}(\Sigma_c) \). The global state is given as a function \( \lambda \) mapping each object to its local state:

\[ \lambda : \mathbb{U} \rightarrow \bigcup_{c \in \mathbb{C}} \mathcal{G}(\Sigma_c) \quad \text{such that} \quad \lambda(x) \in \mathcal{G}(\Sigma_{\gamma(x)}) \]  

(3)

In ObASM computation is always understood as processing of a message. A message consists of three parts, all of which are objects in \( \mathbb{U} \): its sender and receiver, and its content. These three objects are accessible with the constants \( \text{snd}, \text{rcv}, \) and \( \text{cnt} \), respectively. Term evaluation is defined with respect to \( \text{rcv} \), such that normal function applications are evaluated in the local state of \( \text{rcv} \).

In order to make the local state of other objects readable, the result \( x \) of a function application \( f(t_1, \ldots, t_n) \) may be used to evaluate a term \( g(t'_1, \ldots, t'_m) \) with respect to \( x \) in a so-called qualified term:

\[ f(t_1, \ldots, t_n).g(t'_1, \ldots, t'_m) \]

This construction allows to access the local state associated with \( x \) in a read-only fashion.

We will denote the evaluation of a term \( t \) w.r.t. the local state of an object \( x \) in state \( \lambda \) by \( \text{eval}_{\lambda,x} \). Then the evaluation of this term w.r.t. the object \( x \) is defined as follows, writing \( f_{\lambda(x)} \) to denote the interpretation of function \( f \) in the local state of \( x \). If \( T = f(t_1, \ldots, t_n) \) then

\[ \text{eval}_{\lambda,x}(T) = f_{\lambda(x)}(\text{eval}_{\lambda,\text{rcv}}(t_1), \ldots, \text{eval}_{\lambda,\text{rcv}}(t_n)) \]

In other words, evaluation with respect to an object \( x \) applies \( x \)'s interpretation of \( f \) to the evaluation of the arguments w.r.t. \( \text{rcv} \). Now we can define the evaluation of a qualified term by evaluating the second term w.r.t. the result of the first:

If \( T = [t_1, t_2] \) then

\[ \text{eval}_{\lambda,x}(T) = (\text{let } y = \text{eval}_{\lambda,x}(t_1) \text{ in } \text{eval}_{\lambda,x}(g(t_2))) \]

While the rule computing the updates to be performed may depend on the global state\(^2\), it may only alter the assignment \( \lambda(\text{rcv}) \), i.e. the local state of the object. In other words, terms can be evaluated with respect to other objects, but

---

\(^2\) In fact, at any given moment, it can depend only on a part of \( \lambda \), viz. those objects and their state which are reachable from \( x \) in a fixed number of steps given statically by the nesting of the terms in its rules.
only functions in the signature of \( rcv \) may be updated. Thus updates can only have the form

\[
f(t_1, \ldots, t_n) := g_1(\ldots), \ldots, g_m(\ldots)
\]

In such a context, the equation (1) relates \( \lambda(rcv) \) with the local successor \( \lambda'(rcv) \) whereas the rest of the state remains unaltered.

As far as the rules are concerned, computing their denotation (i.e. the transition function the define) is identical to the ASM case, except that we replace \( eval \) with \( eval_{\lambda,rcv} \).

If an object has to effect a state change in another object, the only way to do this is by passing it a message. A rule may produce an arbitrary number of messages, by using the \( send \) rule. The update-denotation of the send construct is the empty update set (cf. the appendix for a formal definition).

### 3.2 State transition

Each class \( c \) has three transition functions associated with it:

1. The initialization function \( I_c \) describes the initialization behavior when an instance of the class is newly created,
2. The initial transition function \( R^0_c \) that defines the first step in reaction of incoming messages.
3. The iterated transition function \( R_c \) that defines the subsequent iteration behavior\(^3\)

In ObASMs, these transition functions, when applied to a state, do not only result in an update set, but they also produce a number of events that were sent. Correspondingly, rules do not only have an update-denotation (as in ASMs), but also a message-denotation. The formal definition of the message-denotation for the rules in our ObASM language is given in the appendix.

In the system state at any given time, any number of messages may be pending, i.e. they have been sent but not yet received and acted upon. However, these messages do not form an unstructured set, because messages sent by the same object in consecutive steps are guaranteed to be processed in the order they were sent, i.e. the pending messages form a partial order \( (M, \prec) \) (or \( M \) for short), where \( m_1 < m_2 \) iff \( m_1 \) has been sent by the same object strictly before \( m_2 \). Given such a system state \( S = (\lambda, M) \), we compute a successor state as follows:

If \( M \) is empty, there is no event to be processed, and the system terminates. Otherwise, \( M \) contains an \( m \) such that there is no earlier \( m' \), i.e. \( m' < m \). Of course, there may be several such elements, in which case one is chosen\(^3\) of course, there will be many primitive classes with no behavior attached to them (empty rules/transition functions), i.e. integers, booleans, etc. This may have far-reaching consequences, both in terms of the theoretical analysis of such systems as well as their implementations.
nondeterministically. Given \( m \), let us call the sender, the receiver and the content \( \text{snd}_m \), \( \text{recv}_m \), and \( \text{cnt}_m \), respectively. Now the computation of the successor state \( S' \) is computed iteratively, and the iteration is initialized by applying the initial transition function \( R^0_{\gamma}(\text{recv}_m) \) to the current global state, resulting in a first update set \( \alpha' \) and some new events \( M' \):

\[
R^0_{\gamma}(\text{recv}_m)(\lambda) = (\alpha', M')
\]

and then

\[
\lambda_0 : x \mapsto \begin{cases} 
\lambda(x) & x \neq \text{recv}_m \\
\alpha'(\lambda(x)) & x = \text{recv}_m
\end{cases}
\]

\[
M_0 = M \circ M'
\]

The update set is applied to the local state of \( \text{recv}_m \) (the rest of the global state remains unchanged), while the new events are appended to the existing events in \( M \). The result \( (M, <) \circ (M', <) \) is a new event structure \( (M \cup M', <) \) such that for all \( m, m' \in M \cup M' \) we have

\[
m \prec M \cup M' \iff (m < m') \vee (m' < m) \vee (m \in M \land m' \in M' \land \text{snd}_m = \text{snd}_{m'})
\]

i.e. events that were ordered in either \( M \) or \( M' \) remain ordered, and additionally events originating from the same object at different times are ordered accordingly.\(^4\)

Now assume that for each iteration step, \( R_{\gamma}(\text{recv}_m)(\lambda_i) = (\alpha_i, M'_i) \). Then we define

\[
\lambda_{i+1} : x \mapsto \begin{cases} 
\lambda(x) & x \neq \text{recv}_m \\
\alpha_i(\lambda_i(x)) & x = \text{recv}_m
\end{cases}
\]

\[
M_{i+1} = M_i \circ M'_i
\]

This process is repeated until a fixed point is reached, i.e. we have a \( k \) such that \( \lambda_{k+1} = \lambda_k \) and \( M_{k+1} = M_k \). Then the successor state \( S' \) of \( S \) can be written as follows, removing the event just processed from the event structure:

\[
S' = (\sigma, \lambda_k, M_k \setminus \{m\})
\]

Of course, if such a fixed point is not reached, the computation does not terminate.

\(^4\) Note that the resulting order is transitive, because by construction of the event structures, \( m < m' \Rightarrow \text{snd}_m = \text{snd}_{m'} \).
4 An Example: Petri Nets

In the following we will describe a formal distributed ObASM specification for loop-free Petri nets. Petri nets are a very general formal model, different flavors of which have been successfully applied to the modeling and analysis of a broad range of distributed and concurrent systems [3, 7, 22, 23, 25, 27]. They can be given an operational semantics, but writing interpreters for them can be cumbersome, in particular when these interpreters should execute them in a distributed fashion [11, 10, 21, 28, 29].

The distributed execution suggested in the following example parallels the transition firing protocol in [29]. Basically, each transition and each place have a process associated with them, and they communicate with each other about the availability of tokens and their movement.

![Diagram of a transition with pre- and postplaces](image)

The presented model consists of two classes, one for places (PLACE) and one for transitions (TRANSITION). In Fig. 4 a place is visualized as a circle and a transition as a bar. The places \( \text{pre}_1, \ldots, \text{pre}_n \) form the preset of the depicted transition, and the places \( \text{pre}_1, \ldots, \text{pre}_m \) form the postset. Pre- and postset are modeled by their characteristic functions, e.g., unary functions in the signature of TRANSITION, being set to true for each member. Members of the preset are called preplaces and members of the postset are called postplaces. Another unary function in the signature of TRANSITION denotes the weight of the arrow between the transition and its preplaces (\( w_1, \ldots, w_n \) in Fig. 4) and postplaces (\( w'_1, \ldots, w'_m \) in Fig. 4). This three functions and the instances of PLACE and TRANSITION form the static part of our ObASM model.

The state of a Petri-net is given by the number of tokens at each place, here modeled by a variable (0-ary function) marking in the signature of PLACE. Given a marking a transition can fire if all preplaces have at least as many tokens as the weight of their arrows. If the transition fires, it subtracts \( \text{weight}(\text{pre}) \) tokens from each preplace \( \text{pre} \), and it adds \( \text{weight}(\text{post}) \) tokens to each postplace \( \text{post} \).

Each transition tries to fire, whenever its pre- and postset have enough tokens. Two transitions might compete for the same tokens (held by a common preplace) and a kind of hand-shaking is thus needed. The unary dynamic function reserved
is used to keep track how many tokens are reserved by a transition. The following three kinds of messages can be sent by a transition \( t \) to a place \( p \):

**Request**\( (n) \) requests \( n \) tokens from \( p \). If there are more then \( n \) tokens, \( p \) answers with a **Provide** message and reserves the requested tokens, otherwise it answers with a **Reject** message.

**Release** releases the tokens that have been reserved by \( t \).

**Put**\( (n) \) sends \( n \) tokens to \( p \). In response to that message, \( p \) adds \( n \) to the number of tokens **marking**.

The complete rule for PLACE is:

```java
class PLACE is
    Put\( (n) \): \textit{marking} := \textit{marking} + n
    Request\( (n) \): if \textit{marking} \geq n then
        \textit{marking} := \textit{marking} - n
        \textit{reserved} \( \textit{snd} \) := n
        Provide \( \rightarrow \textit{snd} \)
    else
        Reject \( \rightarrow \textit{snd} \)
    endif
    Release: \textit{marking} := \textit{marking} + \textit{reserved} \( \textit{snd} \)
```

To keep track of the hand-shaking a transition uses a unary relation (function with co-domain Boolean) **asked** to remember those preplaces from which it requested tokens, and it uses a unary relation **providing** to remember those preplaces that provide the requested tokens. The transition is ready to fire again, if none of its preplaces is asked:

\[
\textit{ready} \triangleq \forall p: \textit{preset}(p) : \neg \textit{asked}(i)
\]

We assume to start in a state where all functions are initialized and where the event structure contains one **Fire** message sent to each transition. If a transition receives a **Fire** message, it checks whether it is **ready** and whether the preplaces have currently enough tokens. If not, it resends the **Fire** message to itself. Otherwise, it sends to all preplaces a **Request** message, and it sets **asked** to true for all preplaces. The variable **state** is set to **trying**.

```java
class TRANSITION is
    Fire: if (\textit{ready} \land \forall p : \textit{preset}(p) : \textit{marking}(p) \geq \textit{weight}(p)) then
        do forall p : \textit{preset}(p)
            asked\( (p) \) := true
            Request\( (\textit{weight}(p)) \rightarrow p \)
        enddo
        trying := true
    else
        Fire \rightarrow \textit{rcv}
    endif
```
Once the system is in state trying, it waits for a Provide or Reject message from each preplace. The function providing is set to true for each preplace that sent a Provide message. If providing is true for all preplaces, the firing was successful, a Put message is sent to all postplaces, providing is reset, and the transition sends a Fire message to itself. In the case a Reject message arrives, the state is set to failed, and a Release message is sent to all providing preplaces. Please note, that at this stage the transition might not yet be ready.

In the failed-state, incoming Rejects and Provides are answered by setting asked to false, incoming Provides are additionally answered by sending a Release message. If none of the preplaces is asked anymore, the transition is ready to fire again. The rules are as follows:

Provide:

\[
\text{if } state = \text{trying then}
\]

\[
\text{if } \forall p : \text{preset}(p) : p = \text{snd} \lor \text{providing}(p) \text{ then}
\]

\[
\text{do forall } p : \text{postset}(p)
\]

\[
\text{Put}(\text{weight}(p)) \rightarrow p
\]

\[
\text{enddo}
\]

\[
\text{do forall } p : \text{preset}(p)
\]

\[
\text{providing}(p) := \text{false}
\]

\[
\text{asked}(p) := \text{false}
\]

\[
\text{enddo}
\]

\[
\text{Fire } \rightarrow \text{rcv}
\]

\[
\text{else}
\]

\[
\text{providing(snd ) } := \text{true}
\]

\[
\text{endif}
\]

\[
\text{elseif } state = \text{failed then}
\]

\[
\text{asked(snd ) } := \text{false}
\]

\[
\text{Release } \rightarrow \text{snd}
\]

\[
\text{endif}
\]

Reject:

\[
\text{if } state = \text{trying then}
\]

\[
\text{state := failed}
\]

\[
\text{do forall } p : \text{providing}(p)
\]

\[
\text{providing}(p) := \text{false}
\]

\[
\text{asked}(p) := \text{false}
\]

\[
\text{Release } \rightarrow p
\]

\[
\text{Fire } \rightarrow \text{rcv}
\]

\[
\text{enddo}
\]

\[
\text{else}
\]

\[
\text{asked(snd ) } := \text{false}
\]

\[
\text{endif}
\]

5 Integration in System Engineering

As mentioned in the introduction ASMs have been used to model both hardware and software systems. The case studies show, that ASMs are an implementation
independent formalism, that can account for the peculiarities of both hard and software. In [9] it is shown how to analyze such mixed systems and we believe that ASMs are well suited for hardware/software codesign. ObASMs support the design process by allowing to compose the heterogeneous components horizontally. Due to the locality of the components, verified vertical refinements from abstract ASMs to more concrete ones can still be applied.

Having a formalism suited to specify complex systems on different abstraction levels, the integration in system engineering poses the following additional requirements:

1. It must be very easy to generate prototypes, both for abstract and more concrete models.
2. The ObASM components must run together with existing, non specified hardware and software components.

Prototyping a single object can be done by compiling to executable code [1], or refining to an ASM that corresponds to a simple hardware model, as for instance the RT-formalization in [26], corresponding essentially to a simple form of ASMs. For the communication with other components only the message passing mechanism and the read access must be implemented. The implementation must support all different kinds of subsystem-prototypes and the surrounding existing system components.

One scenario that we implemented in a prototype is the integration of ObASMs in a component software system, in our case JavaBeans, where subsystems can be written in a traditional programming language (Java), rather than specified with ASMs. In our prototype each ObASM class is translated into a Java class, making components specified as ObASMs and directly implemented software components completely interchangeable.

Based on this integration, sophisticated test scenarios are possible, that can be applied whenever verification or model checking fails. One possibility is to have for critical software components both the ObASM specification and an implementation. The reliable, but slow formal model, could be run in parallel with the implementation until one has gained confidence in the implementation.

6 Future Work

In Aslan [1] the long present concept of hierarchical ASMs is implemented and used to structure single ASMs. Hierarchical ASMs allow to call other ASMs, which corresponds to synchronous message passing, in contrast to asynchronous message passing in ObASMs. Although synchronous message passing can be simulated with asynchronous, and vice versa, an integration of both may be useful. Another point is the introduction of object orientedness in ASMs. While Aslan proposes a simple, pure syntactical method mapping everything to the

\footnote{See [4] for verified refinements from Occam code to Transputer code, and [6] for verified hardware refinements of RISC architectures.}
global state, the use of ObASMs may lead to a more natural model, accounting for distributed loading and linking of classes as well as for garbage collection.

We mentioned Montages [24][2], a framework allowing to give semantics to a program by generating its control-/data-flow graph and giving the dynamic semantics of the single nodes by ASM-rules. As in ObASM, the nodes have a type, e.g. the syntactical category, a local state and a behavior. To use ObASM to formalize control flow in Montages might be convenient and fruitful, especially in the case of languages with distributed control flow. On the other hand, Montages might be useful to define control-/data-flow structures for ObASMs in a generic way.

Acknowledgments We would like to thank Matthias Anlauf, Yuri Gurevich, and last but not least Alfonso Pierantoni for the long and insightful discussions about ASMs, Montages, and Aslan.

Appendix: Denotational Semantics

Semantics of ASM rules

The formal semantics of a rule $R$ in a state $\lambda$ is given by a deterministic denotation $\text{Upd}(R, \lambda)$ being a set of updates [15]. The interpretation of $f$ in the successor state $\lambda'$ is defined as follows:

Update sets are defined by transition rules. The basic update denotes one update of a function at some point. The new values and the point are given by terms over the signature. Rules can be composed in a parallel fashion, such that all updates are executed at once. Conditional execution of a rule fires only in certain cases. The do-forall rule allows to fire the same rule for all objects satisfying some condition. Finally the extend-rule allows to introduce reserve-objects, that have not been used before. The last rule is typically used to create new objects of some class.

Formally, a transition rule $R$ is built up recursively by the following constructions. The corresponding denotations are given. Let $\text{eval}_\lambda$ be the usual term evaluation over the state $\lambda$.

$(\text{Upd 1: basic update})$ if $R =$

\[
 f(t_1, \ldots, t_n) := t_0
\]

where $t_0, \ldots, t_n$ are terms over $\Sigma$, then

\[
 \text{Upd}(R, \lambda) = \{(f, \text{eval}_\lambda(t_1), \ldots, \text{eval}_\lambda(t_n)), \text{eval}_\lambda(t_0))\}
\]

$(\text{Upd 2: parallel composition})$ if $R =$

\[
 R_1 \ldots R_m
\]

then

\[
 \text{Upd}(R, \lambda) = \bigcup_{i \in \{1, \ldots, m\}} \text{Upd}(R_i, \lambda)
\]

$(\text{Upd 3: conditional rules})$ if $R =$

\[
 \text{if} t \text{ then } R_{\text{true}} \text{ else } R_{\text{false}} \text{ endif}
\]

then

\[
 \text{Upd}(R, \lambda) = \begin{cases} 
 \text{Upd}(R_{\text{true}}, \lambda) & \text{if } \text{eval}_\lambda(t) = \text{true} \\
 \text{Upd}(R_{\text{false}}, \lambda) & \text{otherwise}
\end{cases}
\]
(Upd 4: do forall) if $R =$ \[\text{for all } e : t \quad R^e \text{ enddo}\] then \[Upd(R, \lambda) = \bigcup_{e \in \lambda} Upd(R, \lambda \cup e \mapsto o)\]

(Upd 5: extend) if $R =$ \[\text{extend } E \text{ with } e \quad R^e \text{ endextend}\] then \[Upd(R, \lambda) = Upd(R, \lambda \cup e \mapsto o),\] where $o$ is a completely new allocated element. \(^6\)

Additional Semantics for ObASM

ObASMs define another atomic rule that sends a message to an object. This rule results in no updates:

(Upd 0: send) if $R =$ \[t_1 \to t_2\] then \[Upd(R, \lambda) = \{\}\]

The other definitions of Upd remain the same, except that for each $x \ e v a l_x$ must be replaced with $e v a l_{x, r c v}$ and that in Upd 4 and 5 the extension $\lambda \cup e \mapsto o$ on the right-hand-side must be replaced by

$$\lambda^{e \mapsto o} = \lambda(\lambda(e) / (\lambda(\lambda(\lambda(e) \cup e \mapsto o))))$$

The message denotation $Mes(R, \lambda)$ defines a set of message triples (sender, receiver, content) as follows:

(Mes 0: send) if $R =$ \[t_{\text{new cnt}} \to t_{\text{new rcv}}\] then \[Mes(R, \lambda) = \{(\lambda(e), r c v(\lambda(e)), r c v(\lambda(e)))\}\]

(Mes 1: basic update) if $R =$ \[f(t_1, \ldots, t_n) := t_0\] then \[Mes(R, \lambda) = \{\}\]

(Mes 2: parallel composition) if $R =$ \[R_1 \ldots R_m\] then \[Mes(R, \lambda) = \bigcup_{i \in \{1, \ldots, m\}} Mes(R_i, \lambda)\]

(Mes 3: conditional rules) if $R =$ \[\text{if } t \text{ then } R_{\text{true}} \text{ else } R_{\text{false}} \text{ endif}\] then \[Mes(R, \lambda) = \begin{cases} Mes(R_{\text{true}}, \lambda) & \text{if } e v a l_{\lambda, r c v}(t) = \text{true} \\ Mes(R_{\text{false}}, \lambda) & \text{otherwise} \end{cases}\]

\(^6\) See [15] for a formalization of “completely new”
(Mes 4: do forall) if $R \neq 0$
do forall $e : t \ R'$ enddo

then

$$Mes(R, \lambda) = \bigcup_{\text{eval}_{\lambda}, e \in \text{eval}(t)} Mes(R, \lambda^{e+o})$$

(Mes 5: extend) if $R = \text{extend}$

then

$$Mes(R, \lambda) = Mes(R, \lambda^{e+o})$$

where $o$ is a completely new allocated element, whose type is set to the class $E$.

Using $Upd$ and $Mes$, the denotation $Den(R)$ of any rule $R$ can be seen as a mapping from global states to pairs of update sets and message structures:

$$Den(R) : \lambda \mapsto (Upd(R)(\lambda), Mes(R)(\lambda))$$

References