MAPPING UML INTO ABSTRACT STATE MACHINES: A FRAMEWORK TO SIMULATE UML MODELS

ALESSANDRA CAVARRA, ELVINIA RICCOBENE, AND PATRIZIA SCANDURRA

Abstract. The paper presents a simulation framework for UML models based upon a mapping schema of UML metamodel elements into Abstract State Machines (ASMs). Structural model elements are translated into an ASM vocabulary as collections of domains and functions, whereas the dynamic view is captured by multi-agent ASMs reflecting the behavior modelled by UML state machines.

In our toolkit, input UML models can be drawn using any UML CASE Tool able to produce the XMI format for diagrams. This textual representation is employed to initialize the ASM model for state machines which can be symbolically executed by AsmGofer, an advanced Abstract State Machine programming system.

We present the tool through an example of simulation of a simple stack-printer UML model showing the interactions among state machines by signals exchange and operation calls.

Keywords: UML, Abstract State Machines, Model Mapping, Simulation of UML Models.

1. Introduction

The Unified Modeling Language (UML) [UMLa] has been widely accepted as the new standard for modeling software systems and is supported by a great number of CASE tools. However, UML tools often provide only little support to symbolically execute models.

Simulation of models gives developers a very efficient way to verify and validate their design models, and even their analysis models. By symbolically executing the system, the developers can ensure that the behavior specified satisfies his/her intent, and can find errors at the very early stages of development, thus saving huge amounts of time later on by reducing the need for redesign work and the amount of programming bugs.

Many commercial UML tools support drawing of UML diagrams and model documentation provide automatic generation of class definitions and function prototyping from class diagrams, allow reverse engineering, but very few tools are suitable for simulation. The difficulty of developing UML models simulators is strongly related to the lack of precise semantic models which can serve as reference model for implementing tools.

A usual argument against the definition of a precise semantics for UML is the fact that UML shall be applicable on various domains with contradictory demands for the semantics of the same entity. However, the lack of a rigorous semantics leads to a more apparent than real understanding of models, difficulty to perform rigorous analysis, validation, verification, and to check integrity of models. To endow UML with a formal semantics while keeping it domain independent, a solution could be to adopt a formalism abstract enough to avoid problems related to over specification. Abstract State Machines (ASMs) [asm] provide a nice symbiosis between expressive power and high level of abstraction.

This work comes as part of a formalization effort necessary to develop a framework for mechanical validation and verification for a subset of the UML in the form as required by the official UML documentation. In [BCR00, BCR] we provide a complete ASM model for UML state machines, which is further refined in [CRS] by formalizing objects communication mechanism (i.e. signals exchange and operation calls) in order to integrate state machines with other UML structural and behavioral diagrams. Observe that, although our work was originally based on UML 1.4, the subset of UML we have formalized has not been modified and therefore it is still valid.

In this paper we present a simulation framework for UML models based upon a mapping schema of UML metamodel elements into Abstract State Machines. Structural model elements are translated into an ASM vocabulary as collections of domains and functions, whereas the dynamic view is captured by multi-agent ASMs reflecting the behavior modeled by UML state machines.

In our toolkit, input UML models can be drawn using any UML CASE Tool able to produce the XMI format for diagrams (i.e. Poseidon [Pos]) and can include class diagrams to define model entities, object diagrams to provide an initial configuration of the system, state diagrams to show objects behavior in response to events received, and sequence diagrams to depict possible interaction scenarios among objects. This textual representation is then used to initialize the ASM model for state machine which can be symbolically executed by AsmGofer [Sch], an advanced Abstract State Machine programming system.

The possibility to simulate the ASM specification for UML state machines allowed us to discover inconsistencies in the model arising from conflict situations never mentioned or discussed in the official documentation, and also to find out other small errors due to our incorrect interpretation of some statechart features. We refer the reader to [BCR] for a complete and exhaustive discussion on conflicting transitions and the most relevant changes made in the model [BCR00] to solve faults arisen during the simulation.

The paper reviews the basic concepts underlying UML in section 2. Section 3 gives an overview of Abstract State Machines. The mapping schema used to translate input UML models into ASMs is presented in section 4. Section 5 describes the toolkit architecture, while section 5.1 presents an example of simulation through the execution of a simple stack-printer UML model. Related work and future directions are given in section 6 and 7.

2. A subset of the UML

Although UML provides a wide range of notations, to describe fully the structure and functionalities of a system it is sufficient to use class diagrams and state diagrams. Class diagrams show collections of declarative elements in the system and their relationships. State diagrams specify the life-cycle of the objects of a class, describing the possible sequences of states and actions through which an object can go during its lifetime as a result of reacting to discrete events. Moreover, interactions among objects are modeled by means of interaction diagrams which specify how and when messages are sent between objects to perform a task.

In the following paragraphs, we briefly describe the comprehensive set of UML diagrams to describe the structure and functionalities of a system and which we will consider in our simulation framework.

UML Class and Object Diagrams. A class diagram (see Fig. 2) describes the static structure of a system, consisting of a number of classes and their relationships. The application concepts are modelled in UML as classes each of which describes a set of objects that hold information and *communicate* to implement a behavior. The information they hold is modelled as attributes; the behavior they perform is modelled as operations. Structural relationships between objects of different classes are represented by *associations*. The definition of associations may be enhanced by a name, role names and cardinality (multiplicity).

An object diagram (see Fig. 3) is a graph of instances of the entities represented in a class diagram, namely objects and links. They model the actual configuration of the system at a certain time.

UML State Diagrams. State diagrams focus on the event-ordered behavior of an object, a feature which is specially useful in modeling reactive systems. A state diagram shows the event triggered flow of control due to transitions which lead from state to state, i.e. it describes the possible sequences of states and actions through which a model element can go during its lifetime as a result of reacting to discrete events. A state reflects a situation in the life of an object during which this object satisfies some condition, performs some action, or waits for some event. According to the UML meta-model [UMLb], states can belong to one of the following categories: *simple states*, *composite states* (sequential, concurrent, submachine), *final*, and *pseudostates* (initial, history, stub, junction, synch).

Transitions are viewed in UML as relationships between two states indicating that an object in the first state will enter the second state and perform specific actions when a specified event occurs provided that certain conditions are satisfied. UML statecharts include *internal*, *external* and *completion* transitions.

The semantics of event processing in UML state machines is based on the *run to completion* (rtc) assumption: events are processed one at a time and when the machine is in a stable configuration, i.e. a new event is processed only when all the consequences of the previous event have been terminated. Therefore, an event is never processed when the state machine is in some intermediate, unstable situation.

Events may be specified by a state as being possibly deferred. They are actually deferred if, when occurring, they do not trigger any transition. This will last until a state is reached where they are no more deferred or where they trigger a transition. Fig. 5 shows an example of state diagram.

UML Sequence Diagrams. An interaction is a unit of behavior that focuses on the observable exchange of information between objects. Sequence diagrams show possible interaction scenarios among objects that communicate with one another by exchanging messages to invoke a desired operation or result. Interactions are arranged in explicit time sequence of stimuli (see Fig. 6).

3. Abstract State Machines

The Abstract State Machines [BS03] represent a systems engineering method which allows the development of embedded hardware/software systems seamlessly from requirements capture to their implementation. The method bridges the gap between the human understanding and formulation of real-world problems and the deployment of their algorithmic solutions by code-executing machines on changing platforms. The method has a simple scientific foundation, which adds precision to the method's practicality.

The notion of ASMs moves from a definition which formalizes simultaneous parallel actions of a single agent, either in an atomic way (*Basic ASMs*) and in a structured and recursive way (*Turbo ASMs*), to a generalization where multiple agents act and interact in an synchronous/asynchronous manner (*Synchronous/Asynchronous Multi-Agent ASMs*). They also allow non-determinism (**choose** or existential quantification) and unrestricted synchronous parallelism (universal quantification **forall**).

We quote here only the essential *working* definitions of the ASMs, i.e. in a form which justifies their intuitive understanding as pseudo-code over abstract data, as defined in [BS03]. For a detailed mathematical definition for the syntax and semantics of ASMs, we refer to section 2.4 of [BS03].

ASMs are transition systems. The *states* of an ASM are multi-sorted first-order structures, i.e. domains of objects with functions and predicates defined on them. The *transition relation* is specified by "rules" describing the modification of the functions from one state to the next.

The basic form of a transition rule is the following *function update*

$$f(t_1, \ldots, t_n) := t$$

where f is an arbitrary n-ary function and $t_1, \ldots, t_n, t$ are first-order terms. To fire this rule to a state S_i, $i \geq 0$, evaluate all terms $t_1, \ldots, t_n, t$ at S_i and update the function f to t on parameters $t_1, \ldots, t_n$. This produces another state S_{i+1} which differs from S_i only in the new interpretation of the function f.
There are some rule constructors.

The *conditional constructor* produces "guarded" transition rules of the form:

$$\textbf{if } g \textbf{ then } R_1 \textbf{ else } R_2$$

where g is a ground term (the *guard* of the rule) and R_1, R_2 are transition rules. To fire a guarded rule to a state S_i, $i \geq 0$, evaluate the guard; if it is true, then execute R_1, otherwise execute R_2. The **else** part may be omitted.

The *parallel constructor* produces the "parallel" synchronous rule of form:

$$\textbf{do forall } x : x \in U \; R(x)$$

where $R(x)$ is a transition rule with a variable x ranging over the universe U. The operational meaning of a parallel rule consists in the execution of $R(x)$ for every $x \in U$.

In order to specify some systems which are characterized by sequential operations, we may need a sequential application of rules. We make use of the *sequential constructor*:

$$\textbf{seq } R_1 \ldots R_n$$

Each rule R_i is fired only after R_{i-1}, $i = 2, 3, \ldots, n$, and causes the transition to a new state.

If we need a sequential transition with more than one rule per each step, we can use the *block constructor*,

$$\textbf{do in-parallel } R_1 \ldots R_n \textbf{ enddo}$$

to apply $R_1, \ldots, R_n$ simultaneously, inside a **seq** constructor.

We also make use of the following construct to let universes grow:

$$\textbf{extend } U \textbf{ with } x_1, \ldots, x_n \textbf{ with } Updates$$

where *Updates* may (and should) depend on the x_i and are used to define certain functions for (some of) the new objects x_i of the resulting universe U.

To avoid the trouble of writing a lengthly term t several times in a rule, we use the constructor,

$$\textbf{let } x = t \ R(x)$$

The **let** rule above is equivalent to the rule $R(x \mapsto t)$ resulting by substituting t for x.

In ASM it is possible to express the non-determinism using the *choose constructor*:

$$\begin{array}{l}\textbf{choose } x \textbf{ in } U \texttt{ s.t. } cond(x) \\ R(x)\end{array}$$

which allows to fire the rule $R(x)$ choosing randomly an x in U satisfying conditions *cond*(x).

An ASM M is a finite set of rules for such guarded multiple function updates. State transitions of M may be influenced in two ways: *internally*, through the transition rules, or *externally* through the modifications of the environment. A non static function f that is not updated by any transition rule is called *monitored function* and its values are given non-deterministically by the environment. Non static functions updated by a transition rule are called *controlled functions*.

A *computation* of M is modeled through a finite or infinite sequence $S_0, S_1, \ldots, S_n, \ldots$ of states of M, where S_0 is an initial state and each S_{n+1} is obtained from S_n by firing simultaneously all of the transition rules which are enabled in S_n.

ASMs cover within a single conceptual framework both design and analysis, for procedural *single-agent* and for *multiple-agent distributed* systems. Multi-agent ASMs may have agents running in a synchronous way, or asynchronous agents that proceed in parallel at their own speed and whose communications may provide the only logical ordering between their actions.

A *multi-agent synchronous ASM* is defined as a set of agents which execute their own ASMs in parallel, synchronized using an implicit global system clock. Semantically a sync ASM is equivalent to the set of all its constituent single-agent ASMs, operating in the global states over the union of the signatures of each component.

An *multi-agent asynchronous ASM* is given by a family of pairs $(a, \mathit{ASM}(a))$ of pairwise different agents, elements of a possibly dynamic finite set *Agent*, each executing its basic or structured ASM $\mathit{ASM}(a)$.

A multi-agent ASM with synchronous agents has *quasi-sequential runs*, namely a sequence of states where each state is obtained from the previous state by firing in parallel the rules of all agents.

A multi-agent ASM with asynchronous agents has *partially ordered runs*, namely a partially ordered set $(M, <)$ of moves *m* (read: rule applications) of its agents satisfying the following conditions: (a) *finite history*: each move has only finitely many predecessors, i.e. for each $m \in M$ the set $\{m' \mid m' < m\}$ is finite; (b) *sequentiality of agents*: the set of moves $\{m \mid m \in M, a \text{ performs } m\}$ of every agent $a \in \mathit{Agent}$ is linearly ordered by $<$; (c) *coherence*: each finite initial segment (downward closed subset) X of $(M, <)$ has an associated state $\sigma(X)$ – think of it as the result of all moves in X with m executed before m' if $m < m'$ – which for every maximal element $m \in X$ is the result of applying move m in state $\sigma(X - \{m\})$.

We use *multi-agent ASMs* [Gur95] to model the concurrent substates and the internal activities which may appear in a UML state machine.

Since ASMs offer the most general notion of state, namely structures of arbitrary data and operations which can be tailored to any desired level of abstraction, this allows us on the one side to reflect in a simple and coherent way the integration of control and data structures, resulting from mapping state machines to the UML object model. In fact, machine transitions are described by ASM rules where the actions become updates of data (function values for given arguments). On the other side also the interaction between objects is naturally reflected by the notion of state of multi-agent (distributed) ASMs.

Futhermore, the constructs of sequentialization, iteration and submachine of sequential ASMs, as defined in [BS03], provide the concept of “stable” state needed to guarantee that the event triggered sequential exit from and entry into nested diagrams is not interrupted by a too early occurrence of a next event.

ASMs are also supported by tools for model simulations [Sch, oSEG, AK], specification-based testing [AG01, AG03], mechanical verification by means of theorem proving (PVS) [AG00] and model checking (SMV, Spin) [AG03] techniques. Furthermore, ASMs have also been successfully exploited for deriving executable code from models through a sequence of refined abstract machine models up to a point where it becomes evident how executable code can be obtained by translating –almost mechanically– the abstract machine instructions into code-procedures [Sch01, EB00].

4. Mapping UML models into ASM

Our approach consists on mapping every UML model into a multi-agent ASM model. Fig. 1 illustrates the schema adopted to map UML (meta) model elements onto ASMs.
The architecture of the UML is based on a four-layer metamodel structure, which consists of the following layers [UMLb]:

- **meta-metamodel**: This layer forms the foundation for the metamodeling architecture. The primary responsibility of this layer is to define the language for specifying a metamodel.
- **metamodel**: A metamodel is an instance of a meta-metamodel. The primary responsibility of the metamodel layer is to define a language for specifying models. Examples of metaobjects in the metamodeling layer are: Class, Attribute, Operation.
- **model**: A model is an instance of a metamodel. The primary responsibility of the model layer is to define a language that describes an information domain. Examples of objects in the modeling layer are: Stack, buffer, pop.

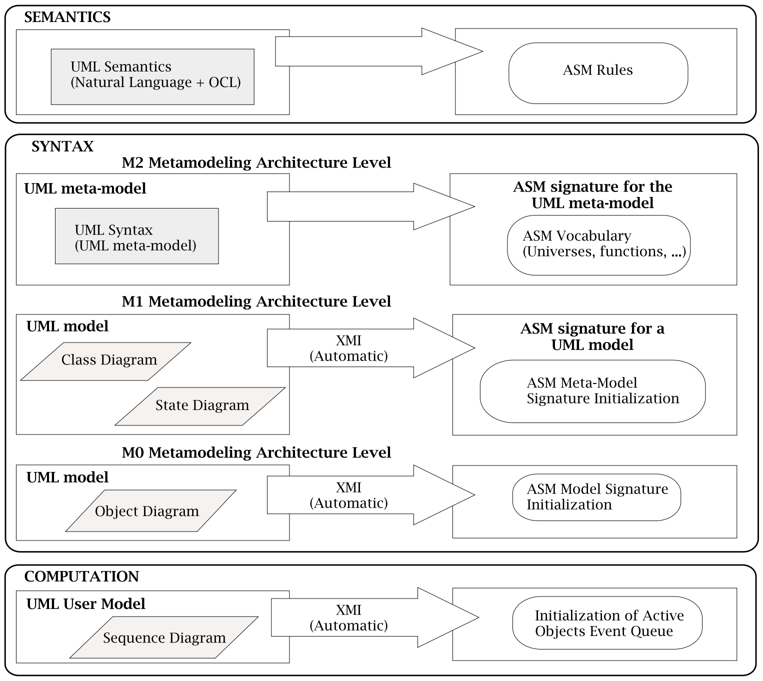

SEMANTICS
UML Semantics (Natural Language + OCL)
ASM Rules
SYNTAX
M2 Metamodeling Architecture Level
UML meta-model
UML Syntax (UML meta-model)
ASM signature for the UML meta-model
ASM Vocabulary (Universes, functions, ...)
M1 Metamodeling Architecture Level
UML model
Class Diagram
State Diagram
XMI (Automatic)
ASM signature for a UML model
ASM Meta-Model Signature Initialization
M0 Metamodeling Architecture Level
UML model
Object Diagram
XMI (Automatic)
ASM Model Signature Initialization
COMPUTATION
UML User Model
Sequence Diagram
XMI (Automatic)
Initialization of Active Objects Event Queue

Figure 1: UML to ASM mapping schema

- **user objects**: User objects (a.k.a. user data) are an instance of a model. The primary responsibility of the user objects layer is to describe a specific information domain.

This mapping addresses the metamodel, model, and user objects layers. The UML notation defined at the metamodel level has been manually translated *once for all* into a corresponding ASM signature, i.e. universes, functions, and predicates.

This mapping schema has been implemented in our tool in order to automatically initialize the ASM signature from the input UML "model". In particular, class diagrams provide information regarding declarative UML elements (classes, operations, relationships,...); object diagrams are used to define the multi-agent ASM model initial states. The ASM signature formalizing the state diagram elements (state, transitions, etc.) in the metamodel is automatically instantiated according to the state diagrams provided in the input model. An ASM model formalizing the semantics of UML state diagrams has been defined (again once for all) according to the formal ASM model presented in [BCR00, BCR].

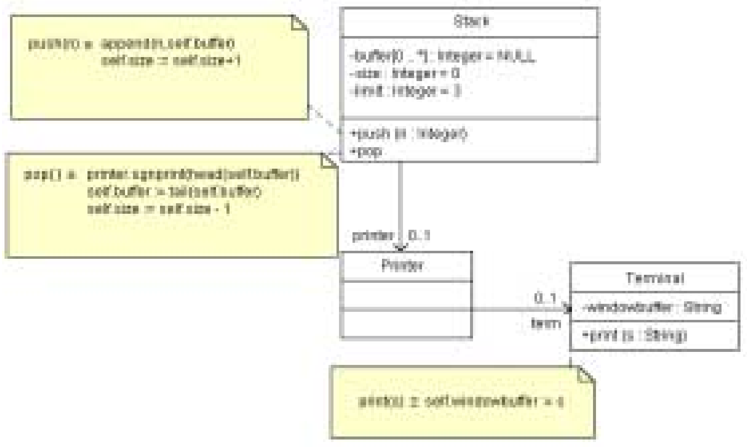

push(n) ≡ append(n,self.buffer)
self.size := self.size+1
Stack
-buffer[0..*]: Integer = NULL
-size : Integer = 0
-limit : Integer = 3
+push (n : Integer)
+pop
pop() ≡ printer.sgnprint(head(self.buffer))
self.buffer := tail(self.buffer)
self.size := self.size - 1
printer 0..1
Printer
0..1
term
Terminal
-windowbuffer : String
+print (s : String)
print(s) ≡ self.windowbuffer := s

Figure 2: Stack-Printer Class Diagram

Observe that we assume input models to be notationally well defined.

4.1. Mapping UML structural elements into ASMs

We present here the translation process adopted to map basic UML structural elements, such as classes, relationships, attributes and operations into ASM.

As example of static mapping we consider the UML model of the following stack-printer case study. A stack structure, of limited size, is an active object which is used by the environment to *push* and *pop* integer elements. The stack signals its state of underflow or overflow by sending a message of *stackEmpty* or *stackFull* to a printer. The printer is always ready to print all messages on a terminal. The terminal has a buffer *windowbuffer* of type String in which messages from the printer are showed.

Fig. 2, 3, 5, and 4 respectively show the class diagram, the object diagram, the state machine of the stack component, and the state machine of the printer component of the UML model of the stack-printer example developed using the CASE tool Poseidon [Pos].

UML metamodel. Each UML metaclass is mapped into a corresponding abstract set with the typewriting convention that a `MetaClass` is mapped into the ASM

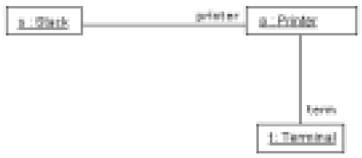


Figure 3: Stack-Printer Object Diagram

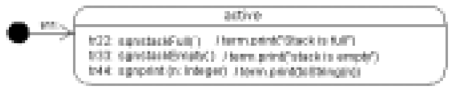

tr1:
active
tr22: signstackFull() / term.print("Stack is full")
tr33: signstackEmpty() / term.print("stack is empty")
tr44: signprint (n: Integer) / term.print(toString(n))

Figure 4: Printer State Machine

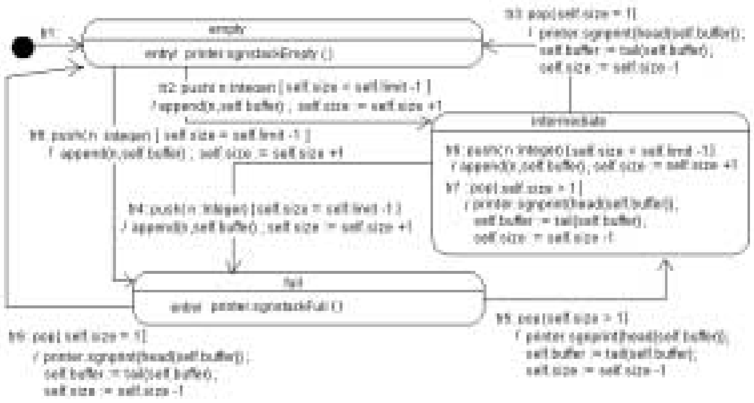

tr1:
empty
entry/ printer.signstackEmpty ()
tr3: pop[self.size = 1]
/ printer.signprint(head(self.buffer));
self.buffer := tail(self.buffer);
self.size := self.size -1
tr2: push(n: integer) [self.size < self.limit -1]
/ append(n,self.buffer) ; self.size := self.size +1
intermediate
tr8: push(n: integer) [self.size = self.limit -1]
/ append(n,self.buffer) ; self.size := self.size +1
tr5: push(n: integer) [self.size < self.limit -1]
/ append(n,self.buffer); self.size := self.size +1
tr7 : pop[self.size > 1]
/ printer.signprint(head(self.buffer));
self.buffer := tail(self.buffer);
self.size := self.size -1
tr4: push(n: integer) [self.size = self.limit -1]
/ append(n,self.buffer) ; self.size := self.size +1
full
entry/ printer.signstackFull ()
tr6: pop[self.size = 1]
/ printer.signprint(head(self.buffer));
self.buffer := tail(self.buffer);
self.size := self.size -1
tr9: pop[self.size > 1]
/ printer.signprint(head(self.buffer));
self.buffer := tail(self.buffer);
self.size := self.size -1

Figure 5: Stack State Machine

universe *MetaClass*, i.e. the metaclass `StateMachine` is mapped into the abstract set *StateMachine* of state machines, the metaclass `State` is mapped into the set *State*, `Transition` into *Transition*, `Event` into *Event*, `Action` into *Action*, etc.

A composition relation between metaclasses is mapped into a composition relation between the corresponding abstract sets. Therefore, the set *State* is partitioned into the sets *SimpleState*, *SequentialState*, *ConcurrentState*, *PseudoState*; the set *Event* is partitioned into the sets *SignalEvent*, *CallEvent*, *TimeEvent*, *ChangeEvent*;

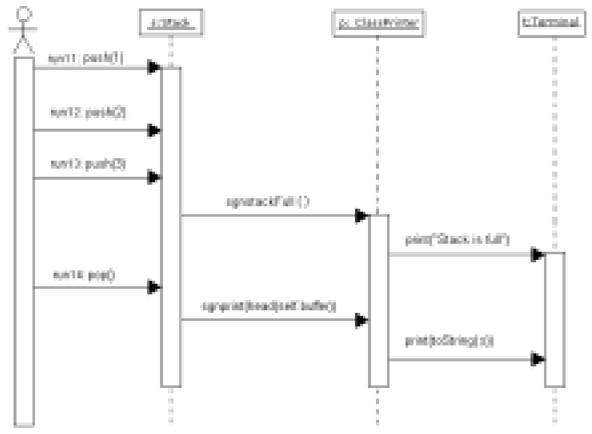


Figure 6: Stack-Printer Sequence Diagram

Action is partitioned into *CallAction*, *SendAction*, *CreateAction*, *DestroyAction*, *UninterpretedAction*; etc.

According to the UML Meta-Model for state machines, the ASM domain *StateMachine* has elements are of type

$$\mathit{statemachine}(\mathit{name},\mathit{init},\mathit{state},\mathit{transition},\mathit{event})$$

where $\mathit{name} \in \mathit{String}$ is the state diagram name, $\mathit{init} \in \mathit{State}$ is the initial state of the diagram, $\mathit{state} \in \mathcal{P}(\mathit{State})$ denotes the set of states in the diagram, $\mathit{transition} \in \mathcal{P}(\mathit{Transition})$ represents the set of (both internal and external) transitions in the diagram, and $\mathit{event} \in \mathcal{P}(\mathit{Event})$ is the set of events labeling the transitions in the diagram.

UML models. Information provided by the diagrams at the Model level, are used to initialize the abstract sets corresponding to the meta-classes at the Meta-Model level.

Let $\texttt{Class}_1, \ldots, \texttt{Class}_n$ the set given by the classes defined in all the class diagrams. For each $\texttt{Class}_i$, we define a *dynamic* domain Class_i in the ASM model, whose elements are the instances of the corresponding class in the object diagrams (and will be initialized at User Model level).

Attributes are formalized as ASM functions, whose domain is the ASM universe corresponding to the class whom the attribute belongs to, and the co-domain is the attribute's type itself. *Operations* are represented by ASM rules parameterized on the universe corresponding to the class encompassing the operation and on the universes corresponding to the types of the operation parameters[1].

Each *association* navigable from $\texttt{Class}_i$ to $\texttt{Class}_j$ is represented as an ASM function having the universe $Class_i$ as domain and $Class_j$ as co-domain. If the association is navigable in both directions, then the inverse function is defined as well. A *generalization* between $\texttt{Class}_k$ and $\texttt{Class}_h$ is mapped into an inclusion relation between $Class_k$ and $Class_h$. Basic data types, such as Integer, String, Boolean, etc, are mapped into the corresponding ASM types. *Enumerations* will be translated as *static* ASM universes.

From class diagram in Fig. 2, universes *Stack*, *Printer*, *Terminal*, *Integer*, *String* are defined (as elements of the abstract set *Class*) together with the functions:

$$\begin{aligned}
&\textit{buffer, size, limit} : \textit{Stack} \rightarrow \textit{Integer*}\\
&\textit{printer} : \textit{Stack} \rightarrow \textit{Printer}\\
&\textit{term} : \textit{Printer} \rightarrow \textit{Terminal}\\
&\textit{windowbuffer} : \textit{Terminal} \rightarrow \textit{String}
\end{aligned}$$

Rules for operations of the class `Stack` are the following:

```
$push(self,e) \equiv$ do $insert(e,buffer(self))$
                         $size(self) := size(self) + 1$

$pop(self) \equiv$ do $send(printer(self),sgnprint,head(buffer(self)))$
                     $buffer(self) := tail(buffer(self))$
                     $size(self) := size(self) - 1$
```

where assignments are actions viewed as specializations of the metaclass `UninterpretedAction`. The operation of the class `Terminal` is defined by the rule:

$$print(self,s) \equiv windowbuffer(self,s) := s$$

Statecharts at Model level allow to initialize the abstract set *StateMachine* and the derived universes. From the state machines in Fig. 5 and 4, the universe *StateMachine* is initialized with the elements:

[1]We assume that operation parameters include possible return types.

$$\begin{aligned} &\textit{st1} = \textit{statemachine}(\textit{StackMachine, initState1}, \{\textit{empty,intermediate, full}\},\\ &\qquad \{\textit{tr1, tr2, tr3, tr4, tr5, tr8, tr9}\} \cup \{\textit{tr6,tr7}\}, \{\textit{push}(e{:}\textit{Integer}), \textit{pop}()\})\\ &\textit{st2} = \textit{statemachine}(\textit{PrinterMachine, initState2}, \{\textit{active}\}, \{\textit{tr11}\} \cup\\ &\qquad \{\textit{tr22, tr33, tr44}\}, \{\textit{print}(s{:}\textit{String}), \textit{stackEmpty}(), \textit{stackFull}()\}) \end{aligned}$$

UML user model. Diagrams developed at User Model level (Object Diagrams) allow to initialize the signature defined at Model level.

Each *dynamic* domain $Class_i$ in the ASM model takes as elements all the instances of the corresponding class in the object diagrams. For instance, from the object diagram in Fig. 3, sets *Stack*, *Printer*, *Terminal* are initialized as $\{s\}$, $\{p\}$, $\{t\}$, respectively. And functions on $Class_i$ are suitably initialized ($buffer(s) = []$; $size(s) = 0$; $limit(s) = 3$; … ; $term(p) = t$, $windowbuffer(p) =$ “”, etc.).

Let *Object* be the union of all sets $Class_i$ defined as above. If the behavior of a class is described by a state diagram[2], the function *classifier* : $Object \rightarrow \{Class_1, \ldots, Class_n\}$ associates to each object the class it belongs to, while the function *context* : $StateMachine \rightarrow \{Class_1, \ldots, Class_n\}$ associates to a state machine its context, i.e. the set of objects behaving according to it; *behavior* is the inverse function of *context*. Each element in *StateMachine* is associated to a class instance which constitutes the context of the machine by the function *stMachine* : $Object \rightarrow StateMachine$ where $stMachine(a) = behavior(classifier(a))$. According to our example, $stMachine(s) = st1$ and $stMachine(p) = st2$.

4.2. Mapping UML models behavior into ASMs

In the UML models under consideration here, the behavior of class instances is described by a state diagram attached to the class. In our ASM model, objects with state machine are modelled as ASM agents (therefore, $AGENT \subseteq Object$)[3] and their state diagrams are formalized as ASM modules of such agents. All the agents (objects) of the same type (class) execute the same module (state diagram). However, each agent runs on its own execution space and handles its own event

[2]It is possible to have classes without an associated state diagram.

[3]Since we do not distinguish between protocol and state machines, we associate to passive objects agents which are active *only* when stimulated by external events.

queue. The overall system behavior is represented by the multi-agent ASM whose initial states are given by the object diagrams. We partition the *eventQueue* of an agent into *ExtEventQueue* which is monitored by the environment and a *IntEventQueue* which is updated as result of *actions* of other agents.

According to our model of the Stack-Printer, *AGENT* = $\{s, p\}$, while t is a location on *Terminal*. The behavior of each agent is described by the state machine given by the function *stMachine*.

The ASM model for the behavioral meaning of state diagrams is defined in [BCR00, BCR, CRS]. This model (a) rigorously defines the UML event handling schema in a way which makes all its "semantic variation points" explicit, including the event deferring and the event completion mechanism; (b) encapsulates the run to completion step in *two* simple rules (**Transition Selection** and **Generate Completion Events**) where the peculiarities relative to entry/exit or transition actions and sequential, concurrent or history states are dealt with in a modular way; (c) integrates smoothly the state machine control structure with the data flow; (d) clarifies various difficulties concerning the scheduling schema for internal ongoing (really concurrent) activities; (e) describes all the UML state machine features that break the thread-of-control; (f) provides a precise computational content to the UML terms of atomic and durative actions/activities, without loosing the intended generality of these concepts, and allows one to clarify some dark but semantically relevant points in the UML documents on state machines.

For each UML state machine in the model, the set of objects defined in the context of the diagram move through the diagram each executing what is required by their *current state* (see [BCR00] for more details).

To fully understand the content of the following sections, a knowledge of the ASM state machines model in [BCR00, BCR] is strongly recommended. The semantics of actions and events is presented in [CRS]. Some relevant parts of such a model concerning the formalization of the run-to-completion-step are reported below.

4.2.1. The *Run To Completion* Step

In this section we formalize the run-to-completion-step semantics of the state machines by rules of an ASM. At the top level, a run-to-completion-step of a state machine consists in choosing an event, a transition enabled by it and then firing such a transition. Apparently, UML leaves it unspecified how to choose between dispatched and releasable events. We reflect this by using a selection function which, at any moment, chooses either a dispatched event triggering a transition, or an event that has been deferred. A dispatched event, if *deferrable*, has to be inserted into the *deferQueue*. A releasable event, when chosen for execution, has to be deleted from *deferQueue*. This implies that when choosing an event which is simultaneously *dispatched* and *releasable*, that event will be deleted from the deferred events.

The main rule for selecting the machine transition to be executed next can thus be stated as follows. For each agent $a \in \mathit{AGENT}$[4]:

```
Rule Transition Selection
choose $e$ : $\mathit{dispatched}(e) \vee \mathit{releasable}(e)$
    choose trans in $\mathit{FirableTrans}(e)$
        $\mathit{stateMachineExecution}(\mathit{trans})$
    if $\mathit{deferrable}(e)$ then $\mathit{insert}(e, \mathit{deferQueue})$
    if $\mathit{releasable}(e)$ then $\mathit{delete}(e, \mathit{deferQueue})$
```

The submachine *stateMachineExecution* is responsible for the execution of transitions (see below).

The rule for selecting and executing a transition fires simultaneously, at each “run to completion step”, with a rule to generate completion events.

Completion events are generated when an active state satisfies the *completion condition* [UMLb]. They trigger a transition outgoing such states. An active state is considered completed if one of the following cases occurs: (1) it is an active pseudostate, (2) it is a sequential composite state with active final state,

[4]The agents execute UML state machines, i.e. they all use the same program (or ASM *Rule*). As a consequence, in the formulation of functions and rules, we implicitly use the 0-ary function *Self* which is interpreted by each agent a as a.

(3) the state internal activity terminates while the state is still active, or (4) it is a concurrent composite state and all its direct substates have reached their final state. We formalize this by the following predicate

$$\begin{aligned}
\textit{completed}(S) = \textit{true} \Longleftrightarrow\ & \textit{PseudoState}(S)\ \textit{or}\\
& (\textit{SequentialState}(S)\ \&\ \textit{final}(S) \in \textit{currState})\ \textit{or}\\
& \textit{terminated}(A(S))\ \textit{or}\\
& (\textit{ConcurrentState}(S)\ \&\\
& \quad \forall S_i \in \textit{concurrentComp}(S)\ \forall a_i \in \textit{SubAgent}(\textit{Self})\\
& \qquad \textit{final}(S_i) \in \textit{currState}(a_i))
\end{aligned}$$

where *terminated*$(A(S))$ is a derived predicate that holds if and only if the *run* of the ASM $A(S)$, which formalizes the internal activity of S, reaches a final state. Each time the completion condition evaluates to true for an active state S that is not a direct substate of a concurrent state[5], a completion event is generated. This is expressed by the rule **Generate Completion Event** that is executed simultaneously for each state $S \in$ *currState*.

```
Rule Generate Completion Event
do forall $S \in \textit{currState}$
        if $\textit{completed}(S)\ \&\ \neg\ \textit{ConcurrentState}(\textit{UpState}(S))$
        then $\textit{generate}(\textit{completionEvent}(S))$
```

where *UpState*$(S))$ yields the direct substate of a compound state S.

Although the order of event dequeuing is not defined, it is explicitly required that completion events must be dispatched before any other queued events [UMLb]. We reflect this requirement as a constraint on the monitored predicate *dispatched*. The above two rules, which fire simultaneously at each *run to completion step*, define the top level behavior of UML state machines. It remains to define in more detail the meaning of the sub-rules appearing in those rules (see the complete model in [BCR00, BCR, CRS]).

[5]This restriction reflects that in UML no direct substate of a concurrent state can generate a transition event. Such substates are required to be sequential composite states.

The UML requirement that an object is not allowed to remain in a pseudostate, but has to immediately move to a normal state [RJB99], cannot be guaranteed by the rules themselves, but has to be imposed as an integrity constraint on the permissible runs.

State machine execution. The subrule *stateMachineExecution* with the parameterization by transitions allows us to modularize the definition for the different types of transitions and the involved states. If an internal transition is triggered, then the corresponding action is executed (there is no change of state and no exit or entry actions must be performed). Otherwise, if an external transition is triggered, we must determine the correct sequence of exit and entry actions to be executed according to the transition source and target state. Transitions outgoing from composite states are inherited from their substates so that a state may be exited because a transition fires that departs from some of its enclosing states. If a transition crosses several state boundaries, several exit and entry actions may be executed in the given order. To this purpose, we seek the innermost composite state that encloses both the source and the target state, i.e. their *least common ancestor*. Then the following actions are executed sequentially: (a) the exit actions of the source state and of any enclosing state up to, but not including, the least common ancestor, innermost first (see macro *exitState*); (b) the action on the transition; (c) the entry actions of the target state and of any enclosing state up to, but not including, the least common ancestor, outermost first (see macro *entryState*); finally (d) the "nature" of the target state is checked and the corresponding operations are performed. For a precise, mathematical definition of the parameters *ToS* and *FromS* refer to [BCR00].

Moreover, if the selected event is a *call event*, processing such event results in executing the method (if one exists) in the full descriptor of the class associated with the operation of the call event, and then the action sequence (if any) before entering the target state. This form of behavior correspond to mix transitions with explicit actions with *protocol transitions*[6] in order to get a hybrid model

[6] A *protocol state machine* for a class defines the order in which the operations of that class can be invoked. The behavior of each of these operations is defined by an associated method, rather than

(the current OMG specification doesn't prohibit this because there may be some applications that might benefit from mixing).

The sequentialization and iteration constructs defined for ASMs in [BS00] provide the combination of black box – atomic step – view and the white box – durative – view which is needed here to guarantee that when the two ASM rules defined above are executed, all the updates which occur in the macros are performed before the next event is dispatched or becomes releasable. This behavior is reflected in particular by the following definition of the parameterized macro *stateMachineExecution*[7].

```
*stateMachineExecution*(*trans*) ≡
**if** *internal*(*trans*) **then** *action*(*trans*)
**else seq**
        *exitState*(*source*(*trans*),*ToS*)
        **if** *isCallEvent*(*event*(*trans*)) **then** *method*(*event*(*trans*)) (*self*,*parList*)
        *action*(*trans*)
        *enterState*(*FromS*,*target*(*trans*))
        **case** *target*(*trans*)
            *SequentialState*: *enterInitialState*(*target*(*trans*))
            *ConcurrentState*: *startConcurrComput*(*target*(*trans*))
            *HistoryState*: *restoreConfig*(*target*(*trans*))
        **endcase**
```

It remains to define the macros appearing in the above rule for exiting and entering states, and for the additional actions for sequential, concurrent and history states (see the complete model in [BCR00, BCR, CRS]).

through action expressions on transitions. A transition in a protocol state machine has as trigger a call event that references an operation of the class, and an empty action sequence.

[7] A similar use of these structuring concepts for ASMs has been made for the analysis of Java and its implementation on the JVM in [RSB01].

UML Element	*ASM Model Element* AsmGofer Encoding
StateMachine	*StateMachine* *statemachine(name,init,state,transition,event)* *st1=statemachine(StackMachine,initS1,{empty,intermediate,full},* *{tr1,tr2,tr3,tr4,tr5,tr8,tr9}* $\cup$ *{tr6,tr7}, {push(e:Int),pop()})*

	`type StateMachine = (String,State,[State],[Transition],[Event])`
	`instance AsmTerm StateMachine`
	`st1:: StateMachine`
	`st1 = (StackMachine,initS1,states1,transitions1,events1)`
	`states1 :: [State]`
	`states1 = [topSt,empty_Stack,intermediat_Stack,full_Stack]`
	`transitions1 :: [Transition]`
	`transitions1 = [tr1,tr2,tr3,tr4,tr5,tr6,tr7,tr8,tr9]`
	`events1 :: [Event]`
	`events1 = [push_Stack,pop_Stack]`

SimpleState	*SimpleState* *state(entry,exit,do(A),defer)* *empty = state(printer.sgnStackEmpty(),_,_,_)*

	`type State = (String,Action,Action,Action,[Event],StateKind)`
	`instance AsmTerm State`
	`data StateKind = S | SS | CS | IS | FS | SHS | DHS`
	`instance AsmTerm StateKind`
	`empty_Stack :: State`
	`empty_Stack = (''empty'',entryEmpty_Stack,noAc,noAc,[],S)`

Action	*Action* *printer.sgnstackEmpty()*

	`type Action = (String,ActionKind,ActionExpression)`
	`instance AsmTerm Action`
	`data ActionKind = UA | CA | SA | CRA | DA`
	`instance AsmTerm ActionKind`
	`type ActionExpression = Dynamic(AGENT -> Parameter -> Rule())`
	`instance AsmTerm ActionExpression`
	`entryEmpty_Stack :: Action`
	`entryEmpty_Stack = ("entryEmpty",SA,expentryEmpty_Stack)`
	`expentryEmpty_Stack :: ActionExpression`
	`expentryEmpty_Stack self parlist =`
	`send (printer_Stack self) sgnstackEmpty_Stack`

Event	*Event* *sgnstackEmpty*

	`type Event = (String,EventKind,Parameter)`
	`instance AsmTerm Event`
	`type EventKind = ComplEv | SignalE | CallE`
	`instance AsmTerm EventKind`
	`sgnstackEmpty_Stack :: Event`
	`sgnstackEmpty_Stack = ("stackEmpty",SignalE,undef)`

Transition	*Transition*
	trans(source,target,event,guard,action)
	tr2 $=$ *trans(empty,intermediate,push(n),self.size* $<$ *self.limit-1,*
	append(n,self.buffer); self.size:= self.size + 1)

	`type Transition = (State,State,Event,Guard,[Action],Internal)`
	`instance AsmTerm Transition`
	`tr2 :: Transition`
	`tr2 = (empty_Stack,intermediate_Stack,push_Stack,g2,`
	`[tr2action1,tr2action2],False)`
	`g2 :: Guard`
	`g2 self = (size_Stack self) < (limit_Stack self) -1`

Table 1. A fragment of ASM and AsmGofer encoding of UML metamodel

5. Toolkit Architecture

Fig. 7 shows the architecture of the toolkit for UML models simulation.

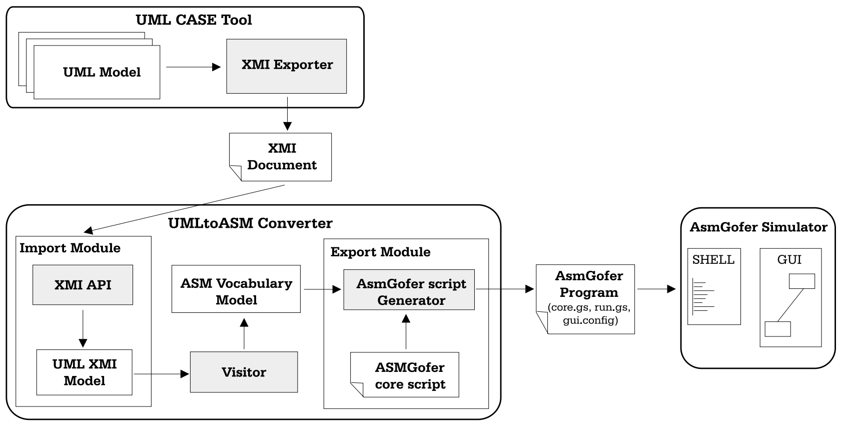


Figure 7: UML to ASM Toolkit Architecture

The main components of this framework are a UML CASE Tool component able to produce the XMI format for diagrams (e.g. Poseidon [Pos]) and AsmGofer Simulator [Sch], an advanced ASM interpreter embedded in the functional programming language Gofer which is a subset of Haskell – the de-facto standard for strongly typed lazy functional programming languages. The simulator takes in input two main scripts: (1) a *core simulator* (`core.gs`), the AsmGofer encoding of the ASM model for UML state machines (signature and rules), plus its initialization according to the input UML model; (2) a set of *simulation scenario rules* (`run.gs`).

The signature and the rules in the core simulator consist of the AsmGofer encoding of the ASM model obtained from UML structural elements and UML model behavior applying the mapping schema presented in section 4; they have been defined *once for all*. The initialization of such signature depends on the specific input model and is done *automatically* by the UMLtoASMConverter component (a Java program) according to the class diagrams and their associated state diagrams in the input UML model. The conversion consists of the following steps: (a) the component XMI API constructs a DOM (Document Object Model) tree-like representation of the XMI document; (b) the Visitor traverses the tree and initializes a Java representation of the ASM vocabulary of UML metamodel elements; (c) the AsmGofer script Generator component exports the ASM signature into AsmGofer. Observe that, should another ASM interpreter be chosen, the exporter component in (c) can be replaced by another component exporting the ASM signature into the input language of the interpreter. In our toolkit we have realized just one export module for AsmGofer tool.

Table 1 shows a fragment of the AsmGofer encoding of the ASM model of UML state machines, and the signature initialization for the *stack-printer* example.

An alternative to our architecture would be to write a plug-in for a specific UML design tool that directly produces the ASM specification. In that way, the overhead of writing the XMI file, parsing it and reconstructing the models could be avoided. However, the plug-in would be specific to the UML design tool and the tool independence that we gain from using the open XMI standard would be lost.

The script `run.gs` is fully generated from the object and sequence diagrams. It contains (a) the rule *scenery#* used to complete the signature initialization at user data level according to a selected input object diagram; (b) a rule run_{ij} for each external stimulus in the sequence diagram to be simulated, used to update the agent event queues monitored by the environment (i.e. *ExtEventQueue*). This script is executed at run-time (see section 5.1).

The file `gui.config` —also produced by the AsmGofer script Generator— is given as input to the AsmGofer guiGenerator to create the script `gui.gs` that contains all the functions and rules for the *graphical interface*.

5.1. A screenshot of simulation

In Fig. 8 we report a screenshot of simulation of the UML stack-printer model.

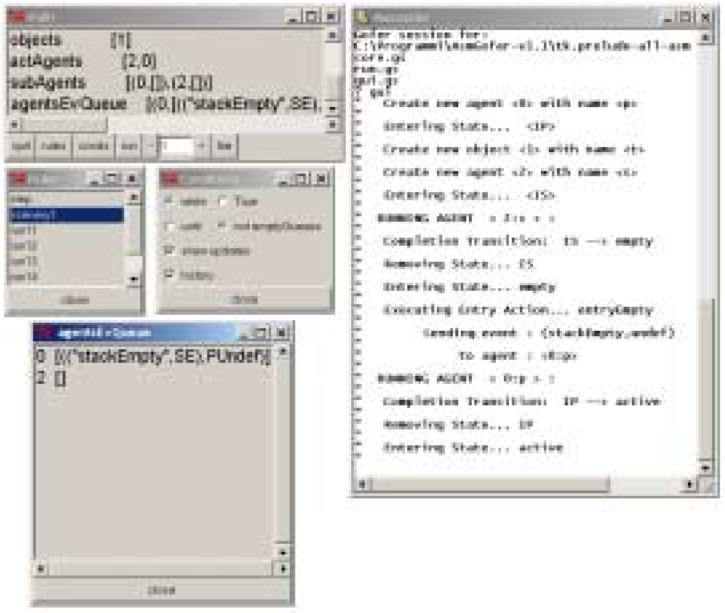
Th graphical interface contains the following windows. The `main` window displays the content of dynamic functions of the ASM model chosen by the user through the `gui.config` file. Double clicking on each dynamic function opens a new window which displays only the values for that dynamic function. Fig. 8

also shows this window for the dynamic function dispatchedQueue. Pressing the run button executes the rule selected in the rule window while or until the selected condition in the condition window (opened clicking on the conds button) holds. If the check button show updates is enabled, then the GUI will be updated in each step during iterative execution with run; otherwise only after the iterative execution. If the history button is enabled, then AsmGofer stores the complete history of execution steps. The buttons + and - move backwards and forwards in the history. Finally, the fire button executes one step of the selected rule. Clicking on the rules button displays the list box containing the simulation scenario rules scenary#, step, and $\mathtt{run}_{ij}$. A rule in the Rules window is executed by selecting it and pressing the fire button. When firing scenary# the signature is initialized (see previous section). step correponds to a single run-to-completion step of each of the active agents running concurrently. $\mathtt{run}_{ij}$ insert an external stimulus (as displayed in the sequence diagram) in the agent event queues monitored by the environment. The combination step and fire when the check button not emptyQueues in the Conditions window is selected, allows the fix-point execution of the model.

For instance, Fig. 8 displays the system configuration after having executed the rules scenery1 and a run-to-completion step. There are two active agents[8] 2 (s: Stack) and 0 (p: Printer), which do not have subAgents, and an object 1 (t: Terminal). agentsEvQueue is a list containing the event queues of all active agents. Agent s, having entered the empty state, has performed the extry action sending the signal sgnstackEmpty to p. Simultaneously, p has entered the active state, ready to consume in the next state the signal stackEmpty inserted in its event queue by s. Note that the shell window reports relevant steps made by the agents: the event processed, the transition selected, the target state, actions performed, etc..

Fig. 9 shows the shell window in the next step. Agent p processes event stackEmpty and fires the internal transition tr33 resulting in executing the print method of terminal object t. The variable windowBuffer of t is set to ''Stack is empty''.

[8] In AsmGofer agents are identified by integers.



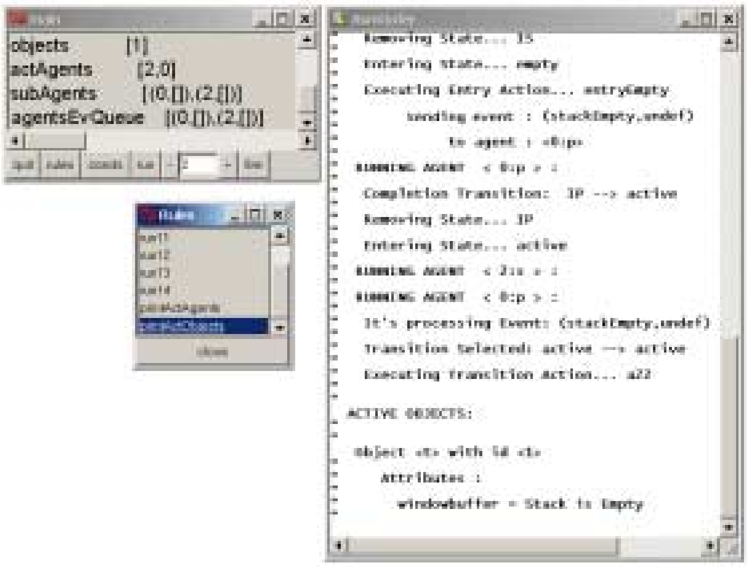
```
objects          [1]
actAgents        [2,0]
subAgents        [(0,[]),(2,[])]
agentsEvQueue    [(0,[(("stackEmpty",SE),
```

```
step
scenery1
sur11
sur12
sur13
sur14
```

```
0 [(("stackEmpty",SE),PUndef)]
2 []
```

```
Gofer session for:
C:\Programme\asmGofer-v1.1\tk.prelude-all-asm
core.gs
rum.gs
gui.gs
? gui
   Create new agent <0> with name <p>
   Entering State...  <IP>
   Create new object <1> with name <t>
   Create new agent <2> with name <s>
   Entering State...  <IS>
  RUNNING AGENT  < 2:s > :
   Completion Transition:  IS --> empty
   Removing State... IS
   Entering State... empty
   Executing Entry Action... entryEmpty
         Sending event : (stackEmpty,undef)
              to agent : <0:p>
  RUNNING AGENT  < 0:p > :
   Completion Transition:  IP --> active
   Removing State... IP
   Entering State... active
```

Figure 8: Stack-Printer: `scenery1` and `step` rules execution



```
objects          [1]
actAgents        [2,0]
subAgents        [(0,[]),(2,[])]
agentsEvQueue    [(0,[]),(2,[])]
```

```
sur11
sur12
sur13
sur14
printActAgents
printActObjects
```

```
   Removing State... IS
   Entering State... empty
   Executing Entry Action... entryEmpty
        Sending event : (stackEmpty,undef)
             to agent : <0:p>
  RUNNING AGENT  < 0:p > :
   Completion Transition:  IP --> active
   Removing State... IP
   Entering State... active
  RUNNING AGENT  < 2:s > :
  RUNNING AGENT  < 0:p > :
   It's processing Event: (stackEmpty,undef)
   Transition Selected: active --> active
   Executing transition action... a22

 ACTIVE OBJECTS:

  Object <t> with id <1>
     Attributes :
       windowbuffer = Stack is Empty
```

Figure 9: Stack-Printer: `step` rule execution

6. Related work

Recently, there is an increasing interest on simulation of UML diagrams and on automatic mapping of executable UML models to formal frameworks (formal methods supported by analysis tools). Some progresses have been made. Several research groups are developing executable UML tools based on specific action languages: **iUML** (intelligent UML) [iUM] from Kennedy Carter, based on the *ASL* language; **Kabira** Design Center [Kab] from Kabira company, which uses *Kabira AS*; **BridgePoint** Development Suite [pro] from Project Technology, with the *AL* action language. These action languages are targeted at object modeling techniques and have primitives built in to support the creation and deletion of objects and links as well as the sending of signals and the invocation of operations, but they do cover most of the key features of the Standard UML Action Semantics [UML03]. **UMLAUT** [umlc] software from IRISA is capable to transform an initial UML model (class, state-charts and deployment diagram) into an executable one – in the form of a labelled transition system (IOLTS) – linkable to validation, simulation tools and test generation tools; **Prosasim UML simulator**, an integrated component of Prosa [**?**] UML modeler, in the UML model context, allows to run the application and to follow the execution in real-time.

Several semantics for UML statecharts have been proposed in the literature. Many of these are concerned with modeling Harel's statecharts, whose semantics is rather different from UML state machines (e.g. in the event handling policy). Although our model [BCR00, BCR, CRS] can be adapted to grasp such differences, our intent was to define the UML state machine semantics "as are", up to the degree of precision one can reach without compromising the desired freedom of the so called "semantic variation points".

Differently from the formalization of UML state machines in [GPP98, Kus01, Var, RACH00, vdB01], our model [BCR00, BCR, CRS] reflects the original structure of machines as described in the UML documents, without imposing any graphical transformation or flattening of diagrams.

[GPP98] uses graph rewriting techniques to transform UML state machines into a "normal form", without considering the execution of actions and activities.

A similar approach based on graph transformation is used in [Kus01] and in [Var] where however history states, actions, and guards are not formalized. Also in the model in [RACH00], which uses an algebraic specification approach, some state machines features are left out or covered by semantical equivalences.

Schäfer et al. [SKM01] defined the semantics of a statechart by encoding it into Promela, the input language of the model checker SPIN [Hol97]. This makes it possible to formally verify statecharts in SPIN.

Latella et al. [MM99] and von der Beeck in [vdB02] exploit Extended Hierarchical Automata (EHA) as an intermediate language for formulating the UML statecharts semantics in terms of Kripke structures. However, they only considered a restricted subset of UML statecharts leaving out important features such as internal transitions, completion events, etc.

In [NB03], Butler et al. propose a formalization of UML state diagrams in terms of CSP processes. The work presented covers the formalization for initial state, final state, simple state, choice state, and composite OR-state, leaving out other important features. One deficiency using CSP, in particular, is that transition priority, when more than one transition from the same source state is enabled to fire simultaneously, cannot be modeled since CSP does not have any construct to capture it.

In [SB04, SBO03, SBO04], Butler et al. propose an integration of UML and the formal notation B. They start from defining a UML profile (in accordance with the UML extensibility mechanisms), called UML-B, and then provide an automatic translator that produces a B specification from UML models. This strategy allows to create intermediate UML-B models that can be translated into B in a style that is amenable to the proof tools. The integrated modelling notation, UML-B, inherits from both UML and B, but primarily it is a specialization of the UML modelling elements via stereotypes, added tagged values to represent B modelling features and imposed constraints to ensure that UML-B models are translated into usable B.

With respect to the ASM formal method, Ober [Obe03] has presented an ASM semantics of UML derived from metamodel and incorporating actions, but she does not define the semantics of state machines and of other behavioral diagrams.

Jürjens [JÖ2] provides a formal semantics for a part of UML that allows one to use UML subsystems to group together several diagrams. It is formulated using ASMs and makes use of the statechart semantics from [BCR00]. The author did not consider complex constructs such as inter-level transitions and history pseudostates.

Compton et al. present in [CHS00] an ASM model for UML statecharts. They make explicit state exiting and entering of a firing transition by adding a transition for every affected states. However, they did not consider compound transitions and history pseudostates.

Using an independent graph definition language GTDL [GTD] and a restricted variant of ASMs, Object Mapping Automata (OMA) [OMA], Jin et al. present in [YJJ] a generic approach to integrate a visual language in a heterogeneous modeling and simulation environment and they use UML statechart diagrams as an example for their approach. In the semantics, they state to give complete solutions for handling completion events and general inter-level transitions, solving *conflicts* between transitions. However, they do not clearly analyze the cases of conflict situations which may arise between transitions, as we did instead in [BCR].

7. Conclusion and Future Work

Simulating UML models is not our primary and exclusive goal nor do we intend to provide a new or alternative action semantics to UML. We are interested in studying the possibility of automatically encode UML executable profiles (already endowed of action semantics) into the well-established ASM formal method for software development, in order to produce precise models, verifiable and executable with the existing formal techniques. In this way, designers can asses consistency, completeness, and robustness of a system design before it is implemented. Results from model analysis can be used and, when possible, automatically processed to provide further feedback for refining and fixing models at UML design level. Furthermore, additional analysis techniques can be exploited for design refinement and code generation from formal models (see sect. 9.4 in [BS03]).

A crucial issue to realize UML-based simulation tools, automated model transformation (for model refinement, model refactoring, code generation and reverse engineering) consists in defining a complete UML formal semantics – in the complementary denotational, operational and axiomatic flavors – taking into account: 1) the abstract syntax and the static semantics given respectively by the MOF-compliant UML metamodel and the OCL rules, which need to be formally specified into the chosen formalism; 2) the dynamic semantics informally given in the official UML documentation, which needs to be formally described by means of the behavioral features provided by the chosen formal approach.

As future work, we aim to extend our validation tool in several directions. Since we have developed a framework for a subset of the UML, currently the ASM signature does not capture the whole UML metamodel. In order to add more consistency and completeness to the rules mapping the UML metamodel into the ASM signature, we intend to make our mapping completely homomorphic as suggested in [MC01]. Homomorphic mappings have the important property to preserve structural relationships between entities in two different structures. We also want to automatize the construction and the application of the mapping rules system for transforming the UML metamodel (also in XMI form) into ASM signature. Such automation guarantees error-prone translations and allows an easy update of the ASM signature subsequent to the frequent changes of the standard UML.

Finally, we want to link the simulator to other existing ASM interpreters (ASML, XASM, ASM Workbench – see sect. 8.3 in [BS03] –) and make the graphical representation of the statecharts in the graphical interface dynamic, in order to animate the statecharts and provide a better visual clue about the ongoing simulation.

References

[AG00] E. Riccobene A. Gargantini. Encoding Abstract State Machines in PVS. In *Abstract State Machines: Theory and Applications*, LNCS 1912. Springer, 2000.

[AG01] E. Riccobene A. Gargantini. ASM-based Testing: Coverage Criteria and Automatic Test Sequence Generation. *Journal of Universal Computer Science, Springer*, 7(11):1050–1067, 2001.

[AG03] S. Rinzivillo A. Gargantini, E. Riccobene. Using SPIN to Generate Tests from ASM Specifications. In *Abstract State Machines. Advances in Theory and Practice*, LNCS 2589. Springer, 2003.

[AK] M. Anlauff and P. Kutter. Xasm: The Open Source ASM Language. `http://www.xasm.org`.

[asm] ASMs Web site. `http://www.eecs.umich.edu/gasm/`.

[BCR] E. Börger, A. Cavarra, and E. Riccobene. A precise semantics of UML State Machines: making semantic variation points and ambiguities explicit. In *Proc. Semantic Foundations of Engineering Design Languages (SFEDL02) - ETAPS 2002*.

[BCR00] E. Börger, A. Cavarra, and E. Riccobene. Modeling the Dynamics of UML State Machines. In Y. Gurevich et al., editor, *Abstract State Machines. Theory and Applications*, volume 1912 of *LNCS 1912*, pages 223–241. Springer, 2000.

[BS00] E. Börger and J. Schmid. Composition and Submachine Concepts for Sequential ASMs. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic (Gurevich Festschrift)*, number 1862 in LNCS, pages 41–60, 2000.

[BS03] E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.

[CHS00] K. Compton, J. Huggins, and W. Shen. A semantic model for the state machine in the Unified Modeling Language, 2000.

[CRS] A. Cavarra, E. Riccobene, and P. Scandurra. Integrating UML Static and Dynamic Views and Formalizing the Interaction Mechanism of UML State Machines. In E. Börger et al., editor, *ASM 2003*. Springer.

[EB00] J. Schmid E. Börger, E. Riccobene. Capturing Requirements by Abstract State Machines: The Light Control Case Study. *Journal of Universal Computer Science, Springer*, 6(7):597–620, 2000.

[GPP98] Martin Gogolla and Francesco Parisi-Presicce. State diagrams in UML: A formal semantics using graph transformations. In Manfred Broy, Derek Coleman, Tom S. E. Maibaum, and Bernhard Rumpe, editors, *Proceedings PSMT'98 Workshop on Precise Semantics for Modeling Techniques*. Technische Universität München, TUM-I9803, 1998.

[GTD] J.W. Janneck. Graph Type Definition Language (GTDL) Specification. Moses project (1997) Computer Engineering and Communications Laboratory, ETH Zurich.

[Gur95] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[Hol97] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[iUM] iUML, from Kennedy Carter's website. `http://www.kc.com`.

[JÖ2] Jan Jürjens. A UML statecharts semantics with message-passing. In *Proc. of the 2002 ACM symposium on Applied computing*, pages 1009–1013. ACM Press, 2002.

[Kab] Kabira Design Center, from Kabira company. `http://www.Kabira.com`.

[Kus01] Sabine Kuske. A formal semantics of UML State Machines based on structured graph transformation. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 - The Unified Modeling Language. Modeling Laguages, Concepts and Tools. Toronto, Canada. Proceedings*, volume 2185 of *LNCS*. Springer, 2001.

[MC01] W.E. McUmber and B.H.C. Cheng. A general framework for formalizing uml with formal languages. In *Proc. of ICSE01*. IEEE Computer Society Press, 2001.

[MM99] Diego Latella I. Majzik and M.Massink. Towards a formal operational semantics of UML statechart diagrams. In *In Proc. of Formal Methods for Open Object-based Distributed Systems*. Kluwer, 1999.

[NB03] M. Y. Ng and M. Butler. Towards formalizing UML State Diagrams in CSP. In *Proceedings of 1st IEEE International Conference on Software Engineering and Formal Methods*, pages 138–147, 2003.

[Obe03] I. Ober. An ASM Semantics of UML Derived from the Meta-model and Incorporating Actions. In *E. Börger et al., ed., ASM 2003*, volume 2589 of *LNCS*. Springer, 2003.

[OMA] J.W. Janneck and P. W. Kutter. Mapping automata - simple abstract state machines. Technical Report 49, Computer Engineering and Networks Laboratory, ETH Zurich.

[oSEG] Microsoft Research Foundations of Software Engineering Group. AsmL: The Abstract State Machine Language. `http://research.microsoft.com/foundations/AsmL/`.

[Pos] Poseidon UML Tool. `http://www.gentleware.com`.

[pro] BridgePoint Development Suite, from Project Technology inc. `http://www.projtech.com`.

[RACH00] G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. Analysing uml active classes and associated state machines – a lightweight formal approach. In T. Maibaum, editor, *FASE 2000 - Fundamental Approaches to Software Engineering*, volume 1783 of *LNCS*. Springer, 2000.

[RJB99] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1999.

[RSB01] R.Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine - Definition, Verification, Validation*. Springer Verlag, 2001.

[SB04] C. Snook and M. Butler. UML-B: Formal modelling and design aided by UML. Technical Report, Electronics and Computer Science, University of Southampton, 2004.

[SBO03] C. Snook, M. Butler, and I. Oliver. Towards a UML profile for UML-B. Technical Report DSSE-TR-2003-3, Electronics and Computer Science, University of Southampton, 2003.

[SBO04] C. Snook, M. Butler, and I. Oliver. The UML-B Profile for formal systems modelling in UML. In Mermet, J., Eds. *UML-B Specification for Proven Embedded Systems Design*, chapter 5. Springer, 2004.

[Sch] J. Schmid. AsmGofer. `http://www.tydo.de/AsmGofer`.

[Sch01] J. Schmid. Compiling Abstract State Machines to C++. *Journal of Universal Computer Science, Springer*, 7(11):1068–1087, 2001.

[SKM01] Timm Schäfer, Alexander Knapp, and Stephan Merz. Model checking UML state machines and collaborations. *Electronic Notes in Theoretical Computer Science*, 55(3):13 pages, 2001.

[UMLa] OMG, The Unified Modeling Language (UML). `http://www.uml.org`.

[UMLb] OMG, UML 2.0 Superstructure Final Adopted Specification, ptc/03-08-02.

[umlc] UMLAUT. `http://www.irisa.fr/pampa/UMLAUT/`.

[UML03] OMG, The Unified Modeling Language Specification, Document ad/03-03-09, version 1.5. `http://www.omg.org/cgi-bin/doc?ad/03-03-09`, 2003.

[Var] D. Varró. A formal semantics of uml statecharts by model transition systems. In *Proc. Graph Transformation*, volume 2505 of *LNCS*, pages 378–392.

[vdB01] M. von der Beeck. Formalization of UML-Statecharts. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools. Fourth International Conference, Toronto, Canada*, volume 2185 of *LNCS*. Springer, 2001.

[vdB02] M. von der Beek. A structured operational semantics for UML-statecharts. In *Software and System Modeling 1(2)*, pages 130–141, 2002.

[YJJ] Robert Esser Yan Jin and Jrn W. Janneck. A Method for Describing the Syntax and Semantics of UML Statecharts. In *Software and Systems Modeling (SoSyM), 3(2)*.

Authors addresses:

Alessandra Cavarra, Oxford University Computing Laboratory, Wolfson Building Parks Road, - OX1 3QD - Oxford, U.K.

Elvinia Riccobene, Università di Milano, Dipartimento di Tecnologie dell'Informazione - Via Bramante, 65 - 26013 Crema (CR), Italy

Patrizia Scandurra, Università di Catania, Dipartimento di Matematica e Informatica - V.le A. Doria, 6 - 95125 Catania, Italy