An ASM Semantics of UML Derived from the Meta-model and Incorporating Actions*

Ileana Ober

VERIMAG
Centre Équation - 2, avenue de Vignate
38610 Gieres, France
ileana.ober@imag.fr

Abstract. We present an approach towards a formal dynamic semantics for UML using ASM. We aim to remain as close as possible to the standard definition of UML and to cover the operational part of the language with particular attention to the behavior description based on actions. To remain close to the standard UML, we automatically translate the UML metamodel in ASM. This allows to take into account all the concepts and relationships contained in the standard, and to minimize the changes subsequent to the frequent updates of the standard.
For the dynamic part, the particularity of our approach is that we focus on actions, as defined in our proposal to the OMG action semantics working group. We deal with concurrency, signal exchange, operation calls, general communication primitives, etc. We do not define the semantics of state machines, but we clearly define their place within the framework of our semantics. We also describe how the ASM domains and functions used in the semantics are built initially from a particular UML model.

1 Introduction

While UML is rapidly becoming the industry standard for modeling, its standard definition does not contain a precise semantics. The semantics of UML is defined, in Chapter 2 of [19], by a textual description coming with some meta-modeling description (an of abstract syntax of the language). A preliminary part of our research, not described in this paper, that can be found in [15, 17] consisted in defining a mechanism for behavior description based on actions. This work was done as a response to an official request by the OMG, in conjunction with the Action Semantics Working Group. To complete the behavior description mechanism based on actions, we formalize it in the framework of the entire UML language.

In this paper we present a UML formalization approach using Abstract State Machines (ASM) [10]. We have chosen the formalism of ASMs for its expressing power coupled with a high level of abstraction, that applied nicely to a related

*The major part of this work was done while the author was at Telelogic (www.telelogic.com). Additional support provided by the OMEGA project IST-33522 (http://www-omega.imag.fr).

formalization targeting the SDL semantics [6, 13]. Our formalization approach can be regarded both in connection with the previous work, and as a stand-alone UML formalization approach. We use as starting point for our formalization the UML meta-model which we translate automatically in ASM. This allows us to consider all the concepts and relationships existing in the standard. Also, as the language is continuously subject to (in general minor) changes, our semantics can easily adapt to new versions. On the other hand, at the moment we write this paper a major update of the language is planned. Our approach will minimize the changes needed in the semantics.

For the dynamic part we use the actions for specifying behavior. Actions cover transformational and control-oriented behavior (through data access, assignments, loops and decision constructs), as well as interactive behavior (through remote operation calls, signal passing, etc). We describe the concurrency model. In this paper we give only the basic principles of the semantics definition and some examples. For more details on the actual semantics the reader is referred to the Annex B of [17].

The rest of the paper is organized as followings: in Section 2 we argue on the need for a precise semantics for UML and we highlight some points to take into account for the semantics definition. Section 3 gives some insight into the approach we have taken. We describe here the overall approach and the structure of the semantics, we briefly present the static and the dynamic parts of the semantics (§3.1-§3.2), and we end this Section by presenting how the semantics applies to specific models (§3.3). In Section 4 we present some related work and we try to position our research in this context. We end in Section 5 by drawing some conclusions and presenting future work directions.

2 Need for precise semantics of UML.

The need for a precise semantics of UML was often stated [7, 8]. A sound semantics for UML allows to build tools that check models, simulate them, and generate code.

Currently, the UML semantics is informally defined in plain text and it is often unclear, ambiguous or it contains contradictory assertions. Tools containing compilers use various methods to solve these problems, which led to different actual UML semantics implemented by different UML tools.

Although the existence of an informal semantics does not preclude the fulfillment of these goals, the practice has proven that a formal semantics brings them closer. That is why we will use formal techniques for defining the UML semantics. Nevertheless, we do not see the definition of a formal semantics as a goal, it only represents a possible way towards increased precision and advanced tool support.

The definition of a precise semantics should take care of two essential UML characteristics: the fact that UML is intended to address the early phases of analysis and design, thus the need to handle incomplete and possibly inconsistent information, and the fact that UML is mostly seen as a family of languages [5], possibly demanding contradictory semantics. The fact that UML shall be applicable on various domains with contradictory demands for the semantics of the same entity was often an argument against the definition of a precise semantics for UML.

This formalization approach is a continuation of previous efforts on defining a precise semantics to UML actions. These efforts [17 chapter 7, 15] resulted in the definition of a framework for behavior description based on actions.

UML state machines are incomplete without a precise actions semantics, as actions describe what happens on state and transitions. On the other hand, we can completely describe behavior using actions (and no state machine). Therefore by focusing on action we can treat complete specifications.

Our semantics applies to a domain where behavior is important and it is described through actions, thus we give special attention to behavior and communication. Part of the task of this approach is to make choices, when the standard leaves place for alternatives (e.g. we will use a specific model for concurrency, a certain semantics for signal exchange, etc.). This choices are of course arguable, but unavoidable to obtain a meaningful semantics.

We use Abstract State Machines [10] for our formalization because they allow an operational description at a high abstraction level. Subsequent changes to the semantics definition will demand a reasonable amount of extra-efforts and the high level of abstraction would solves some of the problems related to over-specification.

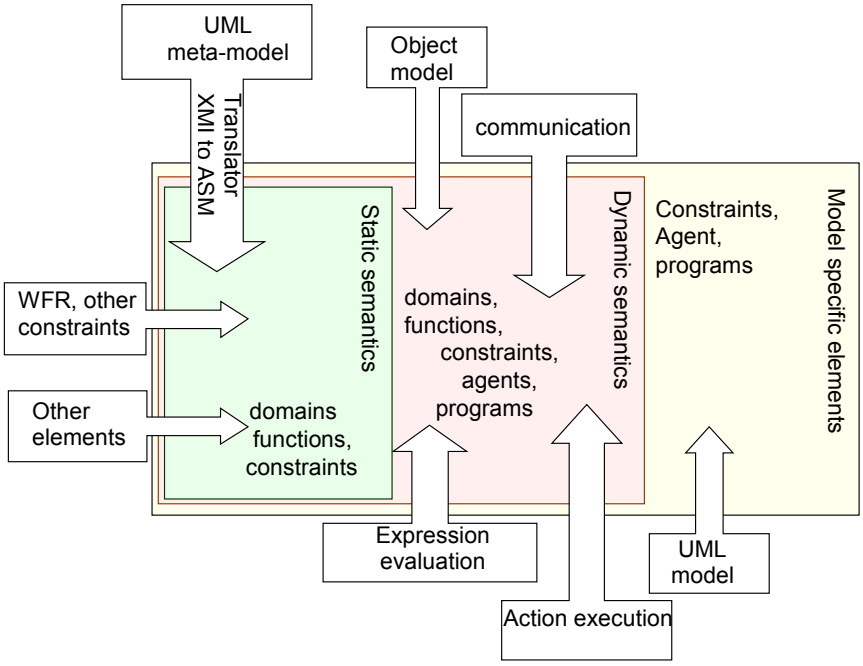

Fig. 1. Structure of the UML semantics

3 UML semantics definition

The definition of the ASM formal semantics of UML consists in associating to each UML model a particular multi-agent real-time ASM. Therefore, given an UML model, which is an instance of the UML meta-model, we have to identify its

corresponding ASM agents, functions and domain names and for each agent the ASM program that describes its behavior.

Fig. 1 illustrates the structure of our formal semantics definition. As shown in this figure, the static semantics part expresses each UML basic concept in terms of ASM names and constraints. In the UML standard [19] the static semantics is defined using the UML meta-model, constrained through some integrity constraints, expressed as well-formedness rules (WFRs) described in OCL, or in plain text.

In order to best use the existing specification, as the UML meta-model contains information expressed in a semi-formal way, we have developed an automatic ASM generator that translates the information contained in the UML meta-model into ASM. In the next Section we will discuss this tool into more detail.

Our *static semantics* definition contains constraints corresponding to the WFRs stated in the definition of UML both in OCL and in plain text. Additionally, the static semantics offers access function to various elements of the static semantics, describes the method lookup mechanisms, etc.

The *dynamic semantics* is constituted by some ASM names, constraints and rules that describe the basis of behavior. These rules formalize the semantics of signal passing, of execution threads, of the signal queue etc. The dynamic semantics does not correspond directly to anything from the UML meta-model, although the basic concepts (operation, signal, etc) are part of the meta-model and have been translated into ASM in the static semantics part.

The dynamic semantics contains functions that describe the evaluation of expressions, and operational semantics of UML actions expressed in ASM, based on an execution engine that describes the basis of behavior (message passing semantics, time, etc.). When defining the dynamic semantics we often have to choose the semantics we give to various constructs, because the UML standard does not cover all details - either deliberately, or by omission.

As one can see from **Fig. 1**, the final result of the semantics definition is an ASM program equivalent to an arbitrary UML model. This ASM program may be used to generate code that will conform the semantics, or as input for ASM simulators which may execute symbolically the ASM program corresponding to an UML model. Some ASM tools offer already part of this functionality, some others may be further developed in order to offer a better coupling between the UML model specification and the simulations performed on the equivalent ASM program. Such tools could be made transparent to the UML modeler, which does not have to be aware of the fact that his model is internally translated into ASM, by feeding back at UML level the results of the simulation and verification.

3.1. Static semantics

Currently, the static semantics of the standard UML is defined by the UML meta-model, by some informal textual descriptions and it contains constraints given as well formedness rules [19]. Our static semantics definition has to capture all of these concepts in ASM. Additionally, the static semantics should contain ASM names for access functions and a method lookup mechanism.

Automatic meta-model translation using *XMItoASM*

The first step of the UML semantics definition consists in giving the ASM names equivalent to UML concepts. The UML meta-model is a formalism for defining entities and relationships between them. In order to translate automatically the information contained in the UML meta-model into ASM, we have developed a tool that has as input the standard UML meta-model (in XMI form) and translates it into ASM. When applied on the UML meta-model, this translator, that we call *XMItoASM*, recovers names (domains and functions) and constraints from the UML meta-model classes, attributes, associations and generalizations.

Being a one-time only process, the translation of the UML meta-model could have been done by hand, but the automatic translation has the advantage that it minimizes the risk of translation errors and it allows an easy update of the ASM names and constraints in the event of further changes in the UML meta-model.

For exemplification we consider the excerpt from the UML meta-model (from the ***Core*** package) depicted in **Fig. 2**. The ASM names corresponding to the classes described in this example are contained at lines 1-9 in **Fig. 3.**

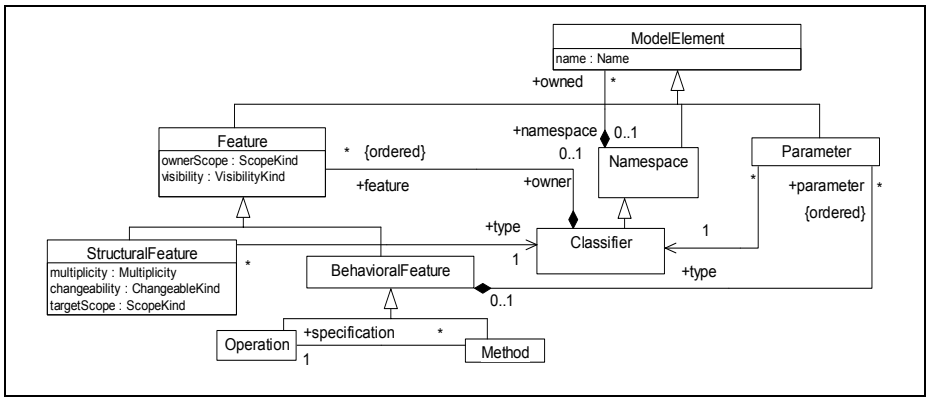

Fig. 2. UML meta-model extract

```
1   static domain UmlModelElement
2   static domain UmlFeature
3   static domain UmlNamespace
4   static domain UmlParameter
5   static domain UmlClassifier
6   static domain UmlStructuralFeature
7   static domain UmlBehavioralFeature
8   static domain UmlOperation
9   static domain UmlMethod
10  constraint UmlNamespace          ⊆ UmlModelElement
11  constraint UmlFeature            ⊆ UmlModelElement
12  constraint UmlParameter          ⊆ UmlModelElement
13  constraint UmlBehavioralFeature      ⊆ UmlFeature
14  constraint UmlStructuralFeature      ⊆ UmlFeature
15  constraint UmlOperation          ⊆ UmlBehavioralFeature
```

```
16  constraint UmlMethod ⊆ UmlBehavioralFeature
17  constraint UmlClassifier ⊆ UmlNamespace
18  static ModelElement_name :UmlModelElement→ UmlName
19  static Feature_ownerScope :UmlFeature → UmlScopeKind
20  static Feature_visibility :UmlFeature → UmlVisibilityKind
21  static StructuralFeature_multiplicity :UmlStructuralFeature
    → UmlMultiplicity
22  static StructuralFeature_changeability:UmlStructuralFeature→
    UmlChangeableKind
23  static StructuralFeature_targetScope :UmlStructuralFeature
    → UmlScopeKind
24  ModelElement_namespace :UmlModelElement→ UmlNamespace-set
25  Namespace_owned :UmlNamespace → UmlModelElement-set
26  constraint ∀ns∈UmlNamespace:
       |ns. ModelElement_namespace| = n, n∈NAT, n ≤1
27  constraint ∀ns∈ UmlNamespace-set, ∀ ms∈ UmlModelElement-set:
     (∀n∈ns: n.Namespace_owned =ms) ⇔ (∀m∈ms:
    m.ModelElement_namespace =ns)
28  Classifier_feature :UmlClassifier → UmlFeature-sequence
29  Feature_owner:UmlFeature → UmlClassifier
30  constraint ∀c ∈UmlClassifier, ∀ fs∈ UmlFeature-sequence:
    c.Classifier_feature =fs ⇔ (f.Feature_owner = c, ∀f∈fs)
31  StructuralFeature_type:UmlStructuralFeature → UmlClassifier
32  BehavioralFeature_parameter:UmlBehavioralFeature →
    UmlParameter-sequence
33  Parameter_behavioralFeature:UmlParameter → UmlBehavioralFeature
34  constraint ∀b ∈UmlBehavioralFeature, ∀ ps∈ UmlParameter-
    sequence:
              b.BehavioralFeature_parameter = ps
        ⇔     (p.Parameter_behavioralFeature = b, ∀p∈ps)
35  Parameter_type:UmlParameter → UmlClassifier
36  Operation_method: UmlOperation → UmlMethod-set
37  Method_specification: UmlMethod → UmlOperation
38  constraint ∀ ms ∈UmlMethod-set, ∀ o∈ UmlOperation:
        o.Operation_method = ms ⇔ (∀m∈ms:
    m.Method_specification = o)
```

Fig. 3. ASM domains corresponding to UML meta-classes

The fact that a meta-class *inherits* another is expressed in ASM through an inclusion constraint between the domain corresponding to the parent class and the domain corresponding to the child class. Remark that here we are talking about the inheritance at *meta-model level*. While it would be of course extremely reductive to see inheritance at UML model level as simple domain inclusion, this is sufficient at the meta-model level, where there are no operation definitions and the inheritance is added to make obvious semantic relationship between classes and to factorize

specification. The ASM constraints (domain inclusions) corresponding to the generalization relationships from **Fig. 2**. are shown at lines 10-17 in Fig.3.

To each attribute and association navigable direction in the UML meta-model we associate an ASM function. For attributes, the function is defined on the ASM domain corresponding to the meta-class that owns the attribute, and with the codomain the ASM domain corresponding to the type of the attribute. Examples of functions corresponding to attributes are given at lines 18-23 **Fig. 3**. For associations, the functions are obtained similarly, as in the case of the functions at lines 24-36.

If the target association end has the multiplicity 1, the codomain of the function corresponding to that navigable direction is the ASM domain corresponding to the class connected to the target association end, as it is for instance the case with the functions at lines 29, 31, 35 in **Fig. 3**.

If the association end does not have the multiplicity 1 and if the target association end is ordered, then the codomain of the ASM function corresponding to that association direction is a *sequence* (ordered set) of the ASM domains of the target class, as in the functions at lines 28, and 32 in **Fig. 3.** Otherwise, the codomain is a new domain: a finite (unordered) set of the ASM domains corresponding to the target association end (ex functions at lines 24, 25, 27, 36 in **Fig. 3**.

In the case of bi-directional associations we add integrity constraints to capture the fact that the functions corresponding to each navigable directions are symmetrical. (lines 27, 30, 34, 38 in **Fig. 3**.)

Our formalization does not cover the entire UML. This means that not all the ASM names and functions obtained from the UML meta-model will be used subsequently. The ASM names and constraints not needed may be removed from the semantics or may be simply ignored. We choose to keep them in the semantics; this ensures that the information contained in ASM matches accurately the information contained in the UML meta-model, and makes further enhancements easier. Annex B of [17] contains most of the ASM code automatically obtained from the UML meta-model.

Additional constraints

Besides the ASM constraints obtained after the translation of the UML meta-model, the static semantics of UML contains constraints corresponding to the well-formedness rules (WFR) existing in *UML Semantics* [19]. In the definition of UML, these constraints are expressed in OCL and we translate them, by hand, into ASM. We have managed to translate all the constraints written in OCL and some of the WFR that could not have been expressed in OCL (e.g. 2nd WFR of the SynchState).

Additional functions

Next to the functions obtained from the UML meta-model, we define a set of functions that facilitate the description of the dynamic semantics: to access all the parents of a GeneralizableElement, to get all the attributes owned by a Classifier and having as *ownerScope* the classifier, to check if two operations have the same signature, etc.

Method lookup

UML distinguishes between operations - signatures of the offered services - and methods –implementations for the operations. Several methods may correspond to the

same operation. However, given a class and an operation defined in that class, there is only one *main* method that actually describes the body of the operation, the other methods being overwritten, as an operation may be defined in the ancestors of a class and may have a local method overriding its body.

We use a dynamic method lookup mechanism. Given an operation and a class, the method lookup mechanism gives the actual method that describes the body of the operation in the context of the considered class.

We use the method lookup mechanism existing in C++. This means that operations are identified by their signature (same as defined by UML, different of the method lookup used in e.g. Eiffel or SDL), and an operation is overwritten in an inheritor if it contains a local method with same signature. The method that implements a given operation is either a local method, or the method implementing the operation defined in the closest ancestor, that we find by traversing the inheritance relations captured from the meta-model. The whole lookup mechanism is embedded in a single function: *GetMethodForClassOperation*, this minimizes the impact of modifying the lookup mechanism. We disallow statically the conflicts generated by repeated inheritance.

```
GetMethodForClassOperation: UMLCLASS × UMLOPERATION → UMLMETHOD
GetMethodForClassOperation (c: UMLCLASS , op: UMLOPERATION ) = def
if ∃o∈c.Classifier_feature ∩ UMLOPERATION :HasTheSameSignature(o,op)
then   let o = take{o ∈UMLOPERATION|
              o∈ c.Classifier_feature ∧ HasTheSameSignature(o, op)}
       in o.Operation_method.main endlet
elseif ∃ directP ∈ UMLCLASS :(p ∈ c.GeneralizableElement_parent ∧
       ∃o1 ∈ p1.Classifier_feature ∩ UMLOPERATION :
HasTheSameSignature(o1, op))
       then   let <directP, op1>=take{<p,o> ∈ UMLCLASS × UMLOPERATION|
       p ∈ c.GeneralizableElement_parent
              ∧      o1 ∈ p1.Classifier_feature
              ∧      HasTheSameSignature(o1, op)}
              in o1.Operation_method.main              endlet
elseif ∃ p ∈ UMLCLASS :(p ∈ c.GeneralizableElement_allParents ∧
              GetMethodClassForOperation(p, op) ≠ undefined)
       then let <p, m> = take{<p,m> ∈ UMLCLASS × UMLMETHOD|
              p ∈ c.GeneralizableElement_allParents
       ∧      m = GetMethodClassForOperation(p, op)
       ∧      m ≠ undefined }in m  endlet
else  undefined
endif
```

Fig. 4. Function describing the method lookup

3.2. UML dynamic semantics structure

In this section we present the entities that concern run-time information and behavior. We complete the definition of the ASM names, started in the static semantics part, and we discuss how the ASM state changes at run-time. The dynamic structure is an important component of the UML dynamic semantics, it contains the ASM domains and functions related to run-time.

The dynamic semantics defines the set of ASM agents and gives their corresponding transition rules. At this point, we focus on the behavior described by actions. We do not treat UML state machines, *i.e.* the dynamic semantics does not contain rules and agents corresponding to state machine, but we clearly identify the place of the state machines in the framework of the semantics definition. We formalize the mechanisms for signal passing, operation call, object creation and deletion, and the ASM program rules that corresponds to UML action executions.

Communication mechanisms

Run-time UML objects interact by exchanging signals and operation calls. In UML the communication is point-to-point, this means that an object can send a signal to another object if it knows its identity. The communication semantics that we use for our formal definition, is based on the following principles: the communication is done by direct addressing, the signal and operation call passing may take time, the order of events is not necessarily preserved during the transport, though events are not lost. Active objects have incoming event queues, whereas passive objects do not.

The ASM communication modeling that we use is inspired by the communication modeling used in the SDL formal semantics [13, 6]. The communication is based on events passing. We add new properties to UML events: the *sender* (which is *undefined* for spontaneous events), the target, and the arrival time (a shared function that captures also the communication time).

In order to model the communication, we attach to each active object a gate (a concept needed for the formalization, not corresponding to anything in UML), which represents a unidirectional active object entry point, that exists at run-time. After an event is produced, it is placed in its target's gate, but it is made available to the target (it is part of the object's queue) only when the system time overpasses the event's arrival time.

This definition of queues and schedules does not constrain in anyway the queue policy. Any criterion could be applied for ordering the events in the *schedule* and in the *queue*, if needed ,some additional signal properties (ex. priority) could be defined.

Concurrency model

In this section, we describe the basic principles of concurrency, as defined in our semantics. The concurrency handles aspects such as the relationship between threads and objects, characteristics of active and passive objects, relationship between operation call and state machine execution (even if we do not detail the state machine execution itself), etc.

The concurrency model that we describe subsequently is inspired by the UML concurrency model we have described in [14]. However, here we focus only on standard UML concepts, we do not add any non-standard concurrency concepts and we do not focus on state machine execution.

In our concurrency model, only active objects have thread(s) of control, while passive objects execute exclusively on the threads active objects. The consequences of these: passive objects do not have input events queue, asynchronous communication with passive objects is not possible, passive objects may not have state machines, the only behavior mechanism existing in passive objects is contained

in its operations (they can at most have protocol state machines, that only constrain their execution and do not describe it), when an active object calls an operation of a passive object, this operation executes on the callers thread.

In ASM, we model each execution thread using an ASM agent.

Active Objects

Each active object has an event gate and a main execution thread, modeled as ASM functions. The main thread manages the incoming events, ensures the dispatch of the events in the input queue and, depending on the nature of the dispatched events, relay them to the state machine or starts new operation executions, on stand-alone threads. If the active object has a state machine, then at least one of the threads owned by the object corresponds to the state machine execution[1].

A call directed towards an active object results in the creation of a new execution thread. The moment of its creation depends on the *concurrency* operation property (defined in [19] §2.5.), which specifies what happens if concurrent calls of the same operation exist. This attribute is mainly designed to protect passive objects against concurrent calls, but UML allows operations of active objects to use this property for specifying the semantics of concurrent calls. We consider that this attribute is useful for active objects, and we take it into account for the semantics of operation calls.

For exemplification, we take the trickiest case, which is when the call is directed towards a *guarded* operation (multiple calls are allowed, but the *call target* shall make sure that only one call is executed at a time, the others being blocked). An active object receiving a call to a *guarded* operation, first checks whether there are any other operations executions for the same operation running. If so, the operation call passes to a waiting status, until it can be executed. When no other operation execution exists for the called operation, a new thread is created on which the operation is executed. The ASM rule that describes these is given in **Fig. 5**. Not all the names used in this rule have been defined in this paper, nevertheless we tried to choose them as suggestive as possible. For their precise definition that reader is referred to [17].

Some basic rules that govern the behavior of active objects:

If the input queue of an active object is not empty, the active object successively treats all events contained in it, resulting in operation initiation, result return of remote calls towards the current object, or event forward to the state machine.

Since each call of an active object operation is treated on a different thread, recursive operation calls of are treated as regular calls, i.e. on several threads.[2]

The active object's state machine may receive call events, but only if they do not correspond to calls directed towards the current active object. This is to avoid the situations when the target object could ignore a call. When the main thread of the active object is available (no new events exist in the input queue), the active object may launch the calls towards guarded operations whose execution was blocked due to the *guarded* operation constraint, provided that they guard condition became valid.

---

[1] Concurrent sub-states of the state machine could be implemented on different threads, however this is a state machine semantics issue.

[2] One of the static constraints that we added, forbids recursive calls for *sequential* operations. Note however that this constraint cannot always be checked statically, thus our model cannot guarantee the absence of deadlocks, which we consider modeling errors.

```
TreatGuardedOperation(a:UmlActiveObject, op: UmlOperation, ev:UmlCallEvent) ≡
 if |Object_operationSlot(a, op).OperationSlot_execution|=0
  then
  if |Object_operationSlot(a, op).OperationSlot_waitingCalls|=0
  then
    LounchOperationThread(ev)
  else
  let callToLaunch = Object_operationSlot(a, op).
      OperationSlot_waitingCalls.head in
      LounchOperationThread(callToLaunch)
      Object_operationSlot(a, op).OperationSlot_waitingCalls :=
      (Object_operationSlot(a, op). OperationSlot_waitingCalls\
              <callToLaunch>)⌢ <ev>    endlet
  endif
 else
  Object_operationSlot(a, op).OperationSlot_waitingCalls :=
  Object_operationSlot(a, op).OperationSlot_waitingCalls ⌢ <ev>
 Endif
```

Fig. 5. ASM Rule describing a call towards a guarded operation in an active object

Passive objects

Passive objects do not own execution threads; they may not have state machines and they do not have an input queue. There is however an implicit state associated to each passive object, this state given by the value of each attribute. The operations of a passive object execute on (one of) their caller's thread. A cascade of calls of passive class operations are all executed on the caller thread that initiated this call chain..

ASM Agents

During the execution we only have ASM agents corresponding to active objects. Each active object has a main thread that manages the event receipt, several threads corresponding to the ongoing operation execution and a (set of) thread(s) corresponding to the active object state machine, that we don't detail here). The agent corresponding to an operation executes the rule that corresponds to the main method of the operation. If this method is described using actions, then the ASM program rule of the agent that executes the operation is obtained from the action specification. If the operation's method is described using other means (state machine or other), then the specification of the behavior is outside the scope of our semantics definition, and we will consider that the agent executes a void program rule.

For each particular kind of action we define an ASM macro that gives its run-time semantics. In addition, we have a special rule that describes the action execution initialization and is executed right after an action starts to execute. It initializes all the functions related to the specific action.

Capturing model specific information through initialization

Until now, we have presented the ASM elements that describe the static an dynamic semantics of UML. These elements are shared by all ASMs corresponding to UML

models[3]. We discuss here on the place of the UML model specific part in the semantics framework (see **Fig. 1.**).

The model specific part consists in specifying the content of each ASM domain at ASM start-time, using model specific information (e.g. populate the domain of classes, associations, attributes, action specifications, etc). The domains that correspond to run-time entities will be all empty at initialization time.

The initialization of names corresponding to dynamic behavior is done using *initially* integrity constraints. However, the system behavior initialization is a dynamic process and integrity constraints are not enough to capture its characteristics.

UML offers no standard mechanism for describing the system initialization. Various techniques could be imagined for capturing this information in UML (object diagrams, particular collaboration diagrams, deployment diagrams, etc.). However, as none of them is generally accepted, we are aware that the specification of the system initialization remains arguable. We assume that the UML model has a special object diagram, we call it *Initial* that gives the snapshot of the system at start time. This object diagram contains the objects and links that shall exist initially and the first operation calls.

Each ASM corresponding to an UML model, starts with an initialization phase. During this phase, a single ASM agent exists, we call it *Creator*, and this agent performs the following actions (in this order):

- it creates the objects described in the *Initial* object diagram. We do not add these objects directly to the ASM because the object creation may be followed by constructor operation call. Due to their definition, the agents corresponding to active objects are automatically created;
- it creates the links that were not created as a result of the object creation;
- it synchronizes the values of ASM names with the values of the fields in the object diagram;
- it creates call events corresponding to each call specified in the object diagram. These calls events will have the sender field unspecified;
- it dies.

When the *Creator* finishes execution, the system is in the status specified by the object diagram, and the call events corresponding to the initial calls are initiated. After the death of the creator the ASM behavior follows the rules described in the previous sections. The ASMs corresponding to UML models are assumed to live forever. However, as soon as no more active objects exist (*i.e.* no running agent), the execution of the ASM may be considered completed.

4 Related efforts

A lot of effort was devoted to the goal of having a precisely defined semantics for UML. First, all UML tools that offer more than editing capabilities embed precisely defined semantics, *without making it explicit*. Code generators give semantics to

[3] some optimizations may be imagined to eliminate the ASM names and rules irrelevant in the concrete case of the ASM corresponding to a precise UML model

classes, attributes, operations, etc, in terms of the target language. The semantics embedded in the code generators is incomplete, as in general only a part of the UML specification is translated to code. Symbolic executors (simulators) generally have a more complete semantics, which is embedded in the implementation of the simulator or it is expressed in the simulator language. In the case of UML tools, the semantics is not explicitly defined, and, in general, it is hidden in the tool implementation.

The academic tools using UML specifications, provide sounder semantics foundation, in general by focusing on particular aspects, e.g. USE [21] focuses on OCL constraints/ invariants, UMLAUT [12] focuses on formal validation based on protocol validation.

Other efforts to define precise semantics, also issued from the pragmatic need to fill the gap between advanced modeling concepts (such as state machines) and real implementation, are represented by the attempt to transform state machines into executable specifications. Notable results are described in [11].

The first attempts of formalizing UML date back in 1997 [1], when authors argued the need for a UML formal foundation based on a mathematical system model. This work is followed by more elaborated formalization efforts. The most notable of them are those done by the pUML group [7, 8], oriented towards formalizing UML in Z.

[9] formalizes UML state machines using graph transformation systems.

Other formalization attempts use ad-hoc semantics background, such as the collaboration semantics presented in [20], and of the action semantics base semantics [18].

Some UML formalization efforts using ASM already exist: [3] formalizes UML activity diagrams, by adapting ASM to natively support activity diagrams; [2] formalizes UML state machines by extending the basic ASM, with some new constructs to cover UML state machines specific features; [4] defines an ASM based toolset, focused on the state diagrams. These approaches disconnect state machines /activity diagrams of the rest of the language, and they do not consider all UML concepts (associations, inheritance, etc.).

[16] presents preliminary efforts of our study with focus on the static part.

5 Conclusions and future directions

We have used ASM to formalize the semantics of parts of UML that offer relevant information for the behavior description. Namely, we have described the semantics of classifier, class, data, expression, association, inheritance, operation, attribute, action description, and their run-time counterparts. The only operational behavior mechanism that we do not formalize is the state machine.

Our formalization approach consists in giving a precise method to obtain the ASM that corresponds to each UML model.

All the ASMs corresponding to UML models have a common part composed of: names (domains and functions) and constraints obtained from the UML meta-model, constraints corresponding to the WFRs from the UML specification and some additional integrity constraints; names (domains and functions), constraints and transition rules from the object model, action and operation run-time behavior,

definitions of the set of agents, and functions that give their behavior, from the object model.

This is the part that actually gives the semantics of UML. Each UML model can be fed into this semantics framework, to obtain the complete semantics description of each model. From a UML model we initialize the ASM by adding: initially constraints for the content of ASM names corresponding to meta-model entities obtained from the UML model. From the initial list of objects existing at initializations, obtained from the model's object diagram that gives the initialization status of the object. The same object diagram servers as basis for the definition of a *creator* agent that completes the initializations and creates the initial call events that start the system execution.

The research that resulted in the definition of UML semantics has various benefits: it fulfilled our initial goal, which was to define a precise semantics for actions, it proves that the actions defined in the previous section can be integrated not only into the UML meta-model, but more generally in the UML behavior framework, it unambiguously defines the semantics of UML constructs, it better underlines the relationships between various UML entities, it uncovers some UML inconsistencies and missing; it provides a complete UML object model, it uncovers issues to be treated when building UML-based tools, even outside the scope of this semantics, it proves that the goal of having precise definitions of UML ending into symbolic executions is realistic.

We see several directions on which this work could be continued in the future. First, we plan to integrate our semantics with an ASM tool. For the moment we have only managed to feed parts of our semantics and simulate very simple models. We intend to go forward on this direction. Another natural extension consists in adding the state machine semantics.

Another natural continuation of this work consists in taking into account more UML diagrams: check the run-time compliance of the behavior with UML sequence charts.

Currently a major revision of UML is ongoing, several proposals being submitted [22]. We plan to adapt our semantics to the new standard, as soon as this will be made available. For the static part this will be done almost automatically. For the dynamic part, we will have to review our semantics and check for its consistency with the new standard.

References

1. R. Breu, et al.: Towards a formalization of the unified modeling language. In S. Matsuoka, M. Aksit (eds), ECOOP'97, In ECOOP'97 Proceedings, LNCS 1241, (1997)
2. E. Börger, A. Cavarra, E. Riccobene. Modeling the Dynamics of UML State Machines, Int. Workshop on Abstract State Machines ASM'2000, LNCS 1912, (2000) 223-241,
3. E. Börger, A. Cavarra, E. Riccobene. An ASM Semantics for UML Activity Diagrams. In: T.Rust (Ed.), Proc. AMAST 2000, LNCS, Vol.1816, (2000) pp.292-308
4. K. Compton, Y. Gurevich, J. Huggins, W. Shen.: An Automatic Verification Tool for UML, Technique Report CSE-TR-423-00, Dept. of EECS, The University of Michigan, May, (2000)

5. S. Cook. The UML Family: Profiles, Prefaces and Packages. Proceedings of UML 2000 – The Unified Modeling Language. Advancing the Standard, LNCS 1939, (2000) 255-264
6. R. Eschbach, U. Glässer, R. Gotzhein, M. von Lowis, A. Prinz: Formal definition of SDL-2000 – Compiling and Ronning SDL specifications as ASM Models. In E Börger, U. Glässer (Eds), Journal of Universal Computer Science, vol. 7 no 11(2001), 1024-1049
7. Evans, R. France, K. Lano, B. Rumpe.: The UML as a Formal Modeling Notation. Proceedings of UML'98 – Beyond the Notation, LNCS 1618, (1999) 336-348
8. Evans, S. Kent.: Core Meta-modelling semantics of UML: the pUML approach, Proceedings of UML'99 – Beyond the Standard, LNCS 1793, (1999) 141-155
9. M. Gogolla, F. Parisi-Presicce.: State diagrams in UML: A formal semantics using graph transformation. In M. Broy, D. Coleman, T. Maibaum, B. Rumpe (Eds.), Proceedings PSMT'98. Technische Universitaet Muenchen, TUM-I9803, (1998)
10. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, Specification and Validation Methods, pages 9-36. Oxford University Press, (1995).
11. D. Harel, E. Gery. Executable Object Modelling with Statecharts, IEEE Computer 30(7):(1997) 31-42
12. W. M. Ho, J. M. Jézéquel, A. Le Guennec, F. Pennaneac'h : UMLAUT: an extendible UML transformation framework. In *Proc. ASE'99, Florida*, October 1999
13. International Telecommunication Union – Telecommunication Standardization Sector, Recommendation Z.100 Annex F - Specification and description language Electronic Bookshop, Geneva, (2000)
14. I. Ober, I. Stan. On the Concurrent Object Model of UML, EuroPar'99, Toulouse, LNCS 1685, (1999), 1377-1384
15. I. Ober, B. Coulette, M. Gandriau. Action Language for UML. LMO 2000, Mont Saint Hilaire, Canada, Hermes Science Publications, (2000) pp. 277-291
16. I. Ober. More meaningful UML Models. Proceedings of TOOLS - 37 Pacific 2000, IEEE Computer Society Press, (2000), 146-157
17. I. Ober Harmonizing Design Languages with Object-Oriented Extensions and an Executable Semantics. PhD thesis at Institut National Polytechnique de Toulouse, ENSEEIHT, Available at http://www-verimag.imag.fr/~iober/Thesis/thesis.html, April (2001)
18. Object Management Group. UML Action Semantics Final Submission, OMG Document ad/01-03-01, 24 March, (2001)
19. Object Management Group Unified Modelling Language Specification (UML), v 1.4, September (2001)
20. G. Övergaard. A Formal Approach to Collaborations in the Unified Modeling Language, Proceedings of UML'99 – Beyond the Standard, LNCS 1793, (1999) 99-115
21. M. Richters, M. Gogolla: Validating UML models and OCL constraints. Proceedings of UML 2000 - Advancing the Standard, LNCS 1939, (2000), 265-277
22. U2 Parteners Consortium. Revised UML 2.0 proposal for public review and comment, v 2.0 beta R2, June 2002