3B2v7.51c GML4.3.1 IS : 307 Prod.Type: com pp.1–13(col.fig.:NIL) ED:KCT PAGN: ushamani SCAN: Mallik

ARTICLE IN PRESS

PERGAMON

Information Systems 0 (▮▮▮▮) ▮▮▮–▮▮▮

www.elsevier.com/locate/infosys

Using UML Action Semantics for model execution and transformation ☆

Gerson Sunyé*, Alain Le Guennec, Jean-Marc Jézéquel

IRISA, Campus de Beaulieu, F-35042 Rennes Cedex, France

Abstract

The Unified Modelling Language (UML) lacks precise and formal foundations and semantics for several modeling constructs, such as transition guards or method bodies. These semantic discrepancies and loopholes prevent executability, making early testing and validation out of reach of UML tools. Furthermore, the semantic gap from high-level UML concepts to low-level programming constructs found in traditional object-oriented language prevents the development of efficient code generators.

The recent *Action Semantics* (AS) proposal tackles these problems by extending the UML with yet another formalism for describing behavior, but with a strong emphasis on dynamic semantics. This formalism provides both, a metamodel integrated into the UML metamodel, and a model of execution for these statements. As a future OMG standard, the AS eases the move to tool interoperability, and allows for executable modeling and simulation.

We explore in this paper a specificity of the AS: its applicability to the UML metamodel, itself a UML model. We show how this approach paves the way for powerful metaprogramming for model transformation. Furthermore, the overhead for designers is minimal, as mappings from usual object-oriented languages to the AS will be standardized.

Keywords: UML; Action Semantics

1. Introduction

The Unified Modeling Language (UML) provides many diagrams offering powerful abstractions for modeling systems while preserving an adequate separation of concerns. For instance, state charts, sequence diagrams or collaboration diagrams expose different aspects of the same behavior. But the UML currently lacks some precise and formal foundation for several constructs such as transition guards or method bodies, for which it resorts to semantic loopholes in the form of *uninterpreted* expressions. It is then very difficult to carry on key software quality related best practices such as early simulation and validation or test case generation starting from standard UML models.

To cope with these issues, the Object Management Group (OMG) issued a request for proposal (RFP) for an *Action Semantics* (*AS*) *for the UML* [1]. It aims at integrating a *precise*, *software-independent action specification* into the UML. The AS specification [2] is a response to this RFP. It builds on the existing UML and extends its

☆ Recommended by ■ ■

*Corresponding author.

E-mail addresses: gerson.sunye@free.fr (G. Sunyé), aleguen@free.fr (A. Le Guennec), jezequel@irisa.fr (J.-M. Jézéquel).

0306-4379/02/$ - see front matter © 2002 Published by Elsevier Science Ltd.
PII: S0306-4379(02)00014-5

meta-model with a large set of new constructs. It also defines a *model of execution* and a *semantics of actions*. Building on the insight we obtained by contributing some of its precise semantic definition, we briefly present the AS in Section 2. Along the lines of what already exists in the IUT-T Specification and Description Language (SDL) community [3], the integration of the AS into UML should ease the move to tool interoperability, and allow for executable modeling and simulation, as well as full code or test case generation.

But the interest of the AS does not end there. Relying on the fact that the UML metamodel (i.e. the model describing the UML) is itself a UML model, we show in this article how the AS can be used at the metamodel level to help the OO designer carry on activities such as behavior-preserving transformations [4], design pattern application [5] and design level aspects weaving [6]. Our intention is not to propose yet another approach for model transformation, pattern application or refactoring. What we claim here, is that the AS may (and should) go beyond the design level and be used as a metaprogramming language to help the implementation of existing approaches. Also, as we show in the following sections, it has strengths at both levels: at the model level (as a design level language) and at the metamodel level (as a metamodel level programming language).

In all these design level activities, we can distinguish two steps: (1) the identification of the need to apply a given transformation on a UML model; and (2) the actual transformation of that model. Our intention is not to usurp the role of the designers in deciding what to do in step 1, but to provide them with tools to help automate the second step, which is usually very tedious and error prone. Further, when carried out in an ad hoc manner, it is very difficult to keep track of the *what*, *why* and *how* of the transformation, thus leading to traceability problems and a lack of reusability of the design micro-process. This can be seen in maintenance, when one has to propagate changes from the problem domain down to the detailed design by "re-playing" design decisions on the modified part of a model. Automation could also be very worthwhile in the context of product lines, when the same (or at least very similar) design decisions are to be applied on a family of analysis models (e.g. the addition of a persistence layer on many MIS applications).

Because the next OO designer cannot be expected to write complex meta-programs from scratch, in order to elaborate his design, he must be provided with pre-canned transformations (triggered through a menu), as well as ways of customizing and combining existing transformations to build new ones. The main interest of using the UML/AS at the metamodel level for expressing these transformations is that we can use classical OO principles to structure them into *reusable transformation components* (RTC): this is the open–closed principle [7] applied at the metamodel level. Furthermore, since the AS is fully integrated in the UML metamodel, it can be combined with rules written in the Object Constraint Language (OCL) [8], in order to verify whether a transformation (or a set of transformations) may be applied to a given context.

The rest of the paper is structured as follows. The AS proposal is introduced in Section 2. Section 3 shows the interest of using the AS at the metamodel level for specifying and programming model transformations in several contexts. In Section 4 we discuss related work, and we conclude on the perspectives open by our approach.

2. Executable modeling with UML

2.1. The bare-bone UML is incomplete and imprecise

The UML is based on a four-layer architecture, each layer being an instance of its upper layer, the last one being an instance of itself. The first layer holds the living entities when the code generated from the model is executed, i.e. running objects, with their attribute values and links to other objects. The second layer is the modeling layer. It represents the model as the designer conceives it. This is the place where classes, associations, state machines, etc., are defined. The running objects

are instances of the classes defined at this level. The third layer, the metamodel level, describes the UML syntax in a metamodeling language (which happens to be a subset of UML). This layer specifies what a syntactically correct model is. Finally, the fourth layer is the meta-metamodel level, i.e. the definition of the metamodeling language syntax, thus the syntax of the subset of UML used as a metamodeling language. UML creators chose a four-layer architecture because it provides a basis for aligning the UML with other standards based on a similar infrastructure, such as the widely used meta-object facility (MOF).

Although there is no strict one-to-one mapping between all the MOF meta-metamodel elements and the UML metamodel elements, the two models are interoperable: the UML core package metamodel and the MOF are structurally quite similar. This conception implies the UML metamodel (a set of class diagrams) is itself a UML model.

A UML model is said to be syntactically correct if the set of its views merge into a consistent instance of the UML metamodel. The consistency of this instance is ensured via the metamodel structure (i.e. multiplicities on association ends) and a set of well-formedness rules (WFR), expressed in OCL, which are logical constraints on the elements in a model. Examples of WFRs are: *there should not be any inheritance cycle* or *a FinalState may not have any outgoing transitions*.

But apart from those syntactic checks regarding the structure of models, UML users suffer from the lack of formal foundations for the important behavioral aspects, leading to some incompleteness and opening the door to inconsistencies in UML models. This is true, for instance, in state diagrams, where the specification of a guard on a transition is realized by a BooleanExpression, which is basically a string with no semantics. Thus, the interpretation is left to the modeling tool, jeopardizing interoperability. But more annoying is the fact that models are not executable, because they are incompletely specified in the UML. This makes it impossible to verify and test early in the development process. Such activities are key to assuring software quality.

2.2. The interest of an AS for UML

The Action Semantics proposal aims at providing modelers with a complete, software-independent specification for actions in their models. The goal is to make UML modeling executable modeling [1], i.e. to allow designers to test and verify early and to generate 100% of the code if desired. It builds on the foundations of existing industrial practices such as SDL, Kennedy Carter [9] or BridgePoint [10] action languages.[1] But contrary to its predecessors, it is intended that the AS become an OMG standard, and a common base for all the existing and to-come action languages (mappings from existing languages to AS are proposed).

Traditional modeling methods which do not have support for any action language have focused on separating analysis and design, i.e. the *what the system has to do* and the *how that will be achieved*. Whilst this separation clearly has some benefits, such as allowing the designer to focus on system requirements without spreading himself/herself too thinly with implementation details, or allowing the reuse of the same analysis model for different implementations, it also has numerous drawbacks. The main one is that this distinction is a difficult one to make in practice: the boundaries are vague; there are no criteria for deciding what is analysis, and what is not. Rejecting some aspects from analysis makes it incomplete and imprecise; trying to complete it often leads to the introduction of some *how* issues for describing the most complex behaviors.

As described above, the complete UML specification of a model relies on the use of uninterpreted entities, with no well-defined and accepted common formalism and semantics. This may be, for example, guards on transitions specified with the Object Constraint Language (OCL), actions in states specified in Java or C++.

In the worst case—that is for most of the modeling tools—these statements are simply inserted at the right place into the code skeleton. The semantics of execution is then given by the

[1] All the major vendors providing an action language are in the list of submitters.

specification of the programming language. Unfortunately, this often implies an over-specification of the problem (for example, in Java, a sequential execution of the statements of a method is supposed), verification and testing are feasible only when the code is available, usually far too late in the development process. Moreover, the designer must have some knowledge of the way the code is generated for the whole model in order to have a good understanding of the implications of her inserted code (for instance, if you are using the C++ code generator of a UML tool, a knowledge of the way the tool generates code for associations is required in order to use such code in your own program).

At best, the modeling tool has its own action language and then the model may be executed and simulated in the early development phases, but with the drawbacks of no standardization, no interoperability, and two formalisms for the modeler to learn (the UML and the action language).

2.3. The AS proposal submitted to the OMG

The AS proposal is based upon three abstractions:

- *A metamodel*: It extends the current metamodel. The AS is integrated smoothly in the current metamodel and allows for a precise syntax of actions by replacing all the previously uninterpreted items. The uninterpreted items are viewed as a surface language for the abstract syntax tree of actions (see Fig. 1).
- *An execution model*: It is a UML model. It allows the changes to an entity over time to be modeled. Each change to any mutable entity yields a new snapshot, and the sequence of snapshots constitutes a history for the entity. This execution model is used to define the semantics of action execution.
- *Semantics*: The execution of an Action is precisely defined with a *life cycle* which unambiguously states the effect of executing the action on the current instance of the execution model (i.e. it computes the next snapshot in history for the entity).

2.4. Surface language

The AS proposal for the UML does not enforce any notation (i.e. surface language) for the specification of actions. This is intentional, as the goal of the AS is certainly not to define a new notation, or to force the use of a particular existing one. The AS was, however, conceived to allow an easy mapping of classical languages such as Java, C++ or SDL.[2] Thus, designers can keep their favorite language without having to learn a new one.

3. Metaprogramming with the Action Semantics

An interesting aspect of UML is that its syntax is represented (or metamodeled) by itself (as a class diagram, actually). Thus, when using a reflexive environment (where the UML syntax is effectively represented by a UML model), the AS can be used to manipulate UML elements, i.e. transform models.

In the following sections, we present three different uses for this approach: implementing refactorings, applying design patterns and weaving design aspects.

3.1. Design refactorings

The activity of software design is not limited to the creation of new applications from scratch. Very often software designers start from an existing application and have to modify its behavior and functionality. In recent years, it has been widely acknowledged as a good practice to divide this evolution into two distinct steps:

(1) Without introducing any new behavior on the conceptual level, re-structure the software design to improve quality factors such as maintainability, efficiency, etc.
(2) Taking advantage of this "better" design, modify the software behavior.

This first step has been called *refactoring* [4], and is now seen as an essential activity during software development and maintenance. By definition, refactorings should be behavior-preserving

[2] Some of these mappings are illustrated in the AS specification document [11].

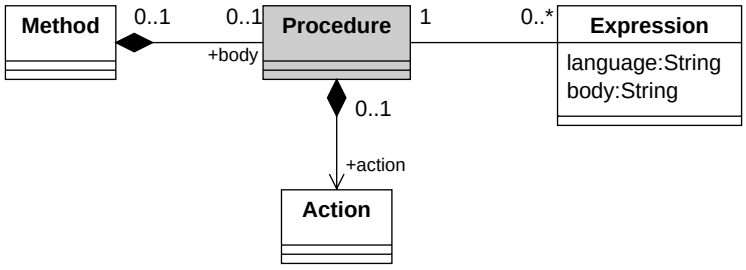

Fig. 1. Action Semantics immersion into the UML.

transformations of an application. But one of the problems faced by designers is that it is often hard to measure the actual impact of modifications on the various design views, as well as on the implementation code.

This is particularly true for the UML, with its various structural and dynamic views, which can share many modeling elements. For instance, when a method is removed from a class diagram, it is often difficult to establish, at first glance, what is the impact on sequence and activities diagrams, collaborations, statecharts, OCL constraints, etc.

Still, the UML also has a primordial advantage in comparison with other design languages: its abstract syntax is defined by a metamodel, where the integration of the different views is given meaning. Therefore, the metamodel can be used to control the impact of a modification, which is essential when it should preserve the initial behavior an application.

The AS opens important perspectives for design refactorings. To begin with, if the mapping between object-oriented languages and the AS syntax is possible, then we will be able propose language-independent refactorings. Moreover, since it proposes an abstract syntax tree modeling the contents of methods, we are able to analyze the contents of methods and consequently, to fully express refactorings pre and postconditions. We can easily determine, for instance, in which methods a given attribute is referenced. Finally, and maybe the most important perspective, the UML will be able to fully represent an application in one single abstract instance of the same modeling constructs. This is an essential issue, since refactorings only make sense when restructuring an existing application.

Also, the AS represents a real gain for refactoring implementation, not merely because it can directly manipulate UML constructs, but also because of the possibility of combining it with OCL rules to write pre and postconditions. More precisely, as refactorings must preserve the behavior of the modified application, they cannot be applied blindly: every refactoring ought to verify a set of conditions before the transformation is carried out.

Since the goal of this paper is not to present a comprehensive list of possible refactorings (which can be found in [12]), but to illustrate the use of the AS for their implementation, only two refactorings are presented below. The first one is a simple transformation, used to move up an attribute inside an hierarchy of classes. The second transformation is noticeably more complex than the first one: it moves an operation from a class to another one. In this particular case, the code of the concerned operation must be analyzed, to determine whether the transformation can be triggered.

Each refactoring is defined by a triad of: precondition, actions, and postconditions. The actions describe how the transformation accomplishes its intent while the pre- and postconditions are used to verify whether the transformation can be applied and if its application reaches its goals. Since our approach concerns the UML, we naturally use the OCL to specify pre- and postconditions. Also since the AS does not have an official surface language, we have adopted an "OCL-like" version of it in our examples.

The transformations presented here manipulate instances of concepts from the UML and the AS metamodels and require some precise knowledge of these metamodels.

3.1.1. Attribute generalization

The first transformation presented here is the generalization of equivalent attributes. An attribute belongs to a classifier, which is its *owner*, and has *siblings*, the children of its owner. In addition to the equivalence, which must be satisfied for exactly one attribute of each sibling, two other preconditions should be satisfied. First, private attributes cannot be moved, since they are not visible outside the scope of the owner and are not inherited. Second, the owner must have exactly one parent. The attribute generalization refactoring is expressed as follows:

: Attribute Generalization

```
Attribute:: generalize
pre:
        self . visibility  <>#private and
        self .owner.parent.size = 1 and
        self .owner.parent.children→forAll(aClass|
                aClass.feature→exists(a| a.isBasicEquivalentTo(self)))
actions:
        aList := self .owner.parent.children→collect(aClass|
                aClass.feature)→select(a| a.isBasicEquivalentTo(self))
        self .owner.parent.addFeature(self.copy)
        aList→forAll(each|each.delete)
post:
        self@pre.owner.parent.feature→exists(a|
                a.isBasicEquivalentTo(self))) and
        not self@pre.owner.parent.children→forAll(aClass|
                aClass.feature→exists(a| a.isBasicEquivalentTo(self)))
```

The specification can be explained as follows. The preconditions ensure that the attribute is not private, that the class owning the attribute has exactly one superclass, and that for all siblings of the class owning the attribute, there exists a feature which is equivalent to that attribute. In the actions part, the features of all subclasses are collected (including the owner of the attribute) and the equivalent ones are selected. Then, a copy of the attribute is added to the superclass and all selected features are deleted. The postconditions ensure that for the superclass of the previous owner class (self@pre.owner), there will exist a feature which is equivalent to the concerned attribute, and that the siblings of the previous owner class will not own an equivalent feature.

An OCL expert might rightfully notice that the operation *children* is defined neither in the OCL documentation nor as an additional operation in the UML metamodel. We have defined it symmetrically to the *parent* operation, defined for classifiers.

3.1.2. Move operation to classifier

This transformation moves an operation from a source classifier to a target one and creates a *forwarder* (see below) operation in the target. The constraints required by this transformation are rather complex. Initially, it implies the existence of an association, possibly inherited, between both classifiers. This association must be binary and its association ends must be both navigable, instance level and have a multiplicity of 1.

Although this transfer could be applied to any operation, some other constraints were specified, in order to keep it coherent. The body of the concerned operation should not directly access attributes and should only navigate through an association to the target classifier. After the transformation, the actions within the body of the moved operation that used to refer to "self" shall now refer to the source object (be it passed as a first parameter or found by navigating the association backward), and actions that used to refer to the target recurring expression shall now refer to "self" instead.

: Move Operation

```
Operation::moveTo(class: Classifier)
pre:
   class .allOperations→select(each|each.matchesSignature(self))→isEmpty()
   and let opositeAEs =
   self .owner.allAssociationEnds→
            select(each:AssociationEnd|each.isNavigable = #true
                     and each.targetScope=#instance and each.multiplicity.max = 1
                     and each.multiplicity .min = 1).association→
            select(each:Association|each.connection→size = 2).allConnections→
            select(each:AssociationEnd|each.isNavigable = #true
                     and each.targetScope=#instance and each.multiplicity.max = 1
                     and each.multiplicity .min = 1 and each.type = class) in
   opositeAEs→size() = 1 and
   self .proccdurc.allNcstcdActions()→
            forAll(a| a.oclIsKindOf(AttributeAction)
     implies (a.oclAsType(AttributeAction).attribute. visibility = #public))
actions:
   source = self .owner
   self .setOwner(class)
   oae := opositeAEs→first .association .connection→excluding(opositeAEs)
   self .allNestedActions()→select(a| a.oclIsKindOf(ReadSelfAction))→
            forAll(rsa| rl := ReadLinkObjectAction.new;
                     rl .setResult(rsa . result ); rl .setObject(rsa.object);
                     rl .setEnd(opositeAEs→first);rsa . delete)
   ca := callAction .new; ca.setOperation(self)
   newOp := Operation.new;newOp.setOwner(source)
   newOp.setSignature(self.signature .copy);newOp.addAction(ca)
post:
   let source = self@pre.owner in
   let newOp = source.allOperations→
            intersection (source.allOperations@pre)→first in
   newOp.matchesSignature(self)
   newOp.procedure.action.forwardsTo(self)
   self .procedure = self.procedure@pre
   self .allNestedActions()→collect(a : Action | a.outputPin)→
            forAll(o:OutPutPin|o.action@pre.oclIsKindOf(ReadSelfAction)
            implies (o.action.oclIsKindOf(ReadLinkObjectAction) and
            o.action.oclAsType(ReadLinkObjectAction).end.type = source)
```

The preconditions first ensure that the set of all operations owned by the target classifier does not contain any operation having the same signature as the concerned operation. Then, a set of opposite association ends is built in three steps: we select the association ends that are navigable, instance level and have a multiplicity of exactly one and create a list of associations; among these associations, we select the binary ones and create a list with all association ends connected to these associations; among these association ends, we select those that are also navigable, instance level, having a multiplicity of one and whose connected classifier is the target class. The size of the resulting set should be one (note that OCL, set operations like collect, select, etc., always return sets of elements, since nested sets are not accepted). Finally, for all actions contained by the operation, if the kind of an action is 'attribute action' (there exists read and write attribute actions), then the visibility of the read or written attribute should be public.

In the actions part, we first store the class owning the operation, change the owner of the last and find the association end used to reference the source class. Then, in a set composed of all actions contained by the operation, we select the references to 'self', the read self actions, and replace these actions with links to the source classifier, the read link object actions. The replacement is completed when the references to result and object are set in the new action, and the old action is deleted. Finally, a call action is created, and the called action is set to the moved operation. A new operation is created, it is added to the source class, its signature is set and its only action is a call action.

In the postconditions part, we first make an intersection between the set of operations that the source class owns and the set of operations it owned before the transformation. This intersection is the new operation. The new operation should have the same signature as the moved operation, and its only action should be a forwarder. The condition then collects a list of output pins of all its nested actions and verifies that, for all actions linked to these output pins, if the action was a reference to self, then the actual action should be a read link object action, whose association end is connected to the source class.

A forwarder method is a method that has one action only, a call to another operation. Since the body of a method is set of AS actions, this set must contain exactly one action, whose kind is Call Action. This operation is defined as follows:

: Forwards To

```
Action::forwardsTo(op:Operation): Boolean
post:
result = let actions = self.allNestedActions in
        actions→size = 1 and
        actions→ first .oclIsKindOf(CallAction) and
        actions→ first .operation = op
```

3.2. Design patterns

Another interesting use for AS is the application of the solution proposed by a Design Pattern, i.e. the specification of the proposed terminology and structure of a pattern in a particular context (called instance or occurrence of a pattern). In other words, we foresee the application of a pattern as a sequence of transformation steps that are applied to an initial situation in order to reach a final situation, an explicit occurrence of a pattern.

This approach is not, and does not intend to be, universal since only a few patterns mention an existing situation to which the pattern could be applied (see [13] for further discussion on this topic). In fact, our intent is to provide designers with metaprogramming facilities, so they are able to define (and apply) their own variants of known patterns. The limits of this approach, such as pattern and trade-offs representation in UML, are discussed in [14].

As an example of design pattern application, we present below a transformation function that applies the Proxy pattern. The main goal of this pattern is to provide a placeholder for another object, called *Real Subject*, to control access to it. It is used, for instance, to defer the cost of creation of an expensive object until it is actually needed:

```
Class::addProxy
pre:
  let classnames = self.package.allClasses→collect(each : Class | each.name) in
  classnames→excludes(self.name+'Proxy') and
  classnames→excludes('Real'+self.name)
actions:
  str := self.name
  self.name := str.concat('Proxy')
  super := self.package.addClass(str, self.allSuperTypes(),{}→including(self))
  real := self.package.addClass('Real'.concat(str),{}→including(super),{})
  ass := self.addAssociationTo('realSubject',real)
  self.operations→forAll(op : Operation | op.moveTo(real))
```

This function uses three other functions, that actually happen to be *refactorings*. The first function, *addClass()*, adds a new class to a package, and inserts it between a set of super-classes and a set of subclasses. The second, *addAssociationTo()*, creates an association between two classes. The third, *moveTo()*, presented in the previous section, moves a method to another class and creates a forwarder method in the original class.

This transformation should be applied to a class playing the *role* of real subject.[3] Its application proceeds as follows:

(1) Add the 'Proxy' suffix to the class name.
(2) Insert a super-class between the class and its super-classes.
(3) Create the *real subject* class.
(4) Add an association between the *real subject* and the *proxy*.
(5) Move every method owned by the *proxy* class to the *real subject* and create a forwarder method to it (move methods).

As we have explained before, this is only one of the many implementation variants of the Proxy pattern. This implementation is not complete, since it does not create the *load()* method, which should create the *real subject* when it is requested. However, it can help designers to avoid some implementation burden, particularly when creating forwarder methods.

3.3. Aspect weaving

Finally, we would like to show how AS can support the task of developing applications that contain multiple aspects. Aspects (or concerns) [15,16] refer to non-functional requirements that have a global impact on the implementation. The approach used in dealing with this is to separate these aspects from the conceptual design, and to introduce them into the system only during the final coding phase. In many cases, the merging of aspects is handled by an automated tool. In our example, we attempt to show how aspects can be woven at the design level through model transformation [17], using the AS to write the transformation rules.

The class diagram in Fig. 2 illustrates a model of a bank personal-finances information-management system. In the original system, the accounting information was stored in a relational database and each class marked with the "persistent" stereotype can be related to a given table in the database.

The aim of this re-engineering project is to develop a distributed object-oriented version of the user front-end to support new online access for its customers. One of the non-functional requirements is to map these "persistent" objects to the instance data stored in the relational database.

The task involves writing a set of proxy classes that hide the database dependency, as well as the database query commands. An example of the required transformation is illustrated by the model in Fig. 3. In this reference template, the instance variable access methods are generated automatically and database specific instructions are embedded to perform the necessary data access.

Since the re-engineering is carried out in an incremental manner, there is a problem with concurrent access to the database during write-back commits. The new application must cooperate with older software to ensure data coherence. A provisional solution is to implement a single-ended data coherence check on the new software. This uses a timestamp to test if data has been modified by other external programs. If data has been modified since the last access, all commit operations will be rolled back, thus preserving data coherence without having to modify old software not involved in this incremental rewrite. Fig. 4 shows the template transformation required. It adds a flag to cache the timestamp and access

[3] Patterns are defined in terms of roles, which are played by one or more classes in its occurrences.

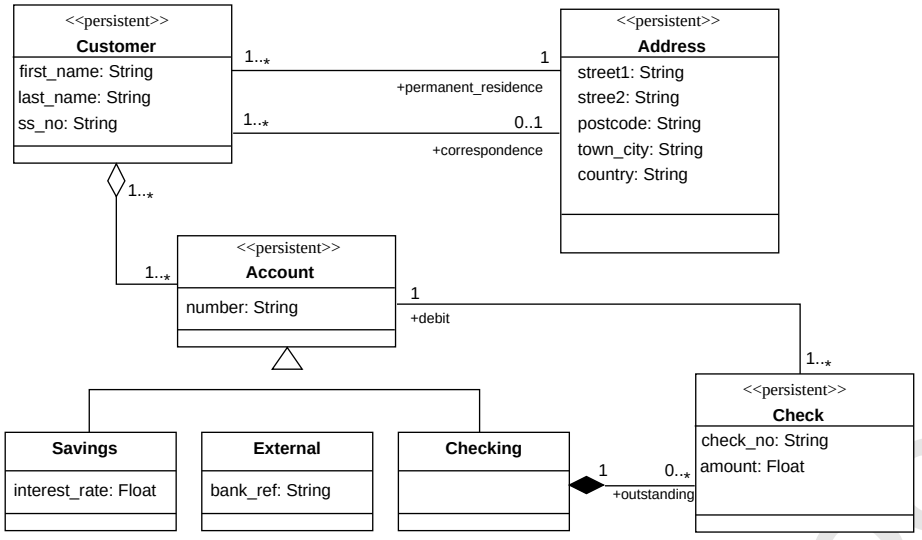

Fig. 2. Information management system for personal finance.

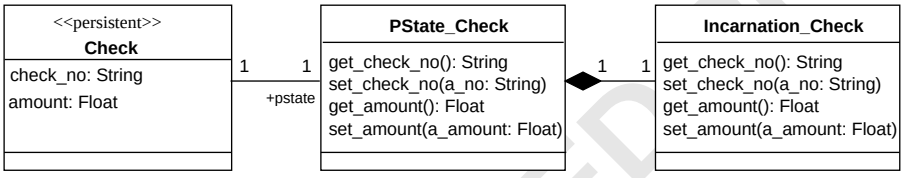

Fig. 3. Persistence proxies and access methods.

methods will be wrapped by timestamp-checking code.

The metaprogram needed to generate the proxy classes of Figs. 3 and 4 is composed of several operations. The first one is defined in the context of a Namespace (i.e. the container of UML modeling elements). It selects all classes that are stereotyped 'persistent' and applies the *implementPersistent()* transformation:

```
Namespace:: implementPersistentClasses
actions:
  self.allClasses→select(each : Class | each.stereotype→notEmpty)→
    select(each : Class | each.stereotype→first.name = 'persistent')→
      forAll(each : Class | each.implementPersistent)
```

The *implementPersistent()* operation is defined in the context of a Class. This operation will first create two classes, *state* and *incarnation*, and then creates, in these classes, the access methods to its own stereotyped attributes. This operation is defined as follows:

```
Class:: implementPersistent
actions:
  pstate := self.package.addClass('PState '.concat(pclass.name),{},{})
  pstate.addOperation('Load'); pstate.addOperation('Save')
  self.addAssociationTo(pstate, 1, 1)
  incarnation := self.package.addClass('Incarnation_'.concat(pclass.name),{},{})
  pstate.addCompositeAssociationTo(incarnation, 1, 1)
  attrs := self.allAttributes→select(a : Attribute| a.stereotype→notEmpty)
  attrs→select(a : Attribute | a.stereotype→first.name = 'getset')→
      forAll(a : Attribute |
          pstate.createSetterTo(a); pstate.createGetterTo(a)
          incarnation.createSetterTo(a); incarnation.createGetterTo(a))
  attrs→select(a : Attribute | a.stereotype→first.name = 'get')→
      forAll(a : Attribute |
          incarnation.createGetterTo(a); pstate.createGetterTo(a))
  attrs→select(a : Attribute | a.stereotype→first.name = 'set')→
      forAll(a : Attribute |
          pstate.createSetterTo(a); incarnation.createSetterTo(a))
```

The creation of the access methods is implemented by the *createSetterTo()* and *createGetterTo()* operations. They are both defined in the Class context and implement a similar operation. They take an Attribute as parameter and create a Method for setting or getting its value. These

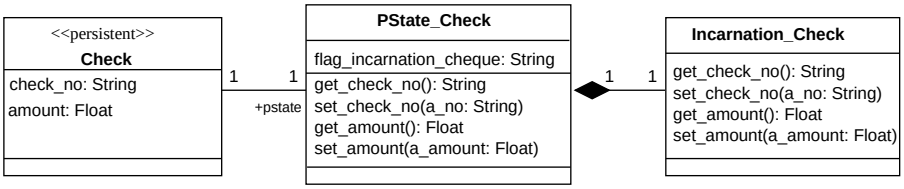

Fig. 4. Timestamp cache flag for concurrent data coherence.

operations use two other operations, *createMethod()* and *createParameter()*, which are explained above:

```
Class::createSetterTo(att : Attribute)
actions:
  newMethod := self.createMethod('set_'.concat(att.name))
  newMethod.createParameter('a_'.concat(attrib_name), att.type, 'in')
```

```
Class::createGetterTo(att : Attribute)
actions:
  newMethod := self.createMethod('get_'.concat(att.name))
  newMethod.createParameter('a_'.concat(attrib_name), att.type, 'out')
```

The *createMethod()* operation is also defined in the Class context. Its role is to create a new Method from a string and to add it to the Class:

```
Class::createMethod(str : String)
actions:
        newMethod := Method.new
        newMethod.name := str
        self.addMethod(newMethod)
        result := newMethod
```

Finally, the *createParameter()* operation creates a new parameter and adds it to a Method, which is the context of this operation:

```
Method::createParameter
(name : String, type : Class, direction : String)
actions:
        newParameter := Parameter.new
        newParameter.name := name
        newParameter.setType(type)
        newParameter.setDirection(direction)
        self.addParameter(newParameter)
        result := newParameter
```

The attractiveness of this example is not immediately evident. Let us consider a different implementation for the persistent proxy of Fig. 3. In the case where there are composite persistent objects, it is possible to use a single persistent state proxy for a composite object and all its components (see Fig. 5). Through the use of metaprogramming, it is now possible to consider these different implementation aspects independently from the concurrency implementation. It enables the designer to conceptualize the modifications in a manageable manner. Making changes to a model by hand as a result of a change in an implementation decision is not a viable alternative as it is laborious and prone to error.

Therefore, it can be seen that metaprogramming using the AS can facilitate implementation changes at a higher abstraction level. It also leverages the execution machine for the AS by using it to perform the model transformation.

4. Related work

Transformations of a UML model can be classified into two types according to the objective in applying them: model (or schema) manipulation and code generation. In the former, the syntax is preserved, i.e. the source and the target models are represented by the same language. This is the approach used, for instance, to add particular details to UML models. In the latter, the syntax of the target model is a programming language.

While code generation generally concerns the whole UML model, model manipulations are often smaller and concern fewer modeling elements. In both approaches, a transformation can be divided into two different parts: the *selection* (or filter) of the elements concerned and the *actions* performing the transformation itself. A complementary part can be added to the later: the validation of the target model.

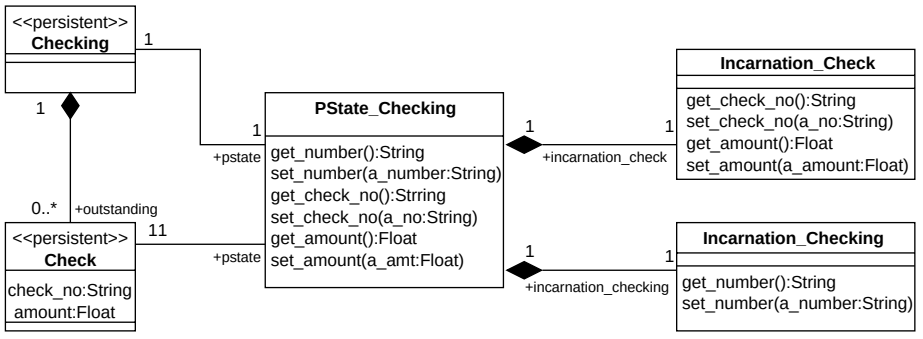

Fig. 5. Implementation template for shared proxy.

Due to the current lack of semantics when specifying behavior in UML, commercial UML tools often propose metaprogramming languages for both model manipulation and code generation, allowing designers to specify their own semantics using UML-specific modeling elements, such as Stereotypes or Tagged values. This is the case, for instance, of Object Domain and Softeam's Objecteering, which use Python and J, a "Java-like" language [18], respectively. This is also the case of Rational Rose and of Rhapsody from Ilogix, which use Visual Basic.

The use of well-known object-oriented languages for model manipulation has at least one clear advantage; they discharge designers from learning yet another tool-dedicated language. However, the choice of such a language may also be a weakness: these languages are not as well adapted to code generation as other techniques, such as template-based code generation [19], and do not offer model-specific facilities, such as navigation or pattern recognition for filtering model elements.

The emergence of the XML [20] and its related standards brought an alternative approach to model transformation. Indeed, one may use the XMI [21] rules to represent a UML model in the XML format and use a XLST engine to transform it into another UML model or its source code in different languages. However, this approach is more attractive for code generation than for model transformation, since it performs only purely syntactic transformation and does not take into account the constraints of the UML static semantics. Moreover, this approach is difficult to use when a model must be significantly modified.

The AS, used as a metaprogramming language, has the same weaknesses as the languages proposed by commercial tools: it is not conceived for code generation or for selecting model elements. The use of the OCL to filter model elements (and not only for specifying pre and postconditions), may be an interesting alternative. In this case, OCL would play the same role as XPath in the XSLT context. Concerning the inadequacy of the AS for code generation, this is not actually a problem, since a tool that implements the AS in the design level does not need to "generate" the code, it only adds a concrete syntax to an abstract graph.

5. Conclusion

The AS extends the UML with new elements to precisely specify behaviors. Its advantages over ad hoc notations are its well-founded semantics, and its level of abstraction that nicely fits between UML high level specifications and low level implementation concepts.

Moreover, since the UML metamodel itself is a UML model, the AS can be used as a powerful mechanism for modeling and executing design time model transformations. This particularity opens new perspectives for designers thanks to its perfect integration with the UML: all the features of the UML, such as constraints (pre or

postconditions, invariants), refinements or traces can be applied within the AS.

The use of the AS may bring some possible changes to the traditional software development process and play a major role in realizing the new OMG vision of a Model Driven Architecture (MDA). The AS is an important step towards the use of UML in an effective development environment, since it offers the possibility of animating early design models and evolving or refining them until their implementation. The development approach we propose here starts with an early design model, created by the designers from an analysis model. This model is completely independent from the implementation environment, it assumes an "Ideal World", where the processing power and the memory are infinite, there are no system crashes, no transmission errors, no database conflicts, etc. Since this model contains AS statements, it can be animated by the AS machine and validated. Once this early validation is completed, the designers can add some platform-specific aspects to the design model (database access, distribution), apply design patterns and restructure the model using design refactorings.

We can then foresee a new dimension for the distribution of work in development teams, with a few *metadesigners* being responsible for translating the mechanistic part of a company's design know-how into *Reusable Transformation Components*, while most other designers concentrate on making intelligent design decisions and automatically applying the corresponding transformations.

An implementation conforming to the current version of the AS specification is in development in UMLAUT,[4] a freely available UML modeling tool. The complete integration of the AS and the UML in UMLAUT provides an excellent research platform for the implementation of design refactorings.

[4] http://www.irisa.fr/UMLAUT/

References

[1] Object Management Group, Action semantics for the uml rfp, ad/98-11-01, 1998.

[2] The Action Semantics Consortium, Action semantics for the uml, omg ad/2001-03-01, March 2001.

[3] IUT-T, Recommendation z.109 (11/99)—SDL combined with UML, 1999.

[4] W.F. Opdyke, Refactoring object-oriented frameworks, Ph.D. Thesis, University of Illinois, Urbana-Champaign, Technical Report UIUCDCS-R-92-1759, 1992.

[5] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Professional Computing Series, Addison-Wesley, Reading, MA, 1995.

[6] R. Keller, R. Schauer, Design components: Towards software composition at the design level, in: Proceedings of the 20th International Conference on Software Engineering, IEEE Computer Society Press, Silver Spring, MD, April 1998, pp. 302–311.

[7] B. Meyer, Object-Oriented Software Construction, Prentice-Hall, Englewood Cliffs, NJ, 1988.

[8] A. Kleppe, J. Warmer, S. Cook, Informal formality? the object constraint language and its application in the UML metamodel, in: J. Bézivin, P.A. Muller (Eds.), The Unified Modeling Language, UML'98—Beyond the Notation, First International Workshop, Mulhouse, France, June 1998, pp. 127–136.

[9] Kennedy-Carter, Executable UML (xuml), http://www.kc.com/html/xuml.html.

[10] Projtech-Technology, Executable UML, http://www.projtech.com/pubs/xuml.html.

[11] The Action Semantics Consortium, Updated joint initial submission against the action semantics for uml rfp, 2000.

[12] G. Sunyé, D. Pollet, Y. LeTraon, J.-M. Jézéquel, Refactoring UML models, in: Proceedings of UML 2001, Lecture Notes in Computer Science, Springer, Berlin, 2001.

[13] M. Cinnéide, P. Nixon, A methodology for the automated introduction of design patterns, in: International Conference on Software Maintenance, Oxford, 1999.

[14] G. Sunyé, A. Le Guennec, J.-M. Jézéquel, Design pattern application in UML, in: E. Bertino (Ed.), ECOOP'2000 Proceedings, Lecture Notes in Computer Science, Vol. 1850, Springer, Berlin, June 2000, pp. 44–62.

[15] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, Aspect-oriented programming, in: M. Akşit, S. Matsuoka (Eds.), ECOOP '97—Object-Oriented Programming 11th European Conference, Jyväskylä, Finland, Lecture Notes in Computer Science, Vol. 1241, Springer, New York, June 1997, pp. 220–242.

[16] P. Tarr, H. Ossher, W. Harrison, N degrees of separation: Multi-dimensional separation of concerns, in: ICSE'99, Los Angeles, CA, 1999.

[17] W.M. Ho, F. Pennaneac'h, N. Plouzeau, Umlaut: A framework for weaving UML-based aspect-oriented designs, in: Technology of Object-Oriented Languages and

Systems (TOOLS Europe), Vol. 33, IEEE Computer Society, June 2000, pp. 324–334.

[18] Softeam, UML Profiles and the J Language: Totally control your application development using UML, http://www.softeam.fr/us/pdf/uml_profiles.pdf, 1999.

[19] S. Srinivasan, Advanced Perl Programming — Template-Driven Code Generation, O'Reilly and Associates, Sebastapol, CA, 1997 (Chapter 17).

[20] T. Bray, J. Paoli, C. Sperberg-McQuee, Extensible Markup Language (XML) 1.0—W3C recommendation, February 1998, http://www.w3.org/TR/1998/REC-xml-19980210.html.

[21] OMG, OMG XML metadata interchange (XMI) specification, version 1.0, June 2000, http://www.omg.org/.