Weaving executability into UML class models at PIM level

Elvinia Riccobene
Università degli Studi di Milano
DTI, via Bramante, 65 - Crema (CR), Italy
elvinia.riccobene@unimi.it

Patrizia Scandurra
Università degli Studi di Bergamo
DIIMM, via Marconi, 5 - Dalmine (BG), Italy
patrizia.scandurra@unibg.it

ABSTRACT

Modeling languages that aim to capture PIM level behavior are still a challenge. We propose a high level behavioral formalism based on the Abstract State Machines (ASMs) for the specification and validation of software systems at PIM level. An ASM-based extension of the UML and its Action Semantics is here presented for the construction of executable class models at PIM level and also a model weaving process which makes the execution of such models possible. Our approach is illustrated using an Invoice Order System taken from the literature.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design—*concepts, notations, representation*; D.2.1 [**Software Engineering**]: Methodologies—*model-driven approach*

Keywords

Model-driven Engineering, Behavioral Modeling, Platform-Independent Modeling, UML action language, Abstract State Machines, Model weaving

1. INTRODUCTION

Model-driven Engineering (MDE)[3] promotes *models* as first class artifacts of the software development process and automatic *model transformations* to drive the overall design flow from requirements elicitation till final implementations toward specific platforms. Model Driven Architecture (MDA) [25], which supports various standards including the UML (Unified Modeling Language) [37], from the OMG (Object Management Group) is the best known MDE initiative.

In the MDA context, notations usually based on the UML are used as system modeling languages for producing *platform-independent models* (PIMs) and *platform-specific models* (PSMs). Automatic model transformations allow transforming PIMs into PSMs.

BM-MDA '09 June 23, 2009, Enschede, the Netherlands

Although MDE frameworks (OMG/MOF, Eclipse/Ecore, GME/MetaGME, AMMA/ KM3, XMF-Mosaic/Xcore, etc.) are currently able to cope with most syntactic and transformation definition issues, *model executability* is still remarked as a challenge [27], especially at PIM level. One of the main obstacles is the lack of adequate models for the behavior of the software and of mechanisms to integrate behavioral models with structural models and with other behavioral models. Although there are many different approaches for modeling behavior (see related work in Sect. 2), none of them enjoys the same universality as the UML class diagrams do for the structural parts of the software. Further evidence of confusion about PIM level behavioral modeling is the lack of agreement on what basic behavioral abstractions are required, and how these behavioral abstractions should be used. However, PIM executability is considered a remarkable feature for the system development process, since it allows verifying high-level models against the requirements goals (possibly using automated analysis tools), and it can be exploited to provide conformance for implementations at PSM and code level by generating test-cases.

A current crucial issue in the MDA context is, therefore, that of providing effective specification and validation frameworks able to express the meaning or semantics of each modeling element and interaction occurring among objects, rather than dealing with behavioral issues depending on the target implementation platform. We believe this goal can be achieved by integrating MDE/MDA structural modeling notations with a behavioral formalism having the following features: (i) it should be abstract and formal to rigorously define model behavior at different levels of abstraction, but without formal overkill; (ii) it should be able to capture heterogenius models of computation (MoC) in order to smoothly integrate different behavioral models; (iii) it should be executable to support model validation; (iv) it should be endowed with a model refinement mechanism leading to correct-by-construction system artifacts; (v) it should be supported by a set of tools for model simulation, testing, and verification; (vi) it should be endowed with a metamodel-based definition in order to exploit MDE techniques of automatic model transformations.

In this paper, we address the issue of providing executability to PIMs by using the ASM (Abstract State Machine) [6] formal notation that owns all the characteristics of preciseness, abstraction, refinement, executability, metamodel-based definition, that we identified above as the desirable properties for this goal.

We propose an ASM-based extension of the UML and its

Action Semantics to define a high level behavioral formalism for the construction of executable PIMs. This is achieved by *weaving* behavioral aspects expressed in terms of ASM elements into the UML metamodel. The ASM formal notation becomes, therefore, an abstract action language for UML at PIM level, and, by automatic models mapping, we are able to associate an ASM executable model to a UML model. In particular, we apply our technique to the UML class diagrams since they are considered to be the standard for modeling the structural parts of software. The approach is anyway applicable to any other part of the UML metamodel and to any modeling language whose abstract syntax is given in terms of a metamodel.

This paper is organized as follows. Sect. 2 provides a description of related work along the lines of our motivation. Some background concerning the ASMs is given in Sect. 3. Sect. 4 presents the proposed weaving approach between the UML and the ASM metamodels in order to provide a high level formalism to specify behavior at PIM level. In Sect. 5 we define how to automatically map UML class models into executable ASM models and the action semantics provided to the UML class diagrams by the ASM elements. Implementation details about the automatic model transformation are given in Sect. 6. Sect. 7 presents the application of our UML/ASM-based modeling notation to the Invoice Order System case study. Finally, Sect. 8 concludes the paper and outlines some future directions of our work.

2. RELATED WORK AND MOTIVATION

There are different approaches for modeling and executing behavior in the UML at PIM level. They may mainly fall into the following categories.

(I) *Not include behavior in the PIM at all*, but instead add it as code to structural code skeletons later in the MDA process. This, however, prevent us from making significant early validation of the system.

(II) *Provide preliminary executability at meta-language level.* Some recent works (like Kermeta [29], xOCL (eXecutable OCL) [38], or also the approach in [35], to name a few), have addressed the problem of providing executability into current metamodelling frameworks like Eclipse/Ecore [13], GME/MetaGME [19], XMF-Mosaic/Xcore [38], etc. This approach is merely aimed at specifying the semantics of a modeling language (another key current issue for model-based engineering) and thereby at providing techniques for semantics specification natively with metamodels.

(III) *Use the OCL [31]* (and its various extensions, see [9] for example) to add behavioral information (such as pre- and post-conditions) to other, more structural, UML modeling elements; however, being side-effect free, the OCL does not allow the change of a model state, though it allows describing it.

(IV) *Joint use of an action language*, based on the UML action semantics (AS) [37, 14], *and of UML behavioral diagrams* such as state machines, activity diagrams, sequence diagrams, etc., possibly strengthening their semantics. Behavioral diagrams can be used to capture complete behavioral information as part of the PIM. The UML AS use a minimal set of executable primitives (create/delete object, slot update, conditional operators, loops, local variables declarations, call expressions, etc.) to define the behavior of metamodels by attaching behavior to classes operations (the specification of the body counterpart is usually described in text using a surface action language). Probably the most well-known example is the approach known as "Executable UML (xUML)" [32]). Recently, a beta version [14] has been released of an executable subset of the standard UML (the *Foundational UML Subset*) to be used to define the semantics of modeling languages such as the standard UML or its subsets and extensions, and therefore providing a foundation for the definition of a UML virtual machine capable of executing UML models.

(V) *Transform UML diagrams into formal models*; e.g. transform class and sequence diagrams into graphs [11] by using graph-transformation rules in order to create a set of graphs that represent the state-space of the behavior, and then apply model checking techniques on this state-space to verify certain properties of the UML models. Similar approaches based on this *translational* technique are UML-B [36] using the Event-B formal method, those adopting Object-Z like [26, 28], etc.

The approach proposed in this paper is slightly different from the above ones. The objective is to provide a behavioral formalism at PIM level that does not depend on a particular UML behavioral diagram, since it should be general enough for other metamodel-based languages not necessary related to the UML. Moreover, in terms of expressiveness, non-determinism and executability (as ASMs support) are two important features to be taken into account for the specification and validation of the behavior of distributed applications and application components.

The ASMs formalism itself can be also intended as an action language but with a concise, abstract and powerful set of action schemes. That allows to overcome some limits of conventional action languages based on the UML AS. These last, – though they aim to be pragmatic, extensible and modifiable – may suffer from the same shortcomings and complexity of traditional programming languages being too much platform-specific.

Moreover, not all action semantics proposals are powerful enough to reflect a particular model of computation (MoC) underlying the nature of the application being modeled. This, instead, is not true for the ASMs.

Through several case studies, ASMs have shown to be a formal method suitable for system modeling and, in particular, for describing the semantics of modeling/programming languages. Among successful applications of the ASMs in the field of language semantics, we can cite the UML and SDL-2000, programming languages such as Java, C/C++, and hardware description languages (HDLs) such as SystemC, SpecC, and VHDL – complete references can be found in [6]. Concerning the ASM application to provide an executable and rigorous UML semantics, we can mention the works in [30, 5, 7, 23, 10]. More or less, all these approaches define an ASM model able to capture the semantics of a particular kind of UML graphical sub-language (statecharts, activity diagrams, etc.). Other attempts in this direction but generalized to any metamodel-based language – and therefore belonging to category (II) – are the works in [8, 12], to name a few. However, the use of the ASMs we suggest here is different. Here we focus on the use of the ASMs as "modeling language" (at the same level of the UML) rather than as "meta-language" (or semantics specification language). The goal is to provide a general virtual machine at PIM level by "weaving" executable behavior directly into structural models.

3. ABSTRACT STATE MACHINES

Abstract State Machines (ASMs) are an extension of FSMs [4], where unstructured control states are replaced by states comprising arbitrary complex data.

Although the ASM method comes with a rigorous mathematical foundation [6], ASMs provides accurate yet practical industrially viable behavioral semantics for pseudocode on arbitrary data structures. This specification method is tunable to any desired level of abstraction, and provides rigor without formal overkill.

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

Basically, a transition rule has the form of *guarded update* "**if** *Condition* **then** *Updates*" where *Updates* are a set of function updates of the form $f(t_1, \ldots, t_n) := t$ which are simultaneously executed[1] when *Condition* is true.

These is a limited but powerful set of *rule constructors* that allow to express simultaneous parallel actions (`par`) of a single agent, either in an atomic way, *Basic ASMs*, or in a structured and recursive way, *Structured or Turbo ASMs*, by sequential actions (`seq`), iterations (`iterate`, `while`, `rec-while`), and submachine invocations returning values. Appropriate rule constructors also allow non-determinism (existential quantification `choose`) and unrestricted synchronous parallelism (universal quantification `forall`). Furthermore, it supports a generalization where multiple agents interact in parallel in a synchronous/asynchronous way, *Synchronous-/Asynchronous Multi-agent ASMs*.

Based on [6], an ASM can be defined as the tuple:

(*header*, *body*, *main rule*, *initialization*)

The *header* contains the *name* of the ASM and its *signature*[2], namely all domain, function and predicate declarations. Function are classified as *derived* functions, i.e. those coming with a specification or computation mechanism given in terms of other functions, and *basic* functions which can be *static* (never change during any run of the machine) or *dynamic* (may change as a consequence of agent actions or *updates*). Dynamic functions are further classified into: *monitored* (only read, as events provided by the environment), *controlled* (read and write), *shared* and *output* (only write) functions.

The *body* of an ASM consists of (static) domain and (static/derived) function definitions according to domain and function declarations in the signature of the ASM. It also contains declarations (definitions) of transition rules. The body of ASM may also contains definitions of *axioms* for invariants one wants to assume for domains and functions of the ASM.

The (unique) *main rule* is a transition rule and represents the starting point of the machine program (i.e. it calls all the other ASM transition rules defined in the body). The main rule is *closed* (i.e. it does not have parameters) and since there are no free global variables in the rule declarations of an ASM, the notion of a move does not depend on a variable assignment, but only on the state of the machine.

The *initialization* of an ASM is a characterization of the initial states. An initial state defines an initial value for domains and functions declared in the signature of the ASM. *Executing* an ASM means executing its main rule starting from a specified initial state. A *computation* of M is 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.

3.1 The ASM Metamodel and ASMETA

In addition to its mathematical-based foundation, a metamodel-based definition for ASMs is also available. The ASM metamodel, called *AsmM* (*Abstract State Machines Metamodel*) [33, 15, 17, 2], provides an abstract syntax for an ASM language in terms of MOF concepts, and has been defined with the goals of developing a *unified* abstract notation for the ASMs, independent from any specific implementation syntax and allowing a more direct encoding of the ASM mathematical concepts and constructs.

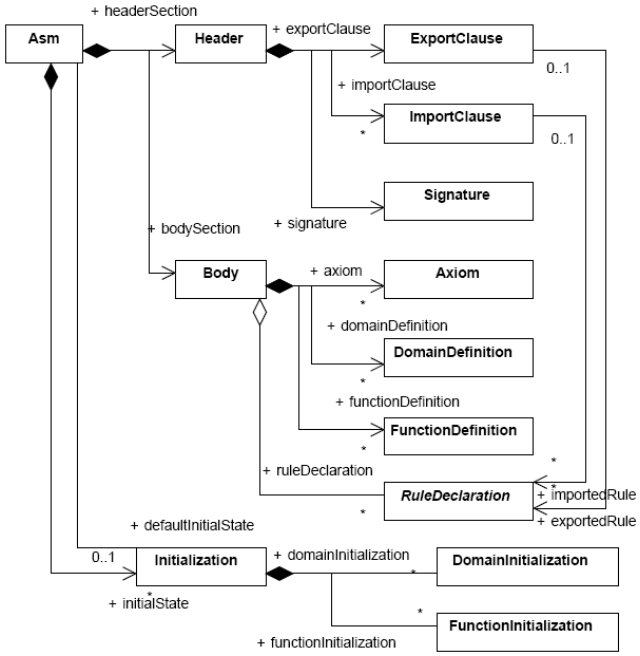

Figure 1: Backbone

Fig. 1 shows a very small fragment of the AsmM metamodel representing the structure of an ASM model.

AsmM is publicly available (see [2]) in the meta-language EMF/Ecore [13].

The AsmM semantics was given by choosing a semantic domain S_{AsmM} and defining a *semantic mapping* $M_S : AsmM \rightarrow S_{AsmM}$ to relate syntactic concepts to those of the semantic domain. S_{AsmM} is the first-order logic extended with the logic for function updates and for transition rule constructors formally defined in [6].

A general framework, called *ASMETA tool set* [16, 2], has been developed based on the AsmM and exploiting the advantages of the metamodelling techniques. It essentially

[1] 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.

[2] *Import* and *export* clauses can be also specified for modularization.

includes: a textual notation, *AsmetaL*, to write ASM models (conforming to the AsmM) in a textual and human-comprehensible form; a text-to-model compiler, *AsmetaLc*, to parse ASM models written in AsmetaL and check for their consistency with respect to the OCL constraints of the metamodel; a simulator, *AsmetaS*, to execute ASM models (stored in a model repository as instances of AsmM); the *Avalla* language for scenario-based validation of ASM models, with its supporting tool, the *AsmetaV* validator; the *ATGT* tool that is an ASM-based test case generator based upon the SPIN model checker; a graphical front-end, called *ASMEE* (ASM Eclipse Environment), which acts as IDE and it is an Eclipse plug-in.

4. WEAVING EXECUTABLE BEHAVIOR INTO UML CLASS MODEL

The aim of this technique is to weave behavioral aspects into the whole UML metamodel or parts of it, depending on those elements one is interested to express behavior about.

Applying this technique demands the definition of a *weaving function* specifying how the UML metamodel and the AsmM are weaved together into a new metamodel which adds to the UML the capability of specifying behavior by ASM transition rules. More precisely, it requires identifying precise *join points*[3] between data and behavior [29], to express how behavior can be attached to structural constructs.

Once a weaving function has been established between the UML and the AsmM, the resulting metamodel, in the sequel referred as UML^+, enriches the UML with behavior specification capability in terms of ASM transition rules. Therefore, UML$^+$ can be considered an abstract structural and executable language at PIM level.

As example of weaving executable behaviors into structural models by using ASMs, we here consider the portion of the UML metamodel concerning with class diagrams. However, the weaving process described here is directly applicable to any object-oriented metamodel and meta-metamodel like the OMG MOF, EMF/ECore, AMMA/KM3, etc.

4.1 Join Points Identification

In case of the UML metamodel, as for any other MOF metamodel, it might be convenient to use transition rules within meta-classes as class operations to hold their behavioral specification. Therefore, a join point must be specified between the class `Operation`[4] of the UML (see Fig. 7.11 in [37]) and the class `RuleDeclaration` of the AsmM.

Fig. 2 shows how simply the composition may be carried out. The MOF `Operation` class resembles the AsmM `RuleDeclaration` class. The name `Operation` has been kept instead of `RuleDeclaration` to ensure UML conformance; similarly, the name `Parameter` has been kept instead of `VariableTerm`. Finally, the new property `isMain` has been added in order to designate, when set to true, a *closed* (i.e. without formal parameters) operation as (unique) main rule of an *active* class (the *main* class) to start model execution.

[3]Inspired from the Aspect-oriented Programming (AOP) paradigm, join points are intended here and in [29] as places of the meta-metamodel where further (executability) aspects can be injected.

[4]An operation is a behavioral feature of a classifier that specifies the name, type, parameters, and constraints for invoking an associated behavior.

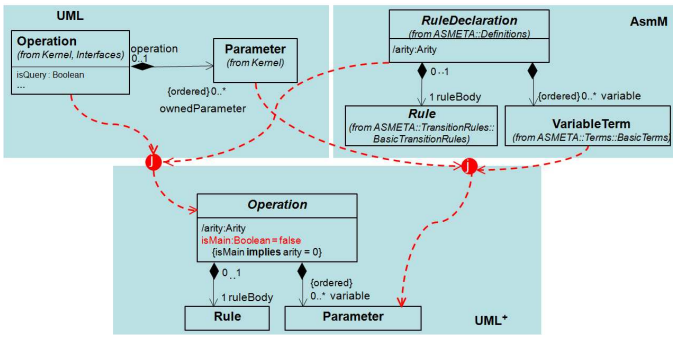

Figure 2: Using operation bodies as join points between data and behaviour

A further join point is necessary to adorn UML class's properties (either attributes or association member ends – see Fig. 7.12 in [37]) to reflect the ASM function classification. Fig. 3 shows how this may be carried out. The UML `Property` class resembles the AsmM `Function` class. Box UML$^+$ presents the result of the composition process. The UML class `Property` has been merged with the class `Function`. A further adornment `kind:PropertyKind` have been added to capture the complete ASM function classification. `PropertyKind` is an enumeration of the following literal values: static, monitored, controlled, out, and shared. Two OCL constraints have been also added stating, respectively, that a *read-only* (attribute `isReadOnly` is set to true) property can be of kind static or monitored, and that if a property is *derived* (attribute `isDerived` is set to true) then the attribute kind is empty.

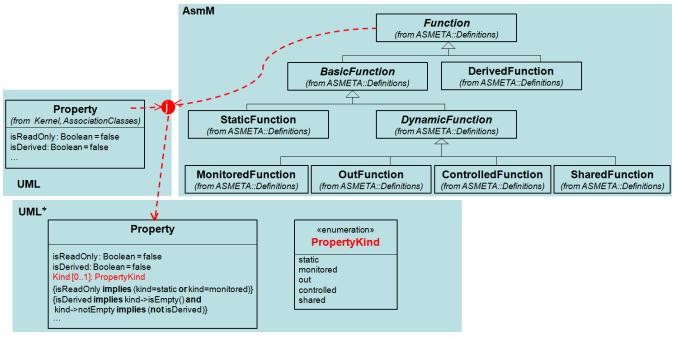

Figure 3: Using properties as join point for ASM adornments

Moreover, in order to merge the two statically-typed systems of the UML and the AsmM, a UML `Type` (a `Class` or a `DataType`) is merged with an ASM `Domain`. Finally, values specification in UML class models (e.g. for specifying default values of attributes) are provided in terms of *opaque expressions* (instances of the `OpaqueExpression` class in the UML metamodel)[5] that are merged with ASM terms (the `Term` class of the AsmM metamodel).

5. SEMANTIC MODEL

[5]In UML, an opaque expression is an uninterpreted textual statement that denotes a (possibly empty) set of values when evaluated in a context

At this point of the weaving process, we are able to design by the UML$^+$ terminal models [24] whose syntactic elements conform to UML and whose operation semantics is expressed in terms of ASM rules. (Sect. 7 reports an example of application). So doing, the ASM formal notation can be considered as an abstract action language for UML at PIM level. However, following our approach, we are able to provide more, namely to define in a precise and clear way the executable semantics of a terminal model conforming to UML$^+$, by associating it with its ASM semantic (executable) model. This is clarified by the following argumentation that refers to a generic metamodel (more details can be found in [18]), but which is then tailored for the UML$^+$ in Sect. 5.1.

A language metamodel A has a well-defined semantics if a semantic domain S is identified and a semantic mapping $M_S : A \rightarrow S$ is provided [21] to give meaning to syntactic concepts of A in terms of the semantic domain elements. By exploiting the ASM formal method endowed with a metamodel representation of their concepts and with a precise mathematical semantics, we can express the semantics of a terminal model [24] conforming to A in terms of an ASM model.

Let us assume the semantic domain S_{AsmM} of the ASM metamodel (see Sect. 3.1) as the semantic domain S. The semantic mapping $M_S : A \rightarrow S_{AsmM}$, which associates a well-formed terminal model m conforming to A with its semantic model $M_S(m)$, can be defined as

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

where $M_{S_{AsmM}} : AsmM \rightarrow S_{AsmM}$ is the semantic mapping of the ASM metamodel and associates a theory conforming to the S_{AsmM} logic with a model conforming to $AsmM$, and the function $M : A \rightarrow AsmM$ associates an ASM to a terminal model m conforming to A. Therefore, the problem of giving the metamodel semantics is reduced to define the function M between metamodels. The complexity of this approach depends on the complexity of building the function M. In the following section, we show how to build the function M for the UML$^+$ metamodel.

5.1 Semantic Model of UML$^+$

The building function $M : UML^+ \longrightarrow AsmM$ is defined as

$$M(m) = \iota(W(m), m)$$

for all terminal model m conforming to UML^+, where:
– $W : UML^+ \rightarrow AsmM$ maps a weaved terminal m conforming to UML^+ into a model conforming to the AsmM and provides the abstract data structure (signature, domain and function definitions, axioms) and the transition system of the final machine $M(m)$;
– $\iota : AsmM \times UML^+ \rightarrow AsmM$) computes the initial state of the final machine $M(m)$ by extracting initial values for data structures of the machine from the source modeling elements in m.

The function W is defined as a mapping in Table 1 and provides semantics both to the basic modeling elements characterizing UML class models (although we apply some restrictions as remarked below) and to the enriched UML$^+$ modeling elements as result of the weaving process. The semantics is given by associating each modeling concept into a corresponding AsmM modeling element. Below, we comment those parts of W concerning UML elements involved into the join points definition, while we leave to the reader intuition the understanding of the remaining parts.

The `Operation` element of the weaved language UML$^+$ is associated to the corresponding `RuleDeclaration` element of the AsmMas involved in the join point definition. We assume, similarly to the use of `this` in the Java programming language, that in the definition of the rule body of an operation *op*, a special variable named $\$this$ is used to refer to the object that contains the operation. The W function application automatically adds the variable $\$this$ as formal parameter of the corresponding rule declaration[6]. Moreover, for simplicity (although these concepts can be handled in ASMs) we assume that an operation cannot raise exceptions (i.e. the set of types provided by the association end `raisedException` is empty) and does not specify constraints.

The W function provides semantics to the `Property` element (an attribute or an association end) of the UML$^+$ language by associating it to the corresponding `Function` element of the AsmM involved in the join point definition.

Due to UML and AsmM types identification, as explained in Table 1, the domain of the ASM function denoting a property has to be intended as the ASM domain (usually an `AbstractTD` type-domain) induced from the exposing class of the property, and the codomain as induced from the property's type.

Opaque expressions are straightforwardly matched (see Table 1) into ASM terms.

Remark. Currently, we apply some restrictions to the UML metamodel[7]. We assume exceptions cannot arise from operations, no default values can be specified for the operations parameters, no pre/post conditions and body conditions for operations, no qualifiers (like derived unions and subsetting) as optional part of association ends can be specified, no visibility kinds, only simple classes are treated (i.e. no composite classes with parts and ports), association classes are not yet supported, and only binary associations with none aggregation type are currently permitted. Moreover, as we concerned to PIM level, we assume that class features (both properties and operations) are not *static*, since for static features two alternative semantics are recognized in UML[8] leading therefore to alternative implementations that should instead be taken at PSM level.

5.2 UML$^+$ Action Semantics

According to the ASM semantic domain, operations can be invoked on an object (an element of an ASM domain) of a terminal model, given a particular set of substitutions for the parameters of the operation. An operation invocation may cause changes to the values of the properties of that object, or of other objects that can be navigated to, directly or indirectly, from the object's context on which the operation

[6]Since operations are intended as ASM transition rules, within the body of an operation *op*, if f is a property (an attribute or an association end) in the same object, then $f(\$this)$ must be used as a full name for that property. If *anotherOp* is another operation in the same object, then $anotherOp(\$this,\ldots)$ must be used to invoke that operation.

[7]They do not limit our approach and can be considered in the future.

[8]A static feature may have different values for different featuring classifiers, or the same value for all featuring classifiers.

is invoked, to its output parameters, to objects navigable from its parameters, or to other objects in the scope of the operation's execution. An operation invocation may return a value as a result, and in this case the semantics is that of a Turbo ASM rule with return value. Operation invocations may also cause the creation and deletion of objects by executing *extend* ASM rules and update rules of set-functions. Expression evaluations are supported as well.

In addition to these basic actions (property getting/setting, expression evaluations, operation invocation, object creation/deletion), the weaving between the UML and the AsmM metamodels allows us to use more sophisticated ASM rule constructors to express behavior: if-then-else, parallel execution (*par* rules), sequential execution (*seq* rules), finite iteration submachines (*iterate* and *while* rules), non-determinism (*choose* rule), etc., as formally defined in [6].

It is possible to make use of the parallel ASM execution model that (a) easens specification of macro steps (refinement and modularization), (b) avoids unnecessary sequentialization of independent actions, (c) easens parallel/distributed implementations [6]. Furthermore, one can exploit the ASMs feature of incorporating non-atomic structuring concepts (by the constructor *seq* for sequentialization) and finite iteration submachines with return values, exception handling, local values, etc., as standard refinements into synchronous parallel ASMs.

The idea here is to extend the UML class model to allow the definition of ASM transition rules working as "pseudo-code over classes" as scheme of a generic control machine, and allow therefore different granularities of computation step for validating objects behavior, even for parallel/distributed behavioral facets, at PIM level.

6. IMPLEMENTATION

We have been implementing an Eclipse-based integrated environment made of: a UML modeler; the ASMETA/AsmetaS tool, as execution environment; and the AMW (ATLAS Model Weaver) [1] and ATL (ATLAS Transformation Language) [22] to handle the merging process between the UML and the AsmM. The tool implemented at the moment is still a prototype; e.g., we have been carrying our experiments with the EMF-based implementation of the UML 2.x metamodel for Eclipse rather then using directly an external UML visual modeling tool.

Once the ASM semantic model is obtained from a UML$^+$ terminal model (see the building function M in Sect. 5.1), several reasoning activities can be carried out, early at PIM level, by exploiting the ASMETA toolset. Model validation is possible by random, interactive, and scenario-based simulation, or by automatic test case generation. Model verification can be done through model checking techniques. Furthermore, this high level ASM model can be exploited for conformance analysis of refined PSMs and code models.

7. CASE STUDY: INVOICE ORDER SYSTEM

The Invoice Order System (IOS), taken from [20], is used as an example to illustrate our approach. The subject is to invoice orders (R0.1). To invoice is to change the state of an order from *pending* to *invoiced* (R0.2). On an order, we have one and only one reference to an ordered product of a certain quantity; the quantity can be different from other orders (R0.3). The same reference can be ordered on several different orders (R0.4). The state of the order will be changed to invoiced if the ordered quantity is either less than or equal to the quantity which is in stock according to the reference of the ordered product (R0.5). From the set of requirements presented in [20], we focus here on the *Case 1*, which is specified as follows:

R1.1 All the ordered references are in stock.

R1.2 The stock or the set of the orders may vary due to the entry of new orders or canceled orders, or due to having a new entry of quantities of products in stock at the warehouse. But we do not have to take these entries into account.

R1.3 This means that you will not receive two entry ows (orders, entries in stock). The stock and the set of orders are always given to you in an up-to-date state[9].

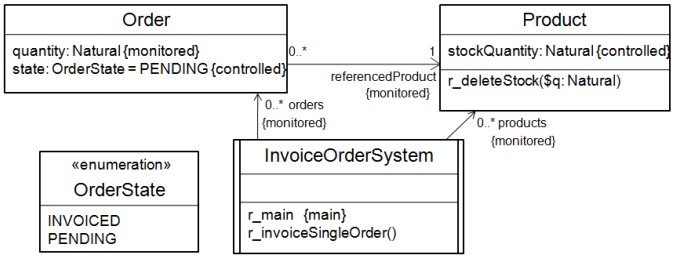

Figure 4: IOS class model

The UML class diagram in Fig. 4 shows the implementation classes for the Invoice Order System at the PIM level. It shows the internal structure of the system for order management with the essential details at this stage. From a structural point of view, there is: a set of orders (the `Order` class), a set of products (the `Product` class), and a container class `InvoiceOrderSystem` used as *active*[10] class to start the application and thus to invoice orders depending on various strategies. Every order has a state, which can be *invoiced* or *pending*. All the orders are initially pending. Every order refers to a product for a certain quantity (greater than zero) and these data cannot be changed (quantity is a `monitored` property). The same product can be referenced by several different orders. Every product is in the stock in different quantity. The quantity of a product in the stock is only updated by the system (hence it is a `controlled` property) when it invoices some orders.

The behavior of each class is specified by means of ASM transition rules, as shown in the operation compartment of the UML classes. Their definition is reported in Listing 1 using the ASMETA/AsmetaL textual notation. The system is intended as a single-agent machine. To invoice orders, the system may follow different strategies. We choose here that of invoicing an order at a time. By invoking the `r_invoiceSingleOrder` operation, the system selects (non-deterministically) on `order`[11] within a set of orders that are pending and refer to a product in the stock in enough quantity, and simultaneously changes the state of the selected order from pending to invoiced and updates the stock by

[9]You do not have to take into account the entry of new orders, cancellation of orders, and entries of quantities in the stock. These are subjects for Case 2.

[10]Each instance of an active class has its own thread of control and possibly may coordinate other behaviors.

[11]Note that a variable `v` is expressed in AsmetaL as `$v`.

UML^+	**AsmM**
An active class C	An ASM containing in its signature a domain `C` as subset of the predefined domain `Agent`
A non-abstract class C	A dynamic `AbstractTD` domain C
An abstract class C	A static `AbstractTD` domain C
An `Enumeration`	An `EnumTD` domain
A primitive type	A basic type domain
`Boolean`	`BooleanDomain`
`String`	`StringDomain`
`Integer`	`IntegerDomain`
`UnlimitedNatural`	`NaturalDomain`
A *generalization* between a child class *C1* and a parent class *C2*	A `ConcreteDomain` C_1 subset of the corresponding domain C_2
An attribute a of a class C, type T, kind k, and multiplicity 1	A function $a : C \rightarrow T$ of kind k
An attribute a of a class C, type T, multiplicity > 1, and *ordered*	A function $a : C \rightarrow T^*$ of kind k, where T^* is the domain of all finite sequences over T (`SequenceDomain`)
An attribute a of a class C, type T, kind k, and multiplicity > 1, *unordered* and *unique*	A function $a : C \rightarrow \mathcal{P}(T)$ of kind k, where $\mathcal{P}(T)$ is the mathematical powerset of T (`PowersetDomain`)
An attribute a of a class C, type T, kind k, and multiplicity > 1, *unordered* and *not unique*	A function $a : C \rightarrow B(T)$ of kind k, where $B(T)$ is the domain of all finite bags over T (`BagDomain`)
A navigable association end	See attribute
An operation *op* of a class C, rule body R, arity n, and owned parameters $x_i : D_i$	A rule declaration $op(\$this$ **in** C, x_1 **in** $D_1, \ldots, x_n$ **in** $D_n)$ = R of rule body R, arity *n+1*, and formal parameters $\$this$ **in** C and x_i **in** D_i
A closed operation *op* of a class C, rule body R, and with *isMain* set to true	The (unique) main rule declaration of form **main rule** *op* = **for all** $\$this$ **in** C **do** R
An opaque expression	A term
OCL constraints	Axioms (optional)
OCL operations/queries	Static functions (optional)

Table 1: *W* mapping: from UML^+ to AsmM

subtracting the total product quantity in the order to invoice (by the `r_deleteStock` operation). The system keeps to invoice orders as long as there are orders which can be invoiced. The system guarantees that the state of an order is always defined and the stock quantity is always greater than or equal to zero. Note that, non-determinism (`choose` rule) is a convenient way to abstract from details of scheduling. Indeed, in the modeled strategy, per step at most one order is invoiced, with an unspecified schedule (not taking into account any arrival time of orders) and with a deletion function under the assumption that `stockQuantity` is updated only by invoicing.

Listing 1: IOS behaviour: single-order strategy

```
rule r_invoiceSingleOrder =
  choose $order in self.orders with orderState($order) = PENDING
        and quantity($order) <=
                        stockQuantity(referencedProduct($order))
    do par
      state($order) := INVOICED
      r_deleteStock[referencedProduct($order),orderQuantity($order)]
    endpar

rule r_deleteStock($p in Product, $q in Natural) =
  stockQuantity($p):= stockQuantity($p) - $q

main rule r_Main = r_invoiceSingleOrder[]
```

Many other strategies and particular scheduling algorithms can be defined and refined as well, in order to get a sufficiently precise, complete and minimal, executable PIM model which can serve as a basis for the implementation of various PSMs. Appendix A reports the complete AsmetaL specification associated to the IOS class model in Fig. 4.

8. CONCLUSION AND FUTURE WORK

In this paper, we proposed a PIM level behavioral language for structural models based on the ASMs formalism. We focused on an *intra-object* perspective by addressing the behavior occurring within structural entities (like UML class models). In the future, we propose to extend the behavioral formalism for the *inter-object* behavior, which deals with how structural entities communicate with each other. The objective of this further effort is to show the applicability of the proposed approach in the area of communication protocols and of inter-process interaction models. This will require identifying suitable join points between structural diagrams describing the collaborative structure of the interactive entities and the AsmM subpart concerning transition rules.

We will also continue working on the implementation of the model execution environment (a virtual machine) and a user interface providing support for a user-friendly model simulation. We could also experiment with a much more lightweight extension of the UML metamodel based on the UML *profile mechanism*, but we preferred a more general matching approach in order to make it reusable for different metamodels rather than only for UML-based metamodels.

Moreover, we want to connect PIM executable models with PSM executable models written with the SystemC UML profile in the context of a model-based development process for embedded systems and System-on-Chip (SoC) [34].

9. REFERENCES

[1] The AMW (ATLAS Model Weaver website). `http://www.eclipse.org/gmt/amw/`, 2007.

[2] The Abstract State Machine Metamodel website. `http://asmeta.sf.net/`, 2006.

[3] J. Bézivin. On the Unification Power of Models. *Software and System Modeling (SoSym)*, 4(2):171–188, 2005.

[4] E. Börger. The ASM method for system design and analysis. A tutorial introduction. In B. Gramlich, editor, *FroCoS 2005, Proceedings*, LNCS 3717, pages 264–283. Springer, 2005.

[5] E. Börger, A. Cavarra, and E. Riccobene. Modeling the dynamics of UML state machines. In Y. G. et al., editor, *Abstract State Machines. Theory and Applications*, LNCS 1912, pages 223–241. Springer, 2000.

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

[7] A. Cavarra, E. Riccobene, and P. Scandurra. Mapping uml into abstract state machines: a framework to simulate uml models. *J. Studia Informatica Universalis*, 3(3):367–398, 2004.

[8] K. Chen, J. Sztipanovits, and S. Neema. Compositional specification of behavioral semantics. In *DATE*, pages 906–911, 2007.

[9] B. Combemale, P. G. X. Crégut, and X. Thirioux. Towards a formal verification of process models's properties - simplepdl and tocl case study. In *9th International Conference on Enterprise Information Systems (ICEIS)*, 2007.

[10] K. Compton, J. Huggins, and W. Shen. A semantic model for the state machine in the Unified Modeling Language. In Proc. of Dynamic Behavior in UML Models: Semantic Questions, UML 2000, 2000.

[11] G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varró. VIATRA - Visual Automated Transformations for Formal Verification and Validation of UML Models. In *ASE*, pages 267–270. IEEE Computer Society, 2002.

[12] D. Di Ruscio, F. Jouault, I. Kurtev, J. Bézivin, and A. Pierantonio. Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs. Technical Report 06.02, LINA, 2006.

[13] Eclipse Modeling Framework. `http://www.eclipse.org/emf/`, 2008.

[14] OMG. Semantics of a Foundational Subset for Executable UML Models, version 1.0 - Beta 1, ptc/2008-11-03, 2008.

[15] A. Gargantini, E. Riccobene, and P. Scandurra. Metamodelling a Formal Method: Applying MDE to Abstract State Machines. Technical Report 97, DTI Dept., University of Milan, 2006.

[16] A. Gargantini, E. Riccobene, and P. Scandurra. A metamodel-based simulator for ASMs. In A. Prinz, editor, *Proceedings of the 14th International ASM Workshop*, 2007.

[17] A. Gargantini, E. Riccobene, and P. Scandurra. Ten reasons to metamodel ASMs. In *Dagstuhl Workshop on Rigorous Methods for Software Construction and Analysis, LNCS Festschrift*. Springer, 2007.

[18] A. Gargantini, E. Riccobene, and P. Scandurra. A semantic framework for metamodel-based languages. *Journal of Automated Software Engineering*, 2009, in print.

[19] The Generic Modeling Environment (GME). `http://www.isis.vanderbilt.edu/Projects/gme`, 2006.

[20] H. Habrias and M. Frappier. *Software Specification Methods: An Overview Using a Case Study*. Wiley, 2006.

[21] D. Harel and B. Rumpe. Meaningful modeling: What's the semantics of "semantics"? *IEEE Computer*, 37(10):64–72, 2004.

[22] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez. ATL: a QVT-like transformation language. In *OOPSLA '06*, pages 719–720. ACM, 2006.

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

[24] I. Kurtev, J. Bézivin, F. Jouault, and P. Valduriez. Model-based dsl frameworks. In *OOPSLA Companion*, pages 602–616, 2006.

[25] OMG. The Model Driven Architecture (MDA Guide V1.0.1). `http://www.omg.org/mda/`, 2003.

[26] H. Miao, L. Liu, and L. Li. Formalizing uml models with object-z. In *ICFEM '02: Proc. of the 4th Int. Conference on Formal Engineering Methods*, pages 523–534, London, UK, 2002. Springer-Verlag.

[27] P. Mohagheghi and V. Dehlen. Where is the proof? - a review of experiences from applying mde in industry. In I. Schieferdecker and A. Hartman, editors, *ECMDA-FA*, volume 5095 of *LNCS*, pages 432–443. Springer, 2008.

[28] A. M. Mostafa, M. A. Ismail, H. E. Bolok, and E. M. Saad. Toward a Formalization of UML2.0 Metamodel using Z Specifications. In *SNPD 2007, Proceedings*, volume 1, pages 694–701, 2007.

[29] P.-A. Muller, F. Fleurey, and J.-M. Jezequel. Weaving Executability into Object-Oriented Meta-Languages. In *Proc. of ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems*, 2005.

[30] I. Ober. More meaningful UML Models. In *TOOLS - 37 Pacific 2000*. IEEE, 2000.

[31] OMG. Object Constraint Language (OCL), v2.0 formal/2006-05-01, 2006.

[32] C. Raistrick, P. Francis, and J. Wright. *Model Driven Architecture with Executable UML*. Cambridge University Press, 2004.

[33] E. Riccobene and P. Scandurra. Towards an Interchange Language for ASMs. In W. Zimmermann and B. Thalheim, editors, *Abstract State Machines. Advances in Theory and Practice*, LNCS 3052, pages 111 – 126. Springer, 2004.

[34] E. Riccobene and P. Scandurra. Model transformations in the UPES/UPSoC development process for embedded systems. *Innovations in Systems and Software Engineering*, 5(1):35–47, 2009.

[35] M. Scheidgen and J. Fischer. Human comprehensible and machine processable specifications of operational semantics. In *ECMDA-FA*. Springer, 2007. LNCS.

[36] C. Snook and M. Butler. Uml-b: Formal modeling and design aided by uml. *ACM Trans. Softw. Eng. Methodol.*, 15(1):92–122, 2006.

[37] OMG. UML v2.2 Superstructure, formal/09-02-02, 2009.

[38] The Xactium XMF Mosaic. `www.modelbased.net/www.xactium.com/`, 2007.

APPENDIX

A. ASM MODEL FOR THE IOS SYSTEM

Listing 2: ASM model for the IOS (single-order strategy)

```
asm InvoiceOrderSystem //Case 1
 import STDL/StandardLibrary
 signature:

 //Domain declarations
  domain InvoiceOrderSystem subsetof Agent
  abstract domain Order
  abstract domain Product
  enum domain OrderState = { INVOICED | PENDING }

  //Function declarations
  dynamic controlled state: Order -> OrderState //the product referenced in an order
  dynamic monitored quantity: Order -> Natural //the quantity in the order
  dynamic controlled stockQuantity: Product -> Natural //the quantity in the stock
  dynamic monitored referencedProduct: Order -> Product //the product referenced in an order
  dynamic monitored orders: InvoiceOrderSystem -> Powerset(Order) //the referenced orders
  dynamic monitored products: InvoiceOrderSystem -> Powerset(Product) //the referenced products

 definitions: /*------ Rules for case 1 (single-order strategy) -------*/

  macro rule r_deleteStock($p in Product ,$q in Natural)= stockQuantity($p):= stockQuantity($p) - $q

  rule r_invoiceSingleOrder =
    choose $order in self.orders with orderState($order) = PENDING and quantity($order) <= stockQuantity(referencedProduct($order))
    do par
         state($order) := INVOICED
         r_deleteStock[referencedProduct($order),orderQuantity($order)]
       endpar

/*--------- main rule ----------*/
main rule r_main =
        r_invoiceSingleOrder[]

//A possible initial state
default init s_1:
        function state($o in Order) = PENDING //defaul value from the class diagram
        function stockQuantity($p in Product) = 100n
```