An executable semantics of the SystemC UML profile[⋆]

Elvinia Riccobene[1] and Patrizia Scandurra[2]

[1] DTI Dept., Università degli Studi di Milano, Italy
elvinia.riccobene@unimi.it
[2] DIIMM Dept., Università degli Studi di Bergamo, Italy
patrizia.scandurra@unibg.it

Abstract. The SystemC UML profile is a modeling language designed to lift features and abstractions of the SystemC/C++ class library to the UML level with the aim of improving the current industrial System-on-Chip design methodology. Its graphical syntax and static semantics were defined following the “profile” extension mechanism of the UML metamodel, while its behavioral semantics was given in natural language. This paper provides a precise and executable semantics of the *SystemC Process State Machines* that are an extension of the UML state machines and are part of the SystemC UML profile to model the reactive behavior of the SystemC processes. To this purpose, we used the meta-hooking approach of the ASM-based semantic framework, which allows the definition of the dynamic semantics of metamodel-based languages and of UML profiles.

1 Introduction

The SystemC UML profile [28, 24] is a modeling language developed to improve the conventional industrial Systems-on-Chip (SoC) design methodology with a model-driven approach [25–27]. It is a consistent set of modeling constructs designed to lift both structural and behavioral features (including events and time features) of SystemC [32] to the UML [33] level. It was defined by exploiting the UML profile mechanism that requires the specification of UML extension elements (stereotypes and tagged values) and of new constraints as Object Constraint Language (OCL) [22] rules.

The profile, while provides a complete description of the modeling syntax and static semantics, suffers from the lack of a precise behavioral semantics that is given in natural language. Indeed, in the OMG framework used to define the profile, as well as in other metamodeling environments (like Eclipse/Ecore, GME/MetaGME, AMMA/KM3, XMF-Mosaic/Xcore, etc.), the way to define the language *abstract syntax* in terms of a metamodel and its *static* semantics as OCL rules is well established, while no standard and rigorous support is given to provide the *dynamic* semantics that is usually expressed in natural language. This lack has negative consequences, as often remarked in the past since the first UML version. Moreover, defining a precise semantics of UML extensions is widely felt, especially now that UML is turning into a “family of languages” (see the OMG standardization activities of UML profiles in [33]).

[⋆] This work is supported in part by the PRIN project D-ASAP (Dependable Adaptable Software Architecture for Pervasive computing).

The definition of a means for specifying rigorously the semantics of UML profiles, as well as of metamodel-based languages, is therefore an open and crucial issue in the model-driven context.

In [11], a formal semantic framework based on the ASM (Abstract State Machine) formal method [2] is presented, which allows us to express a precise and executable semantics of metamodel-based languages using different techniques. We here adapt one of the techniques in [11], the *meta-hooking*, for UML profiles, and we show its application to the SystemC UML profile. This implies to provide a rigorous semantics of the *SystemC Process (SCP) state machines* formalism of the SystemC UML profile used to model the reactive behavior of SystemC processes.

This paper is organized as follows. Some background on the SystemC UML profile is given in Sect. 2. Sect. 3 presents the meta-hooking technique of the ASM-based semantic framework. Sect. 4 shows the application of the meta-hooking technique to the OMG metamodeling framework for the semantics specification of the SCP state machines. Some related work is presented in Sect. 5, while Sect. 6 concludes the paper.

2 The SystemC UML profile

SystemC [32] is an open standard in the EDA (Electronic Design Automation) industry. Built as C++ library, SystemC is a language providing abstractions for the description and simulation of SoCs. Typically, the design of a system is specified as a hierarchical structure of modules and channels. A *module* is a container class able to encapsulate *structure* and *functionality* of hardware/software blocks, while a *channel* (primitive or hierarchical) serves as a container to encapsulate the *communication* functionality of blocks. Each module may contain *attributes* as simple data members, *ports* (proxy objects) for communication with the surrounding environment and *processes* for executing module's functionality and expressing concurrency in the system. Fig.1 shows a module example, `count_stim`, containing a thread process `stimgen`, two input ports `dout` and `clock`, and two output ports `load` and `din`, in the SystemC UML profile.

We here skip the details concerning the structural modeling constructs, as the focus is on the behavioral aspects of the profile. Some basic concepts underlying the SCP state machines are reported below as defined in the SystemC UML profile [28]. This formalism is to be considered a conservative extension of the UML *method* state machine[3] defined through the UML extension mechanism of "profiles" (i.e., stereotypes, tags, and constraints)[33].

SystemC Process State Machines Processes are the basic unit of execution within SystemC and provide the mechanism for simulating concurrent behavior. Three kinds of processes are available: `sc_method`, `sc_thread` and `sc_cthread`. Each kind of process has a slight different behavior, but in general (i) a process is declared within the scope of a class (a module or a hierarchical channel) as a stereotyped operation with no return type and no arguments (see, for example, Fig.1); (ii) all processes run

[3] A UML "method" state machine specifies the algorithm or procedure for a behavioral feature (such as a class's operation).

concurrently; (iii) the process code is not *hierarchical*, i.e. no process can directly invoke another process (processes can cause other processes to execute only by notifying events); and (iv) a process is activated depending on its *static sensitivity* that is an initial list (possibly empty) of events that can dynamically change at run-time realizing the so called *dynamic sensitivity* mechanism. Finally, (v) all processes are usually activated at the beginning of the simulation, but a process can be explicitly not *initialized* – by means of a `dont_initialize` statement –, so it does not execute immediately when the simulation starts, but after a first occurrence of any of the events in its static sensitivity.

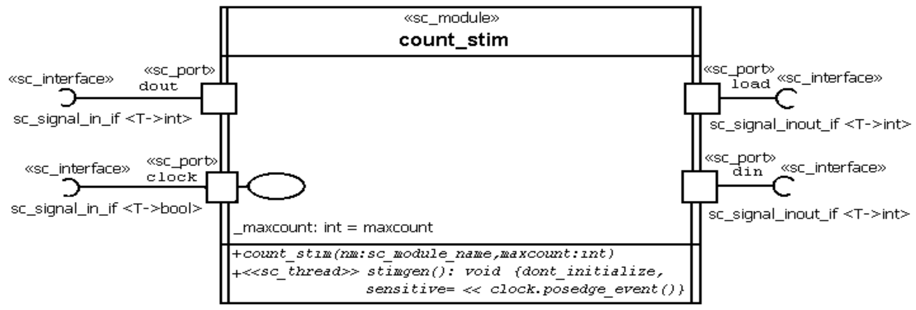
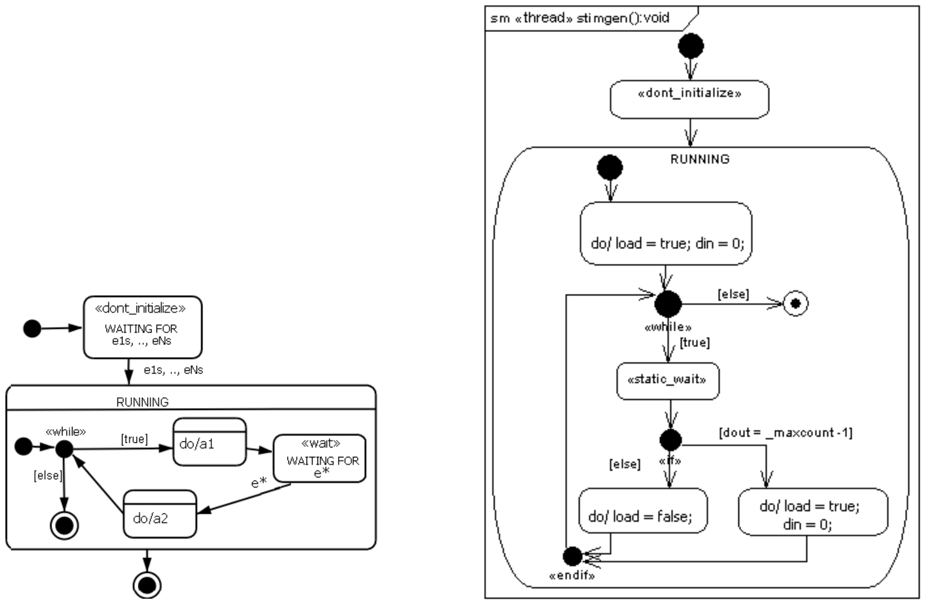

«sc_module»
count_stim
«sc_interface»
«sc_port»
dout
sc_signal_in_if <T->int>
«sc_interface»
«sc_port»
clock
sc_signal_in_if <T->bool>
«sc_port»
load
«sc_interface»
sc_signal_inout_if <T->int>
«sc_port»
din
«sc_interface»
sc_signal_inout_if <T->int>
_maxcount: int = maxcount
+count_stim(nm:sc_module_name,maxcount:int)
+<<sc_thread>> stimgen(): void {dont_initialize,
sensitive= << clock.posedge_event()}

Fig. 1. The `count_stim` module

In this paper, we focus on the `sc_thread` process type. Fig.2 (A) shows the UML schema of a SC thread state machine. This diagram corresponds to a SystemC thread that: (i) has both a static (the list $e_{1s}, \ldots, e_{Ns}$) and a dynamic sensitivity (the state `WAITING FOR e*` with the stereotype `wait`), (ii) runs continuously (by an infinite `while` loop), and (iii) is not initialized (the state with the `dont_initialize` stereotype follows the top initial state). Activities `a1` and `a2` stand for structured blocks of sequential (or not) code without `wait` statements. The wait-state denotes a generic `wait(e*)` statement where the event `e*` matches one of the cases described in Fig.3. The pattern in Fig.2 (A) can be more complex in case of `wait` statements within the scope of nested control structures. In this case, as part of the SystemC UML profile, the control structures `while`, `if-else`, etc., need to be explicitly represented in terms of special stereotyped junction or choice pseudostates.

Fig.2 (B) shows the state machine for the `stimgen` thread of the module shown in Fig.1. It is an instantiation of the pattern in Fig.2 (A). After initializing the `load` and `din` ports, the `stimgen` thread runs continuously: at each clock cycle (by the wait for the positive clock edge event of the static sensitivity list) it checks the received value from the `dout` port and may restart the counter in case it reaches the `_maxcount` attribute's value. Actions are specified using SystemC/C++ as action language.

The stereotype `sc_thread` labels both the operation indicating the thread process in the class of the container module (see, e.g., Fig.1) and the state machine defined for the process (see Fig. 2 (B)). The tag `sensitive` (see Fig.1) is used to declare the static sensitivity list of the thread (if any) using the form $\ll e_{1s} \ll .. \ll e_{Ns}$, where



(A) (B)

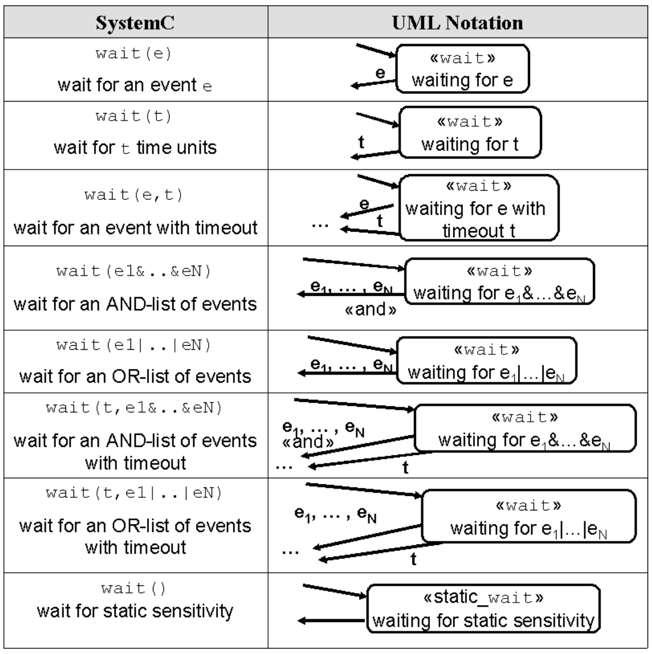
Fig. 2. A thread state machine pattern (A) and a (concrete) thread state machine (B)

$e_{1s}, .., e_{Ns}$ are event types. The boolean tagged value `dont_initialize`, whose default value is `false`, represents the SystemC `dont_initialize` statement. The `dont_initialize` stereotype is also applied to a simple state (see Fig.2) and is used to capture at state machine level the behavioral semantics of the `dont_initialize` statement. A `dont_initialize` state has only one outgoing transition with possibly no explicit triggers; it is assumed that the static sensitivity list of the process are the implicit trigger event list of this transition.

The dynamic sensitivity of a thread is captured at behavioral level in the state machine associated to the thread by the use of the stereotypes `static_wait` and `wait`. These stereotypes are applied to simple states. They model the SystemC `wait()` and `wait(e*)` statements for resuming a waiting process depending on its static and dynamic sensitivity, respectively. A `static_wait` state has only one outgoing transition, the *static resuming transition*, with no explicit triggers since it is assumed that the events of the static sensitivity list of the process are the implicit triggers of this transition. The parameter `e*` of a `wait(e*)` statement is the trigger of the outgoing transitions, the *dynamic resuming transitions*, of a `wait` state (see Fig.3).

3 ASM-based semantic framework

The semantic framework presented in [11] is based on the ASM formal method and allows to link the abstract syntax (metamodel) of a language with its executable be-



SystemC	UML Notation
`wait(e)` wait for an event `e`	«wait» waiting for e; e
`wait(t)` wait for `t` time units	«wait» waiting for t; t
`wait(e,t)` wait for an event with timeout	«wait» waiting for e with timeout t; e; t; …
`wait(e1&..&eN)` wait for an AND-list of events	«wait» waiting for e_1&…&e_N; $e_1, \dots, e_N$ «and»
`wait(e1|..|eN)` wait for an OR-list of events	«wait» waiting for e_1|…|e_N; $e_1, \dots, e_N$
`wait(t,e1&..&eN)` wait for an AND-list of events with timeout	«wait» waiting for e_1&…&e_N; $e_1, \dots, e_N$ «and»; t; …
`wait(t,e1|..|eN)` wait for an OR-list of events with timeout	«wait» waiting for e_1|…|e_N; $e_1, \dots, e_N$; t; …
`wait()` wait for static sensitivity	«static_wait» waiting for static sensitivity

Fig. 3. Dynamic Sensitivity of a Thread

havioral semantics expressed in terms of ASM transition rules. In the sequel, we recall from [11] some basic concepts.

A language metamodel *A* has a well-defined semantics if a semantic domain S is identified and a semantic mapping $M_S : A \rightarrow S$ is provided [13] to give meaning to syntactic concepts of A in terms of the semantic domain elements. In the ASM-based semantic framework, the mapping M_S is defined in terms of the ASM metamodel, *AsmM*, and its semantic domain S_{AsmM}[4]. The semantics of a "terminal model"[5] [15] conforming to A is therefore expressed in terms of an ASM model.

By assuming the semantic domain S_{AsmM} as the semantic domain *S*, the semantic mapping $M_S : A \rightarrow S_{AsmM}$ is defined as

$$M_S = M_{S_{AsmM}} \circ M$$

where $M_{S_{AsmM}} : AsmM \rightarrow S_{AsmM}$ is the semantic mapping of the ASM metamodel and associates a theory conforming to the S_{AsmM} logic with a model conforming to *AsmM*, and the function $M : A \rightarrow AsmM$ associates an ASM to a terminal model m conforming to A. Therefore, the problem of giving the metamodel semantics is re-

[4] S_{AsmM} is the first-order logic extended with the logic for function updates and for transition rule constructors formally defined in [2].

[5] A terminal model is a representation, that conforms to a reference metamodel, of a real world system (or portions of it).

duced to define the function M between metamodels. The complexity of this approach depends on the complexity of building the function M.

Different ways of defining M were presented in [11], classified in *translational* and *weaving*, depending on the abstraction level of the metamodelling stack [15]. Going up through the metamodeling levels, these techniques allow increasing automation in defining model transformations, increasing reuse and decreasing dependency of the final ASM with respect to the terminal model. Among them, we here commit with the translational *meta-hooking* approach that works at meta-metamodel level, and allows us to exploit the definition of the function $\gamma : MOF \longrightarrow AsmM$ (see below for details) defined in [11] and suitable for all languages whose metamodel is defined in terms of MOF. Here γ is adapted to handle also UML profiles, as the SystemC UML Profile.

Meta-hooking for MOF-based metamodels This technique aims at automatically deriving (most of the part of) the signature of the resulting ASM from the source metamodel A and MOF. This resulting algebra is then endowed with ASM transition rules to capture the behavioral aspects of the underlying language. Finally, by navigating a specific terminal model *m*, the initial state is determined.

Formally, the function $M : A \longrightarrow AsmM$ for a MOF-based metamodel A (such as UML or a UML profile) is defined as

$$M(m) = \iota(\omega(m))(\tau_A(\gamma(\omega(m))), m)$$

for all m conforming to A, where:
– $\gamma : MOF \longrightarrow AsmM$ provides signature (domain and function definitions) of the final machine $M(m)$ from the metamodel $\omega(m)$ to which m conforms to,
– τ_A: $AsmM \longrightarrow AsmM$ provides the ASM transition rules capturing the behavioral aspects of A,
– $\iota : MOF \longrightarrow (AsmM \times A \longrightarrow AsmM)$ is an HOT (High Order Transformation)[6] and establishes, for a metamodel A, the transformation $\iota(A)$ that computes the initial state of the final machine $M(m)$ by extracting initial values for data structures of the machine from the source modeling elements in *m*.

Mappings γ and ι are *universal*, i.e. once defined for the MOF, they are applicable to all metamodels conforming to MOF, and therefore to the SystemC UML Profile.

4 Meta-Hooking for the SystemC Process state machines

We here exploit the meta-hooking technique of the ASM-based semantic framework to provide the operational semantics of the SCP state machines. To this purpose, as domain A of the function M, we do not need to consider the whole SystemC UML metamodel, but only its portion related to the abstract syntax for modeling state machines. Figures 4 and 5 shows a simplified[7] portion of the UML metamodel (related

[6] An HOT is a transformation taking as input or producing as output another transformation.
[7] The effect of some OCL constraints of the SystemC UML profile is graphically emphasized by circles. They show that multiplicities have been restricted from many to exactly 1.

to the state machines) together with the stereotypes definitions (only some elements) capturing specific features of the SCP state machines.

The semantics specification of the SCP state machines is captured by an ASM model obtained in three steps: (1) the γ mapping (see the MOFtoAsmM transformation rules provided in [11], Table 1) is applied to the portion of the UML metamodel related to the state machine formalism and to its extension through stereotypes to obtain the ASM signature; (2) the operational semantics of the SCP state machines is then defined by ASM transition rules as form of pseudo-code operating on the abstract data derived from step 1; finally, (3) the initial state of the ASM model for a terminal model (later referred SC-UML) conforming to the SystemC UML profile is provided by an HOT ι similar to the one defined in [11], Table 2.

Note that to fulfill step 1, the γ mapping provided in [11] is further extended here to handle the stereotypes of the UML profile mechanism. To this purpose, a *stereotype* is treated similarly to a class, and therefore mapped into a domain that is subset of the domain corresponding to the (extended) base class. *Tag definitions* of stereotypes are attributes of the stereotype class, and therefore they are mapped to ASM functions having as domain the ASM domain corresponding to the stereotype, and as codomain the ASM domain corresponding to the attribute type. Generalization relationships between stereotypes are mapped as for generalization relationships between classes.

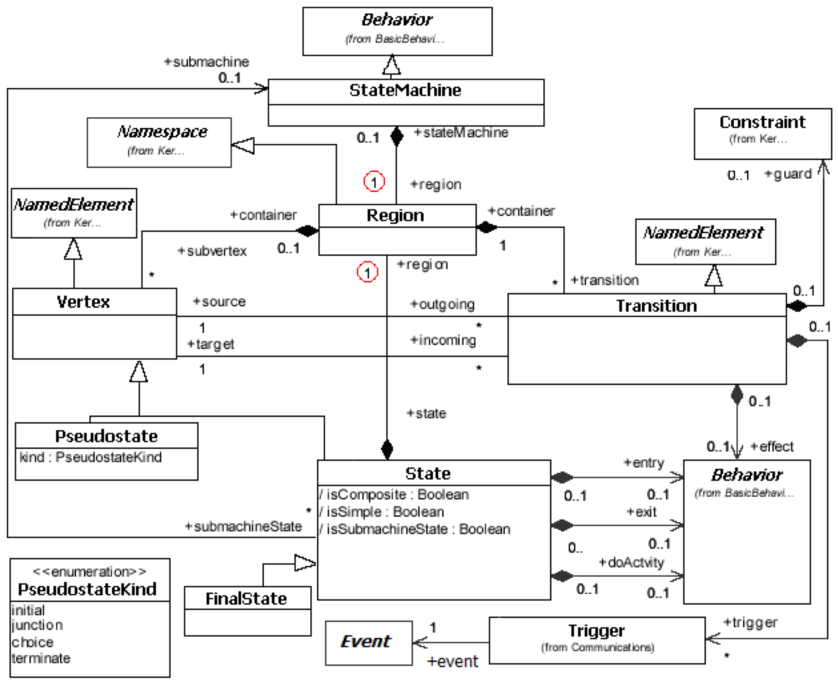


Fig. 4. SCP state machines metamodel (Part 1)

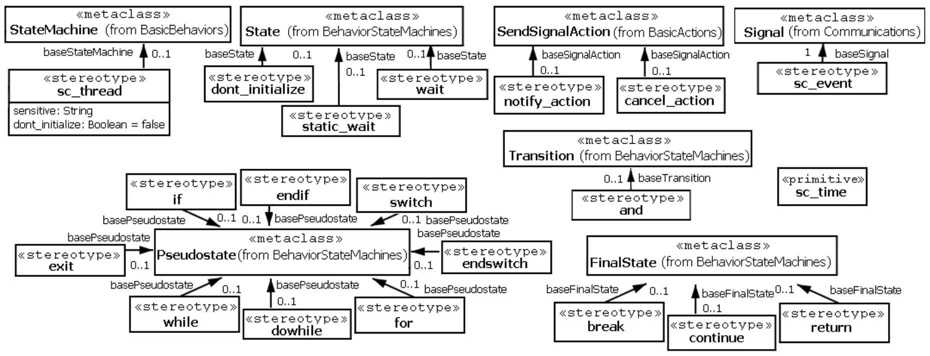


Fig. 5. SCP state machines metamodel (Part 2): some stereotypes

The ASM model described here is an adaptation of the ASM model presented in [1] where an ASM semantics of the UML 1.x state machines is described. Although there are common parts, the model provided here takes into account the UML2 version and the restrictions and the specific behavioral features of the SystemC UML profile. Due to the lack of space, only a subset of the entire set of ASM transition rules is reported. The reader can find more details in the preliminary work [10] and in the implementation available at [16] using the ASMETA/AsmetaL language[8]. Moreover, the reader is assumed to be familiar with the semantics of the UML state machines.

4.1 ASM signature

From the class diagram in Fig. 4 and the stereotypes in Fig. 5, a SCP state machine is a *sequential* state machine made up of just one `Region`, which in turn consists of (control) states and transitions belonging to the classes `Vertex` and `Transition`.

By applying γ to the SCP state machines metamodel, classes are mapped into ASM domains, generalization relationships are mapped into subset domain relations, and class attributes and associations are mapped into ASM functions suitable defined on the domains corresponding to the related classes. For example, the `State` class (see Fig. 4) is mapped in a subdomain of `Vertex`. Predicates *isSimple*, *isComposite*, and *isSubmachine* are defined on the domain `State` to distinguish among UML simple states, (sequential) composite states, and submachine states. In particular, simple states are of the form $state(name, container, incoming, outgoing, entry, exit, doActivity)$ where the parameter *name* is the name of the state, *container* denotes the region containing the state, $incoming/outgoing$ specify the transitions entering/departing from the state, $entry/exit$ denote actions that are performed as soon as the state is entered/exited, *doActivity* denotes the internal behavior (if any) that must be executed as long as the state is active. All these parameters induced from the associations of the `State` class (or from the super classes `Vertex` and *`NamedElement`*) are encoded in terms of ASM functions according to the γ mapping rules in [11].

[8] The ASMETA toolset http://asmeta.sf.net/

Stereotypes are mapped into domains, and their corresponding tags are mapped into functions, as well. OCL constraints of the SystemC UML profile, not reported here, state some restrictions on the stereotypes. This implies some constraints on the resulting ASM model. For example, the `wait` stereotype is mapped into a simple state that has no *entry*, *exit*, and *doActivity* behavior[9]. The outgoing transitions of a *wait* state are dynamic resuming transitions of form *trans*$($*container, source, target, trigger*$)$ where *container* denotes the region that contains the transition, *source*$/$*target* provide the source/target vertices of the transition, *trigger* is the label denoting the events (a time event or a signal event or an OR-list of signal events) which may enable the transition to fire. In case of an AND-transition – i.e. the dynamic resuming transition is stereotyped with `and` and labeled with a list of signal events –, the transition has AND-semantics: it may fire when all the events in the list have been notified, not necessarily all in the same delta-cycle[10] or at the same time. To manage the history of the event occurrences of an AND-transition (see *Transition Selection rule* in Sect. 4.2), the controlled function *andHist*$($t,trans$)$ is therefore introduced in the ASM signature and returns the list of events of an AND-transition t which have already been notified. Moreover, we distinguish wait-states using the predicate *isWait* on *State*, and AND-transitions using the predicate *isAnd* on *Transition*.

Control flow Further signature elements not directly induced from the metamodel are added to represent the nesting structure of a state machine and its control flow. Suitable functions to encode and *navigate* nested states, like the functions *Up/DownChain*(s_1, s_2) denoting ascending/descending sequences of nested states between states s_1 and s_2, are defined similarly to the ones used for the same scope in the original ASM model in [1]. These functions can be formulated by composition of ASM functions derived from the metamodel.

In the sequel, elements of the domain `Sc_thread` are referred to as *threads*. A thread t moves through a SCP state machine diagram, *baseStateMachine*(t), executing what is required for its *currently active state*. As effect of calling an operation op (that is not a SC process), during its lifetime a thread can temporarily moves from its base state machine to the state machine *method*$($*op*$)$ associated to the invoked operation[11], and then come back after completing the execution of the operation behavior. The state machine currently executed by t is given by

currStateMachine : *Sc_thread* $\rightarrow$ *StateMachine*

[9] Some other OCL constraints state that concurrent (or orthogonal) composite states (i.e. composite states with more than one region) and other pseudostates (like deepHistory, shallowHistory, entryPoint, join, fork, and exitPoint) of the UML2 are not allowed in the SystemC UML profile. Moreover, *internal* transitions and *deferred* events are also not allowed.

[10] A delta cycle is a very small step of time within the simulation, which does not increase the user-visible time.

[11] The mechanism for determining the method (behavior) to invoke as effect of an operation call is unspecified in the UML. In the SystemC UML profile [28], state machines designed similarly to the SCP state machines are associated to operations as behaviors to invoke when the operations are called. We assume, therefore, that threads temporarily execute such a kind of state machine diagram.

initially set to $\textit{baseStateMachine}(t)$, while the calling state machine is provided by

$$\textit{callMachine} : \textit{Sc\_thread} \times \textit{StateMachine} \rightarrow \textit{StateMachine}$$

In the current state machine of a thread, a state becomes active when it is entered as result of some fired transition, and becomes inactive when it is exited. All the composite states that either directly or transitively contain the active state are also active. The *current configuration* of active states w.r.t. the running state machine is given by

$$\textit{currState} : \textit{Sc\_thread} \times \textit{StateMachine} \rightarrow \mathcal{P}(\textit{Vertex})$$

The function $\textit{deepest} : \textit{Sc\_thread} \times \textit{StateMachine} \longrightarrow \textit{Vertex}$ yields the last (innermost) active state reached by a thread running its current state machine.

Event handling In the UML, the semantics of event occurrence processing is based on the *run-to-completion* assumption [33]. Since the event delivering and dispatching mechanisms are open in the UML, here they are explicitly modeled according to the discrete – *absolute* and *integer-valued* – time model of SystemC [32].

First, time is represented by an increasing monotonic function T. The domain `EventOccurr` represents the observable event occurrences (or event notifications) resulting from the execution of the processes. A function $\textit{type} : \textit{EventOccurr} \longrightarrow \textit{Event}$ returns the event type of a particular event occurrence. Event occurrences are collected in the global set *pendingEvents* and they are ordered by their time component $\textit{time}(e)$ with respect to the current simulation time T_c.

Second, each thread t is endowed with a queue, $\textit{eventQueue}(t)$, of event occurrences. One event is processed at a time by each thread. In the context of a thread execution, an event is dispatched when it is taken from the head of the event queue. At this point, the event is considered consumed and referred to as the current event.

Third, as delivering (or resuming) mechanism, it is assumed that the threads'event queues are also updated by a *scheduler* modeled as a separate agent. This special agent has the responsibility to place the events, upon their occurrence, into the queues of the processes that are sensitive to them. Threads' event queues are therefore shared functions. The behavior of the scheduler agent is not formalized here (though we are recalling here the ASM functions adopted for the interaction with the scheduler). For an in depth description, we refer the reader to [20], where an ASM formalization of the SystemC 2.0 scheduler is given, and to its implementation in AsmetaL available at [16]. According to the scheduler formalization in [20], a shared function $\textit{status}(t)$ ranging over the enumeration {*READY, EXECUTING, SUSPENDED*} is used to manage a thread life cycle. A thread is selected for execution by the scheduler, one after the other, from the set of *ready* processes. The set of all processes sensitive to an event type e is given by the function $\textit{processes}(e) : \textit{Event} \longrightarrow \textit{Sc\_thread}$. Upon an event occurrence, the scheduler examines the process list of the event type to determine the processes (threads) to which deliver the event occurrence and turn them ready in case they were waiting for that event. An event occurrence can be explicitly required to be immediate in the current delta cycle, or for future time cycles.

In the SystemC UML profile, events can essentially be signal events (the `SignalEvent` class in the UML metamodel) or time events (the `TimeEvent` class). Signal events represent the receipt of asynchronous *sc_event* signals (as stereotyped), and are generated as a result of some *notify* actions (stereotyped `SendSignalAction`) executed by other processes (other threads or methods processes) either within the context module (or a hierarchical channel) or in the surrounding environment. Time events are timeouts caused by the expiration of a time deadline always relative to the time of entry of the thread into a wait-state. *Completion events* (which originate from UML rather than SystemC) are directly handled by threads.

Finally, a function *dispatched*(t) yields for a thread t the head element of the thread's event queue to indicate the dequeued event to be processed[12]. At any moment, for a thread t the only transition $trans$ that is eligible to fire when an event e occurs is the one departing from the deepest active state of t, with an associated guard (if any) evaluating to true (*eval*$(g,trans) = true$),and with e triggering $trans$. This is expressed by the following function:

$$enabled(e,t) = \begin{cases} trans \text{ if } triggering(trans,e,t) \\ undef\ otherwise \end{cases}$$

where *triggering*$(trans, e, t)$ is a derived predicate defined as follows:

$$\begin{aligned} \textit{triggering}(trans, e, t) \equiv \textit{source}(trans) = \textit{deepest}(t, currStateMachine(t))\ \&\\ \textit{eval}(guard(trans), trans)\ \&\ \textstyle\bigvee_i p_i(trans, e, t) \end{aligned}$$

Each p_i(*trans,e,t*) formalizes a different case of the semantics of dynamic resuming transitions (ranging, see Fig. 3, over timeout, events, and AND/OR lists of events), static resuming transitions, and completion transitions. In case $trans$, for example, is an AND-dynamic resuming transition, p_i(*trans,e,t*) holds if and only if *e* is in *trigger*$(trans)$ and all the other events in *trigger*$(trans)$ have already been notified to t

$$\begin{aligned} &\textit{isAnd}(trans)\ \&\ (\exists\ e' \in \textit{trigger}(t) : \textit{event}(e') = \textit{type(e)})\ \&\\ &(\forall\ e'' \in \textit{trigger}(trans), \textit{event}(e'') \neq \textit{type(e)} : \textit{event}(e'') \in \textit{andHist}(t,trans)) \end{aligned}$$

4.2 ASM transition system

This section describes the ASM semantics of the *run to completion step* of the SystemC thread state machines. The behavior of a thread consists of the two rules *TransitionSelection* and *GenerateCompletionEvent* for simultaneously (i) selecting the machine transition to be executed next, and (ii) generate completion events. The next paragraph defines the exact meaning of "executing a state machine" by a parameterized macro rule *stateMachineExecution.*

In the *TransitionSelection* rule, a check is done in parallel to the machine execution for treating *dynamic resuming* transitions with an AND-semantics (the OR-semantics is the default): an AND-transition may fire when all the labeling triggers have been effectively triggered – not necessarily all in the same delta-cycle or at the same time –, and in this case the history of the occurrences of its AND-list of events is reset to empty. If a dispatched event does not trigger any transition in the current state of a thread, it is lost unless (the **else** branch) it must be collected in the history of an AND-transition.

```
**rule** $TransitionSelection(t) =$
**if** $status(t) = EXECUTING$
**then let** $e = dispatched(t),\ trans = enabled(e,t),\ s = deepest(t, currStateMachine(t))$
```

[12] It should be noted that at this point the ASM model differs from the one in [1] since the mechanism here for selecting the event to consume is deterministic.

```
in if $trans \neq undef$
  then par
        $stateMachineExecution(t,trans)$
        if $isAnd(trans)$ then $andHist(t,trans) := []$
  else if $isAndWait(s,e)$ then $andHist(t,trans) := add(e,andHist(t,trans))$
```

where $isAndWait(s,e) \equiv isWait(s)\ \&\ (\exists\ trans \in outgoing(s) \mid isAnd(trans)\ \&\ e \in trigger(trans))$

Completion events are generated by a thread when an active state satisfies the *completion condition* [33]. This is formalized similarly as in the ASM model in [1] by a rule *GenerateCompletionEvent* parameterized with *t*, with the only difference that the completion event generated is added to the head of the thread event queue.

The rule macros This paragraph reports only a very small subset of the rule macros used in the top level rules. The subrule *stateMachineExecution* is described first. It formalizes the run-to-completion semantics which consists into sequentially executing: (a) the exit actions of the source state and of any enclosing state up to, but not including, the least common ancestor *LCA* (i.e. the innermost composite state that encloses both the source and the target state), innermost first (see macro *exitState*); (b) the action on the transition; (c) the entry actions of any enclosing state up to, but not including, the least common ancestor, outermost first (see macro *enterState*); finally, (d) the "nature" of the target state is checked and the corresponding operations are performed.

```
macro rule $stateMachineExecution(t,trans)$ =
seq
   $exitState(source(trans),ToS,t)$
   $execute(effect(trans),t)$
   $enterState(FromS,target(trans),t)$
   case $target(trans)$
       $isSimple$: $enterSimpleState(target(trans),t)$
       $isComposite, isSubmachine$ : $enterCompositeState(target(trans),t)$
       $isWait$: $enterWaitState(target(trans),t)$
       $isStatic\_wait, isDont\_initialize$: $enterStaticWaitState(target(trans),t)$
       $isFinal, isIf, isEndif, isEndswitch$: $enterNextState(target(trans),t)$
       $isSwitch$: $enterSwitchState(target(trans))$
       $isWhile, isDowhile, isFor$: $enterLoopState(target(trans),t)$
       $isReturn$: $enterReturnState(target(trans),t)$
       $isBreak$: $enterBreakState(target(trans),t)$
       $isContinue$: $enterContinueState(target(trans),t)$
       $isExit$: $enterExitState(target(trans),t)$
   endcase
```

where *ToS* is the direct sub-state of the *LCA* in the nested state chain from $source(trans)$ to *LCA*; while, *FromS* is the direct sub-state of the *LCA* in the nested state chain from *LCA* to $target(trans)$.

Macros *exitState* and *enterState* are formalized similarly as in [1]. The exits from nested states should be performed in an order that respects the hierarchical structure of the machine. Starting from the deepest state up to, but excluding, the source/target least common ancestor state, innermost first, a thread sequentially (i) executes the exit actions (if any)[13], and (ii) removes those states from the thread's current state and, when states are exited, their enclosed final state (if any).

```
macro rule exitState(s,v,t) =
loop through $S \in$ UpChain($s$,$v$)
seq
    if $\neg$ isPseudoState($S$) then execute(exit($S$),$t$)
    currState($t, currStateMachine(t)$) := remove($S$,currState($t, currStateMachine(t)$))
```

Similarly, for entering nested states, any state enclosing the target one up to, but excluding, the least common ancestor will be entered in sequence, outermost first. Entering a state means that (a) the state is activated, i.e. inserted in *currState*(*t, currStateMachine(t)*), (b) its entry action (if any) is performed, and (c) the state internal activity (if any) is started.

```
macro rule enterState(s,v,t) =
loop through $S \in$ DownChain($s$,$v$)
seq
    enterNextState($S$,$t$)
    if $\neg$ isPseudoState($S$)
    then seq
            execute(entry($S$),$t$)
            execute(doActivity($S$),$t$)
```

where *enterNextState*(*s*,*t*) $\equiv$ *currState*(*t, currStateMachine*(*t*)) $:=$ *insert*(*s*,*currState*(*t, currStateMachine*(*t*))).

Macros for entering vertices depending on their specific nature are completely defined in [10]. We here report only that for entering a wait-state. When the target state of the triggered transition is a wait state, the thread is suspended as follows. The thread inserts itself in the process list of all events e appearing as triggers in the outgoing transitions of the wait-state, and changes its status to *suspended*. The thread will be turned *ready* by the scheduler when an event that the thread is waiting for will be notified. In case of timeout, i.e. an outgoing transition with a time event, the thread creates an event occurrence with time *timeout*$+T_c$ and adds it to the set of pending events.

```
macro rule enterWaitState(s,t) =
        if $\exists\, e' \in$ trigger(outgoing($s$)) : isTimeEvent($e'$)
        then extend EventOccurr with $e$
                time($e$) := $T_c +$ eval(when(event($e'$)))
                type($e$) := event($e'$)
                pendingEvents := add(pendingEvents,$e$)
        forall $e \in$ trigger(outgoing($s$)) do processes(event($e$)) := add(processes(event($e$)),$t$)
        status(t) := SUSPENDED
```

where *when(e)* for a time event *e* is an expression specifying a relative instant in time.

[13] Note there is no reason to stop the internal ongoing activities (if any) before exit, since the only outgoing transitions from a non stereotyped state are completion transitions.

4.3 ASM initial state

The initial state is the result of the mapping $\iota(\text{SC-UML})$, defined by the HOT ι applied to a terminal SystemC-UML model. It provides the initial values of domains and of (dynamic) controlled functions of the ASM signature necessary to execute each process state machine (like the `stimgen` thread machine shown in Fig. 2) appearing in the terminal model. For the lack of space, the result of this final step is not reported here.

5 Related work

There are different ways currently used to specify the semantics of metamodel-based languages, and therefore of UML profiles. They mainly fall into the following categories.

(I) *Using natural languages* to describe language semantics informally.

(II) *Using the OCL [22]* and its extensions, see [4, 8, 7] to name a few, to specify static semantics through invariants and behavior through pre/post-conditions on operations; however, being side-effect free, the OCL does not allow the change of a model state, though it allows describing it.

(III) *Weaving behavior*. Recent works like Kermeta [19], xOCL (eXecutable OCL) [34], approaches in [29, 31], to name a few, propose ways of providing executability natively into metamodeling frameworks. A minimal set of executable actions is usually defined to describe (create/delete object, slot update, conditional operators, loops, local variables declarations, call expressions, etc.) behavioral semantics of metamodels by attaching behavior to classes operations Some approaches use imperative or objected-oriented (sub)languages, other use abstract pseudo-code. Furthermore, [9] provides an executable subset of standard UML (the Foundational UML Subset) to define the semantics of modeling languages such as the standard UML or extensions.

Although, these action languages aim to be pragmatic, extensible and modifiable, some of them suffer from the same shortcomings of traditional programming languages; indeed, a behavioral description written in one of such action languages has the same complexity of one (a program) written in a conventional programming language. Moreover, not all action semantics proposals are powerful enough to specify the model of computation (MoC) underlying the language being modeled and to provide such a specification with a clear formal semantics. As shown in [11, 23], when we illustrate a similar technique, the *weaving technique*, of our ASM-based framework, the ASMs formalism itself can be also intended as an action language but with a concise and powerful set of action schema provided by different ASM rule constructors.

(IV) *Translational semantics*, consisting in defining a mapping from the language metamodel to the abstract syntax of another language that is supposed to be formally defined. In [3], metamodels are *anchored* to formal models of computation built upon AsmL, a language to encode ASM models. In [6], the semantics of the AMMA/ATL transformation language is specified in XASM, an open source ASM dialect. Similar approaches based on this *translational* technique are UML-B [30] using the Event-B formal method, those adopting Object-Z like [17, 5], etc. Some previous works using ASMs to provide an executable and rigorous semantics of UML graphical sublanguages (statecharts, activity diagrams, etc.) [21, 1, 14, 8] fall in this category. These approaches can be intended as exemplifications of other translational techniques of the ASM semantic framework, as better described in [11]. The technique used here is also translational, but it is more general as it works at the "meta" level leading more automation, and therefore less user effort, and more reusable mappings and specifications.

(V) *Semantic domain modelling*, where a metamodel for the "semantic domain" – i.e. to express also concepts of the run-time execution environment – is defined, and then OCL rules are used to map elements of the language metamodel into elements of the semantic domain metamodel. This approach was used for the *CMOF Abstract Semantics* [18] and for the OCL [22]. We postponed as future step the evaluation of the effectiveness of the joint-use of this technique with the ASM formal method, as it requires a certain effort in modeling, at metamodel level, also the semantic domain.

6 Conclusions

This paper presents an executable formal semantics for the SCP state machines of the SystemC UML profile by exploiting the meta-hooking approach of the ASM-based semantic framework in [11]. Executability allows *semantics prototyping* to examine particular behavioral features of the profile and to check if the provided extensions of the UML metamodel are *conservative*, i.e. if their semantics does not contradict the UML semantics, and fix explicitly the *semantics variation points* intentionally left in the UML as leeway for the definition of domain-specific UML profiles. The comparison in [10] between the ASM model for the UML state machines in [1] and the ASM model for the SCPs described here shows that SCPs are effectively a conservative extension.

Formal ASM models obtained from graphical SystemC-UML models can potentially drive practical SoC model analysis like simulation, architecture evaluation and design exploration [12].

References

1. E. Börger, A. Cavarra, and E. Riccobene. Modeling the Dynamics of UML State Machines. In *Int. Workshop on Abstract State Machines, LNCS 1912*, pages 223–241. Springer, 2000.
2. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.
3. K. Chen, J. Sztipanovits, and S. Neema. Toward a semantic anchoring infrastructure for domain-specific modeling languages. In *ACM Conf. on Embedded Software*, pp 35–43, 2005.
4. B. Combemale et al. Towards a Formal Verification of Process Models's properties - SimplePDL and TOCL case study. In *9th Int. Conf. on Enterprise Information Systems*, 2007.
5. A. M. Mostafa et al. Toward a Formalization of UML2.0 Metamodel using Z Specifications. In *ACIS Int. Conf. on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, vol. 1, pp. 694–701, 2007.
6. D. Di Ruscio et al. Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs. Tech. Rep. 06.02, LINA, 2006.
7. S. Flake and W. Müller. A UML Profile for Real-Time Constraints with the OCL. In *Proc. of UML'02*, pages 179–195. Springer, 2002.
8. S. Flake and W. Müller. An ASM Definition of the Dynamic OCL 2.0 Semantics. In *Proc. of UML'04*, pages 226–240. Springer, 2004.

9. OMG. Semantics of a Foundational Subset for Executable UML Models, ptc/2008-11-03.
10. A. Gargantini, E. Riccobene, and P. Scandurra. A precise and executable semantics of the SystemC UML profile by the meta-hooking approach. DTI T.R. 110, Univ. of Milan, 2008.
11. A. Gargantini, E. Riccobene, and P. Scandurra. A semantic framework for metamodel-based languages. *J. of Automated Software Engineering*, 16(3-4), 2009.
12. A. Gargantini, E. Riccobene, and P. Scandurra. Model-driven design and ASM-based validation of embedded systems. *Behavioral Modeling for Embedded Systems and Technologies: Applications for Design and Implementation*, pages 24–54, July 2009.
13. D. Harel and B. Rumpe. Meaningful modeling: What's the semantics of "semantics"? *IEEE Computer*, 37(10):64–72, 2004.
14. J. Jürjens. A UML statecharts semantics with message-passing. In *Proc. of the 2002 ACM symposium on Applied computing*, pages 1009–1013. ACM Press, 2002.
15. I. Kurtev et al. Model-based DSL frameworks. In *21st ACM SIGPLAN conf. on Object-oriented programming systems, languages, and applications*, pp 602–616, 2006.
16. `https://asmeta.svn.sf.net/svnroot/asmeta/asm_examples/`.
17. H. Miao, L. Liu, and L. Li. Formalizing UML Models with Object-Z. In *Proc. of the 4th Int. Conf. on Formal Engineering Methods*, pages 523–534, London, UK, 2002. Springer-Verlag.
18. OMG. Meta Object Facility (MOF) 2.0, formal/2006-01-01, 2006.
19. P.-A. Muller et al. Weaving Executability into Object-Oriented Meta-Languages. In *Proc. of ACM/IEEE 8th Int. Conf. on Model Driven Engineering Languages and Systems*, 2005.
20. W. Müller, J. Ruf, and W. Rosenstiel. An ASM based SystemC simulation semantics. *SystemC: methodologies and applications*, pages 97–126, 2003.
21. I. Ober. More meaningful UML Models. In *TOOLS - 37 Pacific 2000*. IEEE, 2000.
22. OMG. Object Constraint Language (OCL), 2.0 formal/2006-05-01, 2006.
23. Elvinia Riccobene, Patrizia Scandurra. Weaving executability into UML class models at PIM level. In *Proc. of European Workshop on Behaviour Modelling in Model Driven Architecture (BM-MDA'09), Enschede, The Netherlands, June 23, 2009, ACM Vol. 379, 10–27*.
24. E. Riccobene, P. Scandurra, S. Bocchio, and A. Rosti. An Enhanced SystemC UML Profile for Modeling at Transaction-Level. *Embedded Systems Specification and Design Languages. E. Villar (ed.), 2008*.
25. E. Riccobene, P. Scandurra, S. Bocchio, and A. Rosti. A SoC Design Methodology Based on a UML 2.0 Profile for SystemC. In *Proc. of Design Automation and Test in Europe*. IEEE Computer Society, 704–709, 2005.
26. E. Riccobene, P. Scandurra, S. Bocchio, and A. Rosti. A model-driven design environment for embedded systems. In *Proc. of the 43rd Design Automation Conference*. ACM Press, New York, NY, USA, 915–918, 2006.
27. E. Riccobene, P. Scandurra, S. Bocchio, and A. Rosti. A Model-driven co-design flow for Embedded Systems. *Advances in Design and Specification Languages for Embedded Systems (Best of FDL'06), 2007*.
28. E. Riccobene, P. Scandurra, S. Bocchio, A. Rosti, L. Lavazza, and L. Mantellini. SystemC/C-based model-driven design for embedded systems. *ACM TECS*, 8(4), 2009.
29. M. Scheidgen and J. Fischer. Human comprehensible and machine processable specifications of operational semantics. In *ECMDA-FA 2007, LNCS 4530*, pages 157–171. Springer.
30. C. Snook and M. Butler. UML-B: Formal modeling and design aided by UML. *ACM Trans. Softw. Eng. Methodol.*, 15(1):92–122, 2006.
31. M. Soden and H. Eichler. Towards a model execution framework for Eclipse. In *Proc. of the 1st Workshop on Behavior Modeling in Model-Driven Architecture*. ACM, 2009.
32. SystemC Language Reference Manual. IEEE Std 1666, 2006.
33. OMG. The Unified Modeling Language (UML), v2.2. `http://www.uml.org`, 2009.
34. The Xactium XMF Mosaic. `www.xactium.com/`, 2007.