

Using UML Action Semantics for Executable Modeling and Beyond

Gerson Sunyé, François Pennaneac’h, Wai-Ming Ho, Alain Le Guennec, and Jean-Marc Jézéquel

IRISA, Campus de Beaulieu, F-35042 Rennes Cedex, France
email: sunye,pennanea,waimingh,aleguenn,jezequel@irisa.fr

Abstract. The UML lacks precise and formal foundations for several constructs such as transition guards or method bodies, for which it resorts to semantic loopholes in the form of “uninterpreted” expressions. The Action Semantics proposal aims at filling this gap by providing both a metamodel integrated into the UML metamodel, and a model of execution for these statements. As a future OMG standard, the Action Semantics eases the move to tool interoperability, and allows for executable modeling and simulation. We explore in this paper a specificity of the Action Semantics: its applicability to the UML metamodel, itself a UML model. We show how this approach paves the way for powerful metaprogramming capabilities such as refactoring, aspect weaving, application of design patterns or round-trip engineering. Furthermore, the overhead for designers is minimal, as mappings from usual object-oriented languages to the Action Semantics will be standardized. We focus on an approach for expressing manipulations on UML models with the upcoming Action Semantics. We illustrate this approach by various examples of model transformations.

1 Introduction

The Unified Modeling Language provides modeling foundations for both the structural and behavioral parts of a system. The designer is offered nine views which are like projections of a whole multi-dimensional system onto separate planes, some of them orthogonal. This provides an interesting separation of concerns, covering four main dimensions of software modeling: functional (use cases diagrams express the requirements), static (class diagrams), dynamic (state-charts, sequence diagrams for the specification of behavioral aspects) and physical (implementation diagrams).

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. The Action Semantics (AS) proposal [4], currently being standardized at the Object Management Group (OMG), aims at filling this gap by providing both a metamodel integrated into the UML metamodel, and a model of execution for these statements. Along the lines of what already exists in the IUT-T Specification and

Description Language (SDL) community [11], the integration of the Action Semantics 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.

The contribution of this paper is twofold. First, it gives an update on the forthcoming UML/AS standard, for which we contributed some of the precise semantic aspects. Second, relying on the fact that the UML metamodel is itself a UML model, we show how the Action Semantics can be used at the metamodel level to help the OO designer carry on activities such as behavior-preserving transformations [16] (see § 3.1), design pattern application [5] (§ 3.2) and design aspects weaving [12] (§ 3.3). But before extending the description of the Action Semantics, let us dispel some possible misunderstanding here. Our intention is not to propose yet another approach for model transformation, pattern application or refactoring. What we claim here, is that the Action Semantics 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 in both cases, as a design level and 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 Joe-the-OO-designer cannot be expected to write complex metaprograms from scratch, in order to develop 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/Action Semantics 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 [15] applied at the metamodel level. Furthermore, since the Action Semantics is fully integrated in the UML metamodel, it can be combined with rules written in the Object Constraint Language (OCL), in order to verify if a transformation (or a set of transformations) may be applied to a given context.

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 RTC, while most other

designers concentrate on making intelligent design decisions and automatically applying the corresponding transformations.

The rest of the paper is structured as follows. Section 2 recalls some fundamental aspects of the UML and gives a quick update on the Action Semantics proposal currently under submission for standardization at the OMG. Section 3 shows the interest of using the Action Semantics at the metamodel level for specifying and programming model transformations in several contexts. Section 4 analyses the impact of this approach on OO development processes, and presents our experience with using it on several case studies. Section 5 describes related work.

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 (via the nine views, cited in the above introduction). 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 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 (expressed in OCL), which are logical constraints on the elements in a model. For instance, there should not be any inheritance cycle and 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, breaking 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 Action Semantics 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 [7], 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 [13] or BridgePoint [17] action languages¹. But contrary to its predecessors, it is intended that the Action Semantics become an OMG standard, and a common base for all the existing and to-come action languages (mappings from existing languages to Action Semantics 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, not to say impossible, 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 make it incomplete and imprecise; trying to complete it often obliges the introduction of some *how* issues for describing the most complex behavior.

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 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, 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

¹ all the major vendors providing an action language are in the list of submitters.

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 Action Semantics Proposal Submitted to the OMG

The Action Semantics proposal is based upon three abstractions:

- A metamodel: it extends the current metamodel. The Action Semantics 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).

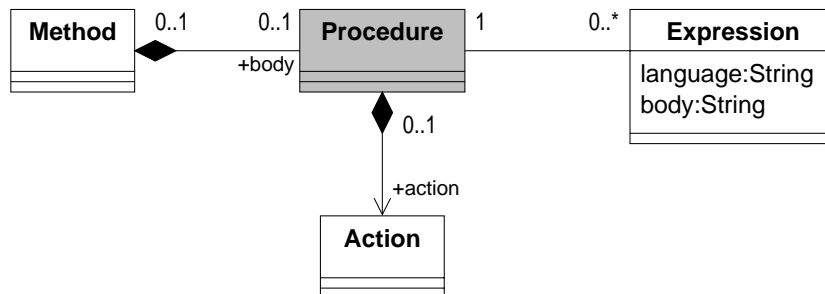


Fig. 1. Action Semantics immersion into the UML

2.4 Surface Language

The Action Semantics 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 Action Semantics is certainly not to define a new notation, or to force the use of a particular existing one. The Action Semantics was, however, conceived to allow an easy mapping of classical languages such as Java, C++ or SDL². 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 Action Semantics 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. Since the Action Semantics does not have an official surface language, we adopt an “OCL-based” version in our examples.

3.1 Refactorings

Refactorings [16] are behavior-preserving transformations that are used to restructure the source code of applications and thus make the code more readable and reusable. They help programmers to manipulate common language constructs, such as class, method and variable, of an existing application, without modifying the way it works. Since we believe that refactorings are an essential artifact for software development, we are interested in bringing them to the design level by means of a UML tool.

The implementation of refactorings in UML is an interesting task, since *design* refactorings – as opposed to *code* refactorings – should work with several design constructs shared among several views of a model. Furthermore, while code refactorings manipulate source code (text files), design refactorings should manipulate an abstract syntax tree.

This difference simplifies the implementation of some refactorings, for instance renaming elements, since an element is unique inside the abstract syntax tree. However, other refactorings (namely moving features) are more difficult to implement because we must take into account other constructs, such as OCL constraints and state charts.

The Action Semantics represents a real gain for refactoring implementation, not merely because it directly manipulates UML constructs, but also because of the possibility of combining it with OCL rules to write pre and post-conditions. More precisely, as refactorings must preserve the behavior of the modified application, they cannot be widely applied. Thus, every refactoring ought to verify a set of conditions before the transformation is carried out.

Below we present an example of a simple refactoring, the removal of a class from a package. This refactoring can only be applied if some conditions are

² Some of these mappings are illustrated in the AS specification document [4].

verified: the class is not referenced by any other class and it has no subclasses (or no features). These conditions and the transformation itself are defined in the Action Semantics as follows:

```

Package::removeClass(class: Class)
pre:
  self . allClasses→ includes(class) and
  class . classReferences→ isEmpty and
  ( class . subclasses→ isEmpty or class.features→ isEmpty)
actions:
  let aCollection := class . allSuperTypes
  class . allSubTypes→ forAll(sub : Class |
    aCollection→ forAll(sup : Class | sub.addSuperClass(sup)))
  class . delete
post:
  self . allClasses→ excludes(class) and
  class @pre.allSubTypes→ forAll(each : Class |
    each.allSuperTypes.includesAll(class@pre.allSuperTypes)) and
  class @pre.features→ forAll(feat : Feature | feat .owner = nil)

```

This refactoring calls two functions that are not present in the Action Semantics, *delete* and *addSuperClass()*, and should be defined elsewhere: the role of the former is to make the delete effective, by deleting the features, associations and generalization links owned by the class. The latter feature is described below. Its purpose is to create a generalization link between two classes:

```

Class :: addSuperClass(class: Class)
pre:
  self . allSuperTypes→ excludes(class)
actions:
  let gen := Generalization.new
  gen.addLinkTo(parent, class)
  gen.addLinkTo(child, self)
post:
  self . allSuperTypes()→ includes(class)

```

3.2 Design Patterns

Another interesting use for Action Semantics 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 envisage 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 [1] 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 [20].

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:
  let name := self.name
  let self.name := name.concat('Proxy')
  let super :=
    self.package.addClass(name,self.allSuperTypes(),{} →including(self))
  let real :=
    self.package.addClass('Real'.concat(name),{} →including(super),{})
  let ass := self.addAssociationTo('realSubject',real)
  self.operations →forAll(op : Operation | op.moveTo(real))

```

This function uses three others (actually, refactorings), that will not be precisely described here. They are however somewhat similar to the *removeClass()* function presented above. 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()*, moves a method to another class and creates a “forwarder” method in the original class.

This transformation should be applied to a class that is to play the *role* of real subject³. 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 Patterns are defined in terms of roles, which are played by one or more classes in its occurrences

3.3 Aspect Weaving

Finally, we would like to show how Action Semantics can support the task of developing applications that contain multiple aspects. Aspects (or concerns) [14,21] 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 weaved as early as the design level through model transformation [9], using the Action Semantics to write the transformation rules.

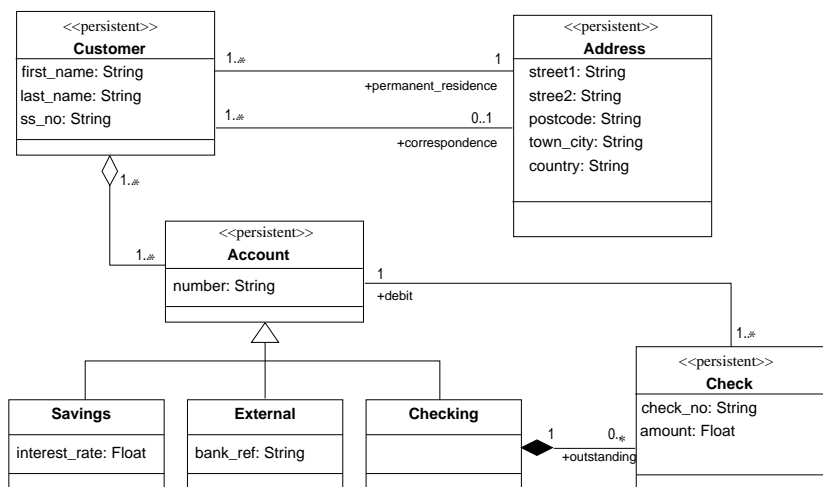


Fig. 2. Information Management System for Personal Finance

The class diagram in Fig. 2 illustrates a model of a bank’s 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.

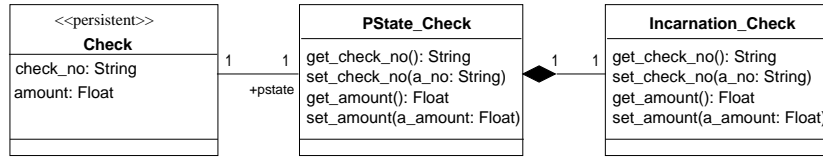


Fig. 3. Persistence proxies and access methods

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 methods will be wrapped by timestamp-checking code.

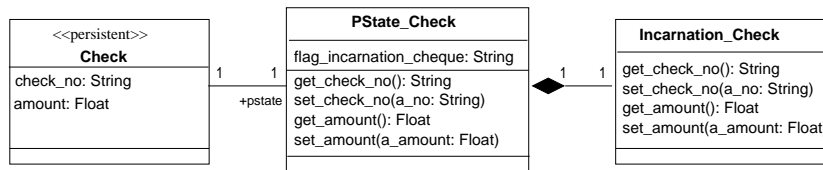


Fig. 4. Timestamp cache flag for concurrent data coherence

The metaprogram needed to generate the proxy classes of figures 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 sends them the *implementPersistent()* message:

```

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:

```
let pstate :=
  self.package.addClass('PState_' . concat(pclass.name), {}, {})
pstate.addOperation('Load'); pstate.addOperation('Save')
self.addAssociationTo(pstate, 1, 1)
let incarnation :=
  self.package.addClass('Incarnation_' . concat(pclass.name), {}, {})
pstate.addCompositeAssociationTo(incarnation, 1, 1)
let 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 operations use two other operations, *createMethod()* and *createParameter()*, which are explained above:

Class :: createSetterTo(att : Attribute)

actions:

```
let newMethod := self.createMethod('set_' . concat(att.name))
newMethod.createParameter('a_' . concat(attrib.name), att.type, 'in')
```

Class :: createGetterTo(att : Attribute)

actions:

```
let 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(name : String)

actions:

```
let newMethod := Method.new
let newMethod.name := name
self.addMethod(newMethod)
let 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:

```

let newParameter := Parameter.new
let newParameter.name := name
newParameter.setType(type)
newParameter.setDirection(direction)
self.addParameter(newParameter)
let result := newParameter

```

The attractiveness of this approach 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.

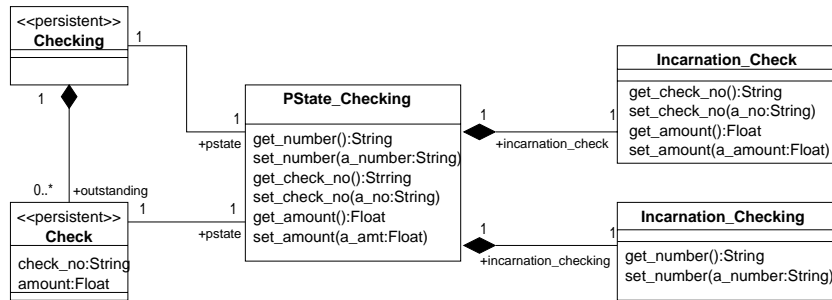


Fig. 5. Implementation template for shared proxy

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

4 Impact in the development approach

The Action Semantics brings some possible changes to the traditional software development process. In other words, the Action Semantics 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 of 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 Action Semantics statements, it can be animated by the Action Semantics machine and validated. Once the validation is finished, the designers can add some environment-specific aspects to the design model (database access, distribution), apply design patterns and restructure the model using design refactorings.

A significant part of this information addition can be automated using the metaprogramming techniques we presented above (§ 3). More precisely, the designers are able to automatically apply a set of predefined transformations, defined by a special software developer, who we call the *metadesigner*. We see the metadesigners as experienced developers, who handle essential knowledge about the implementation environment. Their role is to use their knowledge to define and write model transformations and make them available to designers. The approach ends when the final design model is reached. After that, the implementation code can be generated.

4.1 Organizing Things

UML proposes an interesting extension mechanism called *Profiles*, allowing designers to expand the current metamodel. According to the UML 1.4 documentation (page 4-3), a UML Profile is:

(...) a predefined set of Stereotypes, TaggedValues, Constraints, and notation icons that collectively specialize and tailor the UML for a specific domain or process (e.g. Unified Process profile). A profile does not extend UML by adding any new basic concepts. Instead, it provides conventions for applying and specializing standard UML to a particular environment or domain.

Profiles can be used to organize Action Semantics metaprograms, whilst some other UML concepts, such as refinements and traces, can be used to manage the evolution of metaprograms (versioning). When the “refactoring” Profile is loaded into the metamodel, some operations are added to the UML concepts. For instance, *addClass()* is added to Package, *addSuperClass()* to Class, etc.

5 Related Work

Commercial UML tools often propose metaprogramming languages in order to manipulate models. This is the case, for instance, of Softeam’s Objecteering, which uses J, a “Java-like” language [19]. This is also the case of Rational Rose and of Rhapsody from Ilogix, which use Visual Basic. The purpose of these languages is similar to the one we look for when using the Action Semantics as a metaprogramming language. However there are some notable differences:

To begin with, the Action Semantics only proposes an abstract syntax and no surface language. Thus, it is not a direct concurrent of J or Visual Basic, but it can provide a common underlying model for these languages.

Next, the navigation through the metamodel is done in a language-specific way, which does not reuse the standard OCL navigation facilities.

Finally, since there are discrepancies between the UML metamodel and the metamodels implemented by commercial tools, designers are not able to write truly universal UML transformations.

The lack of a surface language may be an annoying problem, since the choice of the surface language impacts on the practical use. While the choice of using a classic language (e.g. Java), thus possibly the same language at both the design and the meta-model level, is attractive, it may not be the most adequate. The adoption of a declarative language (e.g. OCL) may help bridge the gap between the specification of contracts for design transformations (specified with the OCL) and their implementation.

6 Conclusion

The Action Semantics provides designers with an expressive formalism for writing model transformations in an operational way, and a model for UML statement execution that fills some semantic gaps in the behavior aspects of a UML model. Moreover, since the UML metamodel itself is a UML model, the Action Semantics can be used as a powerful mechanism for metaprogramming. This particularity opens new perspectives for designers, compared to other model manipulations languages (e.g. J or VisualBasic), thanks to its perfect integration with the UML: all the features of the UML, such as constraints (pre or post-conditions, invariants), refinements or traces can be applied within the Action Semantics.

An implementation conforming to the current version of the Action Semantics specification is in development in UMLAUT⁴, a freely available UML modeling tool. The complete integration of the Action Semantics and the UML in Umlaut provides an excellent research platform for the implementation of design patterns, refactorings and aspects.

Indeed, we are presently looking for more UML-specific refactorings with the purpose of extending our existent set, which is based on existing program restructuring transformations [2,3,6,10,16,18]. The use of the OCL and the Actions Semantics for defining refactorings opens two interesting perspectives. The first perspective concern the combinations of refactorings. As D. Roberts stated in his thesis [18], pre and post conditions can be used to verify if each single refactoring, within a set of combined refactorings, is (or will be) allowed to execute, before the execution of the whole set.

The second perspective concerns the property of behavior-preservation, that should be verified after the execution of a refactoring. In a similar way to other

⁴ <http://www.irisa.fr/UMLAUT/>

efforts in this area, we intend to propose a set of basic transformations and demonstrate how they ensure behavior-preservation. We will then be able to combine them in order to define higher-level transformations. We intend to define an approach based on the notion of *refactoring region*, which is a fraction of an instance of the UML metamodel (i.e. the underlying objects that represent a modeled system) that may potentially be modified by a transformation. With the help of bisimulation techniques we will be able to verify if the original and the transformed regions are equivalent. The notion of *snapshot history* - which is present in the Actions Semantics execution engine - provides excellent support for such a comparison.

We are also working on the representation of patterns trade-offs that could help the designer to define the variant she wants to apply. An introduction to this work can be found in [8]. The analysis of the Action Semantics syntax tree will help us to verify if a pattern was correctly implemented. More precisely, our tool will be able to verify if a given implementation of a pattern respects a set of predefined OCL constraints [20].

References

1. M. Cinnide and P. Nixon. A methodology for the automated introduction of design patterns. In *International Conference on Software Maintenance*, Oxford, 1999.
2. P. Bergstein. Maintenance of object-oriented systems during structural schema evolution. *TAPOS*, 3(3):185–212, 1997.
3. E. Casais. *Managing Evolution in Object Oriented Environments: An Algorithmic Approach*. Phd thesis, University of Geneva, 1991.
4. T. A. S. Consortium. Updated joint initial submission against the action semantics for uml rfp, 2000.
5. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, Reading, MA, 1995.
6. W. Griswold. Program restructuring as an aid to software maintenance, 1991.
7. O. M. Group. Action semantics for the uml rfp, ad/98-11-01, 1998.
8. A. L. Guennec, G. Sunyé, and J.-M. Jézéquel. Precise modeling of design patterns. In *Proceedings of UML 2000*, volume 1939 of *LNCS*, pages 482–496. Springer Verlag, 2000.
9. W. Ho, F. Pennaneac’h, and N. Plouzeau. Umlaut: A framework for weaving uml-based aspect-oriented designs. In *Technology of object-oriented languages and systems (TOOLS Europe)*, volume 33, pages 324–334. IEEE Computer Society, June 2000.
10. W. Hursch. *Maintaining Consistency and Behavior of Object-Oriented Systems during Evolution*. Phd thesis, Northeastern University, June 1995.
11. IUT-T. Recommendation z.109 (11/99) - SDL combined with UML, 1999.
12. R. Keller and R. Schauer. Design components: Towards software composition at the design level. In *Proceedings of the 20th International Conference on Software Engineering*, pages 302–311. IEEE Computer Society Press, Apr. 1998.
13. Kennedy-Carter. Executable UML (xuml), <http://www.kc.com/html/xuml.html>.

14. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, New York, N.Y., June 1997.
15. B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.
16. W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, 1992.
17. Projtech-Technology. Executable UML, <http://www.projtech.com/pubs/xuml.html>.
18. D. Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois, 1999.
19. Softeam. UML Profiles and the J language: Totally control your application development using UML. In http://www.softteam.fr/us/pdf/uml_profiles.pdf, 1999.
20. G. Sunyé, A. Le Guennec, and J.-M. Jézéquel. Design pattern application in UML. In E. Bertino, editor, *ECOOP'2000 proceedings*, number 1850, pages 44–62. Lecture Notes in Computer Science, Springer Verlag, June 2000.
21. P. Tarr, H. Ossher, and W. Harrison. N degrees of separation: Multi-dimensional separation of concerns. In *ICSE'99 Los Angeles CA*, 1999.