Action Semantics for an Executable UML
COMP601 Reading Project

Thomas Feng *

April 18, 2003

Abstract

UML currently lacks a rigorously defined semantics for its models, which makes formally analyzing a model, verifying its properties, testing it and generating code from it extremely difficult. Action semantics is a newly developed language aimed at precisely describing the behavior of programs and models. It is a desirable extension to UML.

The history of action semantics, including the pros and cons of its predecessor, denotational semantics, is briefly introduced. Comparison is made on different attempts to achieve an executable UML. The basics of action semantics, including all the defined action types, is discussed in detail. From this discussion, the conclusion is drawn that, though flaws still exist, action semantics is a powerful tool to describe behavior in the level of both modelling and meta-modelling.

1 Introduction to Action Semantics

Action semantics is a newly developed language aimed at precisely describing the behavior of programs and models. It starts from modifying denotational semantics, but finally ends up with a completely new semantics, which is much more modular and extensible than its predecessor.

1.1 Denotational semantics

As a starting point of action semantics, denotational semantics is studied very thoroughly and has lots of applications. Its base, λ-calculus, is accurate and rigorously defined. Because of this, models described in denotational semantics can be easily analyzed.

However, in the contemporary view of a modeler, the deficiency of denotational semantics outweighs its merits. Its major drawback is the lack of modularity. In a description, semantic definitions are globally visible. Part of the description intertwines with another. When change has to be made, even if it is only a minor change, it cannot be confined to a small area. People have to propagate the change throughout the whole description. This process, of course, is boring, time-consuming and error-prone.

This only addresses part of the problem. λ-notation is usually complex and hardly read. For example, a simple statement "A1 then A2" in action semantics corresponds to the λ-notation

$$\lambda\varepsilon_1.\lambda\rho.\lambda\kappa.A_1\varepsilon_1\rho(\lambda\varepsilon_2.A_2\varepsilon_2\rho\kappa)$$

[Mos96]. This is good for computer processing, but not good for humans. When reading this description, the programmer has to decipher the meaning of the λ-notation, and reconstruct the underlying semantics. It turns out that this reverse approach is extremely difficult; and any modification of the description, which is based on this reverse process, is thus formidable.

*Email: thomas@email.com.cn
Homepage: http://moncs.cs.mcgill.ca/people/tfeng/

1.2 Basics of action semantics

Because denotational semantics is inextensible and obscure to humans, people have come up with several alternatives over the years, among which are abstract semantic algebras and structural operational semantics. They are important and interesting improvements of denotational semantics. A combination and mixture of denotational semantics and operational semantics is action semantics [Mos96].

When Peter D. Mosses introduced action semantics in the 1990's, it was considered as a formalism to describe programming languages. From the description written in action semantics, not only was the syntax of the programming language rigorously defined, but also its semantics. Because it is well-defined, analysis, verification, test and code-generation can be done directly from the descriptions. The results of the code-generation, of course, are compilers for specific programming languages.

An action semantics description, contrary to the one written in denotational semantics, has an informal appearance, which looks like the natural English language. It is this appearance that raises a lot of arguments and criticism. However, it is fact that this appearance does not sacrifice any formality in its semantics, because it is rigorously defined.

There are three kinds of semantic entities in action semantics: actions, data and yielders. Actions, the most important part, define the operations that can be performed on data and yielders. An execution of actions usually results in the evaluation of yielders and the manipulation/modification of data. Data are cells which store primitive data entities (i.e. **byte**, **int**, **float**, **char**) and objects (defined by classes). They can be located in the main memory, hard disk or any other type of storage. Yielders are a kind of expression whose evaluation yield data. Note that yielders only yield data but never change them. Data can only be changed by execution of actions, whose semantics, as discussed above, is well-defined.

Now comes the question: what an action semantics description of programming language looks like. In his early proposal, Peter D. Mosses separates the description into three modules: abstract syntax, semantic functions and semantic entities. Each module may comprise several files. There is no general rule on the division of a module into files, as long as the description is clear and the modularity is kept.

1.2.1 Abstract syntax

Abstract syntax defines the grammar of the language. It uses a variant of BNF. An example is given below [Mos96]:

```
module:  Abstract Syntax.  grammar:

 (*) Stmt   [[Id ":=" Expr]]
            | [["if" Expr "then" Stmts "else" Stmts]]
            | [["while" Expr "do" Stmts]].

 (*) Stmts  = <Stmt <";" Stmt>*>.

 (*) Expr   = Num | Id | [[Expr Op Expr]].

 (*) Op     = "+" | "/=".

 (*) Num    = [[digit+]].

 (*) Id     = [[letter (letter|digit)*]].

endgrammar.  closed.  endmodule:  Abstract Syntax.
```

Table 1: An example of abstract syntax

The grammar is easy to read and understand. However, notice that this description is very language specific. I.e., in Pascal, this specification may be correct; but in C, the conditional control has another appearance "if (...) {...} else {...}". This means even though the syntax and semantics of some statements are alike in another language, it is unlikely that it can reuse the existing specification. This is why people have to rewrite the whole module in order to change all these lexical symbols.

To get out of this difficulty, a recent suggestion raises the issue of separating the description into two layers. This will be discussed in the latter part of this paper.

1.2.2 Semantic functions

Semantic functions, as the most important module of a language specification, reveals one of action semantics' innovative characteristics: informal English appearance. However, semantics is rigorously defined under this appearance. Part of this merit attributes to the use of action combinators. Combinators are primitively defined in action semantics, and have their primitive semantics. For instance, combinator "`and then`" means the sequential execution of two statements; so, if we define A_1 and A_2 to be two statements, "A_1 `and then` A_2" gives the meaning of executing A_1 before A_2. The result of the execution is independent of the architecture.

```
execute [["if" E "then" S1 "else" S2]] =
  evaluate E then
  (  ( check the given truth-value and then execute S1 ) or
     ( check not the given truth-value and then execute S2 ) ).
```

Table 2: An equation from the semantic function module

The format of semantic function module is very similar to the abstract syntax module. Table 2 shows an equation extracted from this module.

Even in this simple equation, several combinators are found including **execute**, **evaluate**, **then**, **check [not] the given truth-value**, **and then** and **or**. Not necessary to explain their semantics, every one knows it because they are English verbs or phrases. The design of these combinators, though it seems to be simple and casual, is nontrivial, because they are universal, not limited to a certain kind of statements or data. They are in a higher level of abstraction than the language specification, and described in another (internal) module. This contributes to the modularity of action semantics. To cite a conclusion in [Mos96], "since the above features ensure that ASDs (Action Semantics Descriptions — by the author) have an inherent modularity, the use of explicit modules is almost redundant."

1.2.3 Semantic entities

Data types are specified in the semantic entity module. The syntax is quite like type definition in ordinary programming languages. Examples are "`value = number | truth-value`" and "`number =< integer`". Here, `truth-value` is a primitive data type. `=<` denotes a sort inclusion, which means sort `number` is included in sort `integer` with open bound.

Only data types are defined in this module, without any notation about how to evaluate or manipulate them. Actions and yielders on data are defined as semantic functions.

1.2.4 Cross reference among modules

Whenever a module uses the definition in other modules, it must explicitly import those modules. The keyword for importation is either "`includes`" or "`needs`". On the other hand, to enable another module to import itself and make part of the definition visible to it, the module must explicitly export with the keyword "`introduces`".

The scheme is similar to packages in Java, where a unit `uses` other units, and defines whatever is visible to the outside in its `interface`. This explicit declaration ensures modularity.

1.2.5 Conclusion on merits

From the above discussion, it is clear that action semantics improves semantic notation in many aspects:

- Readability.
  The readability of action semantics description comes from its informal appearance. When a reader goes through a description, he/she can easily understand what is going on without reconstructing semantic concepts from obscure formulas.

- Modularity and extensibility.
  In action semantics, semantic functions and entities are not so entangled with each other than in denotational semantics. A module imports another one only when it requires its functionality. An action is dependent only on the data entities that it performs operations on, without noticing the existence of the others. Universal combinators are primitively defined and thus can be viewed as in another separate module. In such an organization, change on the described programming language only requires proportional change on its description.

- Reusability.
  Because the description is divided into several relatively self-contained modules and the definition of each module is usually divided into multiple files, part of the existing description can be reused in another language, just to be imported by a new one. This is extremely desirable, since contemporary programming languages share a lot of popular ideas and concepts.

1.3 Action semantics and Syntax Definition Formalism (SDF)

As mentioned above, the descriptions in original action semantics are still too language-specific, because lexical symbols are directly embedded. When the language is changed, all these symbols, scattered in the description, have to be changed accordingly. Even worse, sometimes it is the overall appearance of those structures that differs. Even if the semantics is the same, migrating from one language to another results in the change of the whole description.

The way to make action semantics even more modular and reusable is to separate the definition of semantics from that of language-specific constructs. This issue is recently proposed, and nicely introduced in [Mos02].

1.3.1 The scheme of adding SDF to action semantics

The approach to the separation is by using the Syntax Definition Formalism (SDF) and view the language descriptions at a higher level. This higher level is relative to the view-point of programmers or language description authors, not to that of action semantics researchers. This is because the researchers are already at a higher level, where the language description is a concrete model of the action semantics formalism.

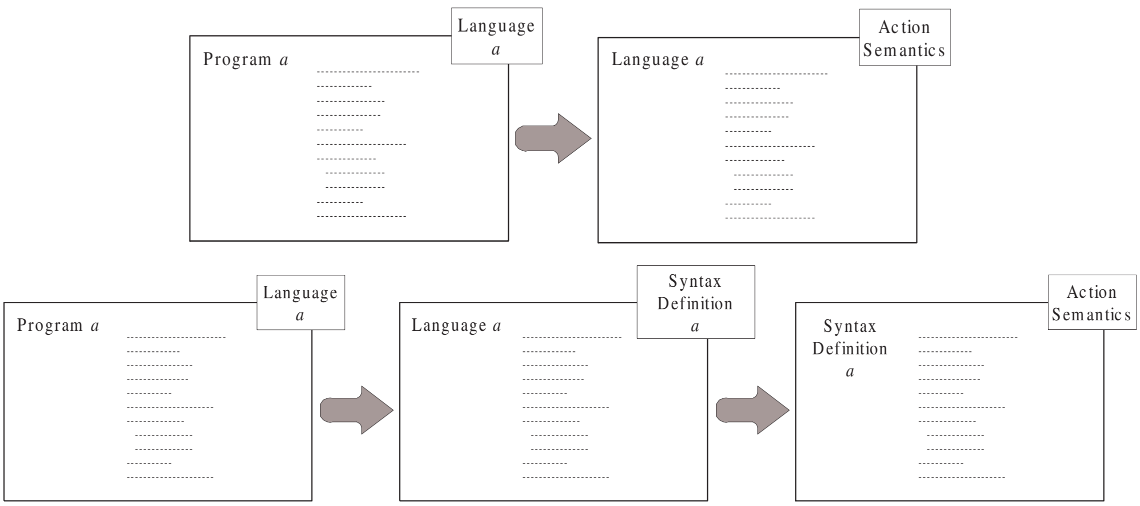

Figure 1: Adding one level to original syntax definition of action semantics

Figure 1 shows the possible three levels in the improved action semantics. The two levels which are labelled "Language" and "Action Semantics" on the upper part of the figure belong to the original action semantics. According to a certain language description (so-called "Language *a*" in the figure), a programmer can write a concrete program, i.e. "Program *a*". Because both the syntax and semantics of the language specification are rigorously defined, the programmer can test it in the implementation phase. 100% code generation from the implementation is also possible.

"Language *a*" itself, however, is specified in the formalism of action semantics, where it is nothing but a concrete model or an example. It is written in an action-semantics-specific format; and since action semantics is rigorously defined (both syntactically and semantically), the language description is rigorously defined, from which a compiler can be automatically generated.

As discussed above, this scheme has a drawback: language-specific symbols are embedded in the language description, limiting aspect[1] reusability among different languages. The solution is found by adding a level between language description and action semantics – syntax definition.

In the new proposal [Mos02], a program is described according to a specific language, the language is described according to the syntax definition formalism (combined with action semantics[2]), and syntax definition is described in underlying semantics. The border between levels is unimportant and obscure in nature, but the concept is clear and powerful. This scheme is shown in the lower part of Figure 1.

1.3.2 Examples of SDF description

To give as concrete examples on the look and feel of SDF description, the following pieces of description are excerpted from [Mos02].

An example of action notation in SDF:

```
Action ActionInfix Action → Action {left}
"moreover" → ActionInfix
```

`{left}` denotes the left associative precedence.

According to this action notation, `A1 moreover A2` is a valid action composed from subactions `A1` and `A2`. `A1 moreover A2 = (A1 and A2) then give overriding_` is a valid statement in the language description, where `give overriding_` is a kernel data operation and is predefined.

An example of data notation in SDF:

```
Data DataOpInfix Data → Data {left}
"+"|"-"|"*" → DataOpInfix
```

According to this data notation, we expect `a1 + a2 - a3` to be a data expression in the language specification (if we define `a1`, `a2` and `a3` to be data previously).

An example of condition notation in SDF:

```
conditional(Cond, Stmt, Stmt) → Stmt {cons}
```

`{cons}` specifies it is a condition statement.

An example of semantics notation:

```
execute Stmt → Action
```

The semantics definition of this syntactically defined `execute` statement would be found in another separate module. It may look like:

```
execute sequence(S1, S2) = execute S1 and then execute S2
```

There can be multiple such semantics definitions in the module, each defining a single semantics pattern. The combination of these definitions gives the full semantics of the `execute` statement in the SDF description.

The last mentioned module is not part of the SDF description, because it defines semantics rather than syntax. `execute`, `sequence` and `and then` are primitive notations and combinators in action semantics.

[1] Though **aspect** is a term from another branch of software engineering, it very appropriately refers to a common feature in different programming languages here.

[2] The role of syntax definition is quite like UML, which only defines the syntax without any notation about the actual semantics. Action semantics helps to rigorously define the semantics.

1.3.3 Reusability enhancement

From time to time we want to migrate from an existing language to another and reuse as much as possible. In case when the two languages share exactly the same semantic features but differ syntactically, the only thing we have to do is modify the SDF description. This is because in all the modules which defines the original language, only the SDF description depends on the concrete syntax. Other modules, of course, must conform to a certain syntax, but that syntax is defined in action semantics and is thus universal.

For instance, by changing the action notation example in the last section to:

```
ActionPrefix "(" Actions ")" → Action {left}
Action → Actions
Action "," Actions → Actions
"actlist" → ActionPrefix
```

we completely change the action syntax. `actlist(A1)`, `actlist(A1, A2, ···, An)`, `actlist(A1, actlist(A2, A3), A4, actlist(A5))` are now acceptable. (Supposing `A1`, `A2`, ···, `An` are actions.)

An other example is to change the data notation in the last section to the following:

```
Data Data DataOpPostfix → Data
"+"|"-"|"*" → DataOpPostfix
```

to let the language accept postfix expressions. Note that because a postfix expression has explicit evaluation ordering and is never ambiguous, we can specify {`left`} for the first statement, or {`right`}, or just ignore this option.

In reality, when the language is modified, it is impossible to keep all the other modules untouched and just modify the SDF description. Fortunately, our reason for introducing the SDF level is not to avoid changes in the language description, but to minimize them.

Without this additional level, in practice people often find rewriting the whole language description more reasonable than modifying the existing one, because it is too difficult to find out all the lexical symbols scattering in the whole description and modify them correctly. Maintainability and readability must also be considered.

2 Semantics in UML

2.1 A lack of semantics in UML

UML, in its original purpose, strives to provide all kinds of programmers with a unified language in which they can share and exchange designs without loss or misunderstanding of information. However, it soon turns out that this attempt fails due to a lack of semantics.

2.1.1 Semantic ambiguity

What UML does define is the syntax of a language, i.e., the exact syntax of statecharts, class diagrams, sequence diagrams and so forth. As to the meaning of these diagrams, e.g., the meaning of a guard in a statechart, the meaning of a *protected* member in a class, the meaning of a method call or message sent from an object to another in the sequence diagram, and so forth, there is no formal definition.

2.1.2 System/platform dependency

It is impossible, for system/platform-independent language without any semantics notation, to be system/platform-independently understood [MTAL98]. People with different backgrounds interpret the same graph in different ways, making it entirely impossible to maintain a common understanding.

2.1.3 Excess requirement of language-specific knowledge

This chaos deeply affects the application of UML modelling tools. Most of them require an understanding of how the model is executed. In another word, suppose the tool is built in C++ and transforms the statechart into C++ before executing it or generating code, more or less the user must know how the statechart is transformed into C++ in order to write expressions in the guards or output correctly.

Even worse, these tools usually accept language-specific expressions, and place them unchanged in the final code. In the last example, guards can be written as C++ boolean expressions and output can be written as C++ method calls. The model is thus very modelling-tool-specific. There is hardly any chance for such a model to run correctly in another tool, knowing that different modelling tools often use different programming languages. Even if they use the same language, they place the above-mentioned expressions in different position of the code, and interpret them in different ways.

2.1.4 Strict distinction between design and implementation

Another defect of this solution to modelling software is that because there is an intrinsic separation of implementation from the prototype design [RFBLO01], modelers are not allowed to test the model until the code is finally generated. Only after the design of the whole model is finished can those statements written in C++ be executed correctly. Before that time, test on a specific region is difficult (if not impossible) because it strongly relies on the other parts, which are still under construction.

To separate design and implementation to such a degree, of course, is not a good idea. Errors are often found in the implementation phase. Correcting those errors, which could have been found if people considered implementation and tested their prototypes in the design phase, requires far more effort. Furthermore, maintaining such a piece of software, whose implementation obscures its original design, possibly full of patches, is a nightmare!

2.2 Attempts to achieve executable UML

There are various ways to extend UML so that models become executable. Each of these approaches has its pros and cons. In spite of all these alternative approaches, the current trend is to unify semantic notations into a universally accepted language, such that various modelling tools can be based on it, and models described in this language is system-independent and can be manipulated by any of these tools.

2.2.1 Implementation based on a specific language

The most straightforward approach to executable UML models is to create a direct link (or dependency) between the meta-model and a specific language. In the meta-model, users are allowed to write statements and expressions in the language (called an “action language”). The libraries of the language are also available for immediate use in the model. When the design of the model is finished, it can be converted into source code (again, in the same language), and can be executed after compilation (Figure 2).

The language chosen is arbitrary. It can be any of the contemporary programming languages such as C++, Java, Python [Fen02] and so on.

This gives model designers much power to express software systems. In a sense, to design a model is much like programming, because they have to write real code in appropriate places of the model [3]. The code has unlimited power to access and modify any object or variable in the run-time system.

Because *almost* all the modelling environments implemented in this scheme naively place the language-specific expressions and statements unchanged in the source code generated from the model [4], the disadvantages are very obvious:

- No programming language contains all the useful software concepts. Though different programming languages may be equivalent in power and are able to produce all kinds of software, they differ in the difficulty or ease with which different problems are solved.

[3] Though this work is only done in the detailed design and implementation phases, it needs to be done at some point.
[4] The model itself is more like a code template in this sense.

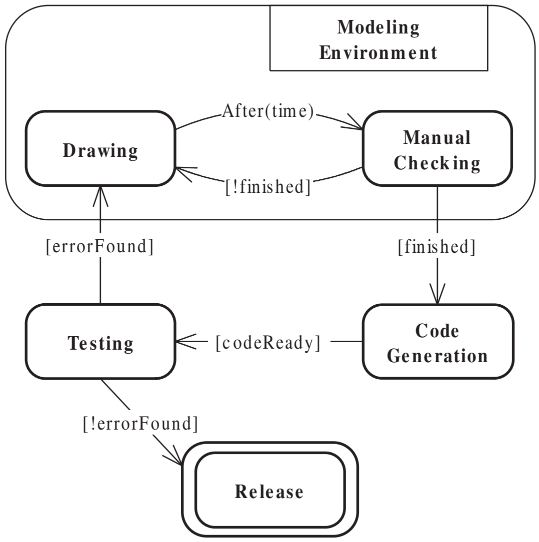

Figure 2: Model design based on specific language and compilation

- UML concepts are not directly supported by any contemporary programming language [MTAL98].
- Using any of the programming languages in the models enforces modelers to focus on verbose implementation details too early.
- The expressiveness of the code embedded in the model makes automatic model analysis and verification difficult. This also makes the generated software error-prone. It is difficult to tell whether the error is caused by the model structure or by the embedded code.
- The requirement for compilation delays the change to the system and increases time-to-market [RFBLO01]. The system must be re-installed.

2.2.2 Declarative Specification

The Object Constraint Language (OCL), as a declarative specification language, is already included in the UML standard. It is very useful in specifying guards in statecharts, pre- and post-conditions of methods, and all kinds of boolean expressions in UML.

However, though the pre- and post-conditions define the input sets and output sets of a method, OCL has no ability to specify algorithms, i.e., which input value matches to which output value [MTAL98]. This task must be taken up by another more powerful and more expressive language.

2.2.3 Virtual machines

The idea of a virtual machine is to build an environment that can actually run the model [RFBLO01]. The environment provides modelers with a set of operations or functions, quite like the instruction set of machine language.

In a virtual machine, like the Java Virtual Machine (JVM), every instruction has its exact meaning, and in such a way the semantics of models is rigorously defined. The virtual machine also enables early analysis and verification. Code generation becomes unnecessary.

Built up as an Integrated Development Environment (IDE), the modelling tool is thus a design environment and a virtual machine at the same time. Models can be modified, and the modification is immediately reflected in the execution. Furthermore, the modification can be done at run time. This makes the model very flexible as shown in Figure 3.

Classes and associations, conventionally considered static, are now placed in a dynamic context and represented as objects in the environment. So, in the environment, every part of the model can be modified.

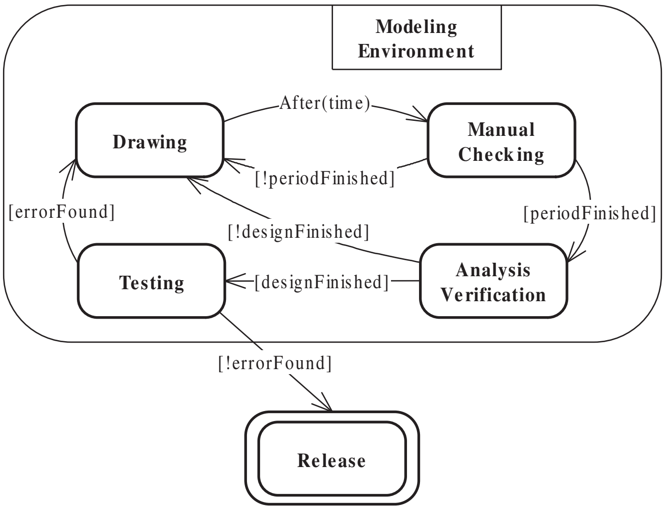

Figure 3: Model design based on virtual machine

There still remains the question: *Is the virtual machine universal?* From the implementation point of view, virtual machines increase productivity and shorten time-to-market. But this idea itself has nothing to do with the semantics. Consider interpreted languages such as Visual Basic, Python and Perl. Which should be chosen as the internal representation of a model? Or should we create a new language? If none of these languages satisfies the requirement, is action semantics (AS) a good choice? If so, the solution is more suitably regarded as an action-semantics-based solution.

2.3 Toward a common language to express semantics

To cope with the problem that UML lacks a rigorous semantics, the OMG (Object Management Group) issued an RFP (request for proposal) for such a semantics for the UML. The Action Semantics Consortium responded to this RFP and suggested new constructs to the UML.

Action semantics, as discussed above, is good at rigorously defining semantic concerns of programming languages. It must be clarified that by combining action semantics with UML, we can get a rigorously defined modelling language and also a rigorously defined meta-modelling language. Models designed with such a rigorous semantics can then be simulated, analyzed and verified at an early time.

Based on the industrial practice such as SDL, Kennedy Carter and BridgePoint, the Action Semantics proposal aims at providing modelers with a complete, software-independent specification for actions in their models [SGJ02]. As a future OMG standard, action semantics goes one step further than its predecessors.

3 Action semantics at the model level

Action semantics as a *package* for UML, helps to define the actions in models. Its goal is not to usurp the responsibility of all the other packages such as OCL, but to cooperate with them in a harmonious way. OCL, as a well established part of UML, is still used to specify logical expressions in the new UML configuration. Those logical expressions give the pre- and post-conditions of methods and guards of transitions. Action semantics, however, is used to specify the body of the methods, i.e. their behavior.

3.1 Primitive actions and flows

Action semantics defines a core package for UML, which serves as the basis of all kinds of actions. The core consists of a set of fundamental actions and their semantics, as is illustrated in Figure 4 [BB01].

Two kinds of flows control the execution sequence of actions: control flow and data flow. All actions are treated as executing concurrently except those sequenced by a data flow or a control flow

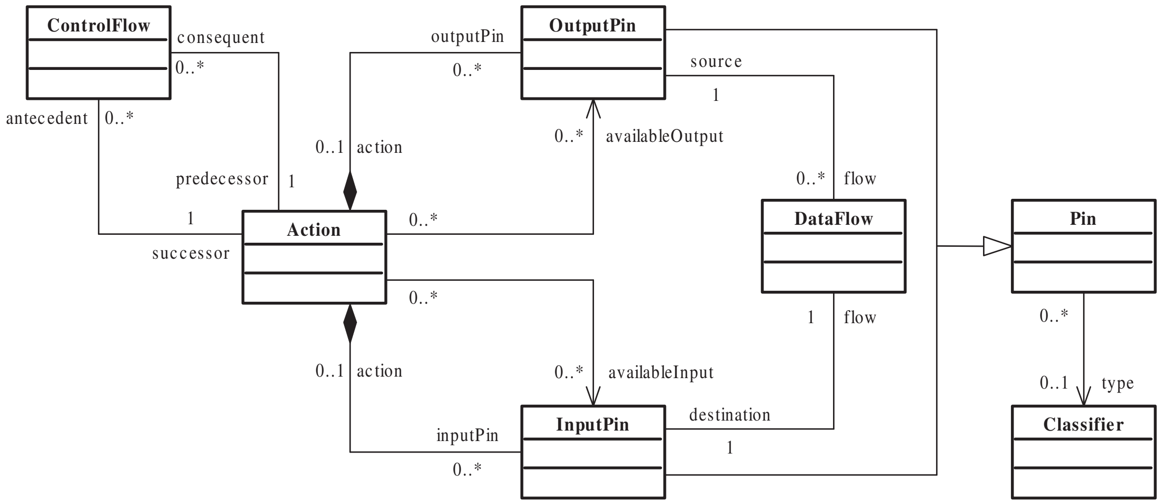

Figure 4: The minimal core of the action semantics package

[AILKC+00].

- Control flow. An action is executed only after all its antecedents are completed. The termination of an action may activate one or more of its consequents, provided that their other antecedents are also finished executing. So, as the class diagram shows, an action is associated with a set of antecedents and a set of consequents.
- Data flow. Every action has two sets of input pins and two sets of output pins associating with it. One of the input pin sets (denoted by A)contains all the required input pins for the action. They can be considered as the formal input parameters. The other input pin set (denoted by B) contains available pin data at a certain point in time. Since this set is associated with the action and only provides information which it interests in, it is a subset of B. An action can be executed only after all the associating input pins are available, i.e. $A = B$. Similarly, only when the action is completed does output pin set A' equal to B'. As the data flow carries on, data from the output pins of a preceding action become a source of the data flow, and then the confluence reaches the input pins of another action.

3.2 Composite actions

Composite actions allow the composition of simpler actions into more complex ones. They are recursive structures. This means composite actions themselves can be a component of a larger composite action.

There are three kinds of composite actions: group actions, conditional actions and loop actions.

3.2.1 Group actions

Actions may be grouped to represent a specific design concern. On the one hand the grouped entity is conceptual. Its boundary does not hinder data flow or control flow. Data flow must be directly connected to the input pins of the actions in the group, because a group action has no input pins. Control flow may also cross the group boundary and be directly connected to subactions. In this view, a group action is nothing but a logical view of its subactions.

On the other hand, when group actions are physically connected with control flows, they are placed in an execution sequence with their predecessors and successors. In this case, they follow the rule of sequential execution, i.e., they cannot be executed before all their predecessors are completed.

3.2.2 Conditional actions

A conditional action is a composition of clauses, each of which contains *exactly one* test action and one body action. A test action accepts input data from the available input pins, and produces a truth

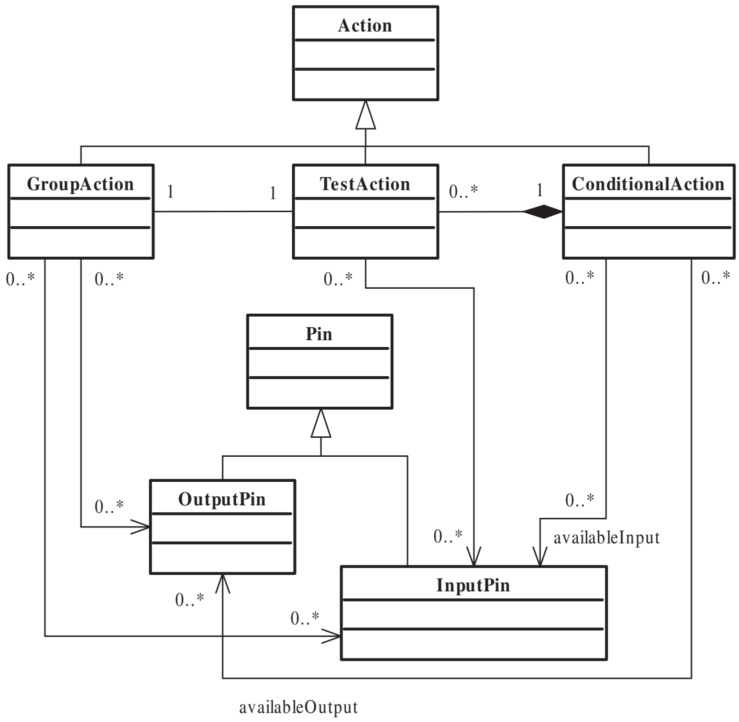

Figure 5: Conditional actions

value. Its associated body action is executed if and only if the test result is *true*. When executed, the body action, which can be viewed as a group action as a whole, accepts available input data and produce output for the available output pins at completion (Figure 5).

Different body actions, though they may require different input, must produce output to the same set of output pins. So no matter which body action is executed, the output of the conditional action is always complete.

One might notice from the figure that a conditional action is not explicitly associated with input pins or output pins. It is only associated with available input pins and available output pins, since the input and output is implicitly given by or given to the execution context of the conditional actions.

As another underlying rule, there should be no data flow between two clauses of a conditional action.

Clauses can have noncyclic predecessor-successor relationships among them. In this way, a clause may be defined as a successor of some others, meaning that its test action (and body action if possible) is executed after the predecessors. Of course, according to the rule that exactly one clause gets executed at a time, if any of its predecessors is executed, the successor clause could not be executed. Test actions of unrelated clauses may be executed concurrently.

An application of the clause relationship is to define a default body action for the conditional action: the default action is specified as the successor of all other clauses and its test action always return true.

In this way, clauses form a hierarchy. Test cases in each level of the hierarchy must be *mutually exclusive*, though they may not cover all possible conditions.

The above assertion assures the model to be deterministic under all conditions. This conforms to the well-formedness rule in [AILKC$^+$00].

3.2.3 Loop actions

A loop action contains a single clause of a test action and a body action. Its test action is executed repeatedly. When the evaluation result is *true*, the body action is executed; otherwise, the loop action finishes.

Besides its input pins and output pins, a loop action also has loop variable pins. Before its execution, all the required input must be available from the input pins. Data on the input pins are then copied to the loop variable pins. The test action each time tests the condition based on these loop variables, and the body action takes the loop variable pins as both input pins and output pins. Whenever the test result becomes *false*, the loop stops and the loop variables are copied to the output pins as the result.

3.3 Read and write actions

Read actions retrieve values from objects, attributes, links and variables, while write actions write values to them. The read/write actions described here are primitive actions and only provide simple functions. More complex read/write actions can be defined by composition.

3.3.1 Object actions

When considered as general targets of read and write actions, the only property of objects is their classes. An object at run time has an identity (i.e., a unique identifier). Its only attribute, *classifier*, gives the *current* class of the object. The classifier can be set at model time and also at run time. Read action *ReadIsClassifiedObject* accepts a classifier from one input pin and an object from the other, and returns the result whether the object belongs to the class specified by the classifier. Write action *CreateObjectAction* produces an object and puts it on an output pin; *DestroyObjectAction* takes an object from its input pin and destroys it. *ReclassifyObjectAction* changes the classifier to another one (i.e., reclasses the object).

3.3.2 Attribute actions

Attributes of objects can be read and written by attribute actions. The names of the attributes referred to are statically bound to the actions. This means that their names and order of appearance are specified in the model, and cannot be changed at run time. So for read actions, the only required parameter from the input pins is the object instance (or object identity). These actions read the specified attributes and put their values on the output pins. The order of the values is the same with the order of the attribute names given at model time. For write actions, they take an object instance and also new values as input, and modify the attributes accordingly. No output is produced.

Attributes may be sets or ordered sets; for example, a composition with a class whose multiplicity is 0..* in a class diagram. In both cases duplicated values in an attribute are not allowed. A read action reads all the values of an attribute, regardless of how many there are. However, it is possible to insert a value into an attribute with a write action. When the attribute is ordered, the insert action takes one more input as (run-time) position parameter; if the new value already exists in the attribute, it is moved to the new position. When the attribute is unordered, the position is omitted, and inserting a duplicated value to the attribute has no effect.

A special object *ownerScope* is a first class object. Such an object represents a classifier instead of an object instance. When a read/write action is carried out on it, the identity of the *ownerScope* object is not required from the input pins, because the target of the action is statically bound to the classifier.

3.3.3 Association actions

Associations are the classes of links. When instantiated, a link connects two or more objects[5]. There can be multiplicity of more than 1 for each end of an association, but the multiplicity for a link end can only be 1..1, because when instantiated, there should be exactly one object at a link end. (Link ends with no object are not considered.)

Association actions actually perform read/write actions on links. A link cannot be passed to an action from input pins. Instead, it is the objects on link ends that are passed to the action. So, each association action is statically bound to a specific association, with exactly one input pin for each end

[5]A link of binary association class connects two objects; other kinds of links each connect more objects.

of the association. From those input pins, objects can be passed to the action as parameters specifying a link instance.

For write actions, all the link objects are given at run time. The action can thus locate the link and modify it. There are strict limitations for write actions: when a link is created, it can only be destroyed or reordered; neither qualifiers nor link objects can be changed.

For read actions, $n-1$ objects are given on input pins, where the link actually has n objects. The read action is to find the omitted object, which becomes the result on an output pin when the action is completed.

3.3.4 Variable actions

Each variable action is statically bound to a variable. A read action takes no input and produces exactly one output, which is the value of the associated variable. A write action *usually* takes one input as the new value of the variable and rewrites it, with a special case that an insert action may take one more input as the insertion point where the new value is to be inserted into the variable (if it is a set or a list).

3.4 Implementation-dependent Computation actions

Computation actions take in input values, perform pure functional computation on them, and then return output values to the output pins. They are comparable to the *constant* functions in programming languages, which are self-contained and make no change to the current state.

Computation actions are an implementation-dependent part of action semantics for the UML, because there is no rigorous definition for primitive actions. Primitive actions can be written in a specific programming language and is left entirely to the modelling environment. The only requirement for the primitive actions is their signatures, including names, a list of input types and a list of output types.

Researchers find it very hard to define primitive actions for computational purposes, given that different application areas have different requirements [AILKC$^+$00]. However, this lack of formal definition for primitive actions forbids the porting of models from a modelling environment to another. Even worse, in practice few models run without any computation action. Actually, the power of computation actions usually tempts designers to abuse them. It is still necessary to define a primitive action set or a primitive language to maximize portability, if it is not ensured.

3.5 Collection actions

A collection as a whole can be manipulated by a collection action. Elements in the collection are ordered or unordered. The action applies a subaction on one element at each time. The collection is given to the collection action from an input pin, and the result is produced and put on an output pin.

Some of the collection actions described below require the collection to be ordered, while others do not.

A collection action may also take multiple collections as input. In this case, the parameter to the subaction is a tuple with multiple elements, each of which is taken from a different collection. The collections must be ordered and have the same cardinality to have their elements matched in a unique way.

It is also possible that, apart from the collections, the action has more input pins which take single values. Those values are passed to the subaction in each execution. They are considered as constants, and will not be put on the output pins.

3.5.1 Filter actions

The subaction of a filter action is a boolean function mapping an element to a boolean value. The result decides whether the element is included in the output collection. The filter action may perform concurrent subactions on different elements, depending on the implementation. But, the subaction must support concurrency and re-entrance.

The input collection is considered unordered in this case, and the output collection is also unordered.

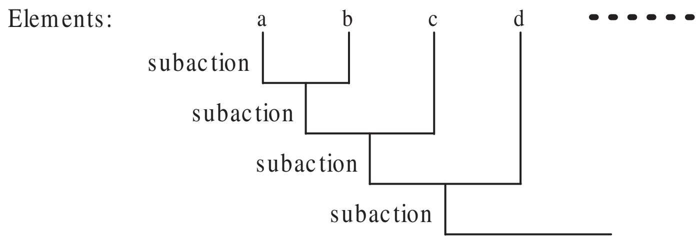

Figure 6: The process of reduce actions

3.5.2 Iterate actions

An iterate action applies its subaction on collection elements in sequence, until all the elements are exhausted or the evaluation of loop variables becomes *false*.

Both the input collection and the output collection are ordered.

3.5.3 Map actions

A map action applies the subaction on each of the element in parallel, and returns the collection containing the results of the subaction. This is like a functional mapping between the elements of two unordered collections.

3.5.4 Reduce actions

The subaction of a reduce action takes two input values of the same type as the elements, and produces an output also of the same type. The reduce action applies the subaction sequentially on pairs on elements until the final result, which is the same type as the elements, is produced. The process is illustrated in Figure 6.

The input collection is ordered. The output is not a collection but a single element.

3.6 Messaging Actions

Messaging actions provide a mechanism to facilitate invocations between objects. The invocation through messages is so generally termed that it encompasses virtually all kinds of run-time interactions among objects. The objects may be located on a single machine or distributed throughout the network; the invocations may be procedure calls (as usually found in most programming languages), or transitions triggered by events in a state machine.

At model time, messages are modelled as classes, with attributes as parameters or return values. Each message class designates a specific request from the requestor. In this sense, the type of request is statically bound to a class; a message at run-time is an instance of the class. As the type of a message is a class, different types form generalization hierarchies [AILKC+00]. The run-time environment requires a mechanism to dynamically look up the hierarchy to match a message with behavior, because the behavior of the receiver is not statically bound to the class.

Parameters and return values of a request or a reply are stored in an instance of the message class. Sometimes a request does not need any parameter or an invocation does not return any value (asynchronous requests described below are an example), but the message instance must be sent or received. As mentioned above, the classifier is an inherent property of an object, the receiver may use the classifier as a parameter, even if the object contains no extra information. In this case, a dynamic lookup in the class hierarchy is necessary.

3.6.1 Asynchronous invocation and synchronous invocation

There are two kinds of invocations: *asynchronous* and *synchronous*. Each kind has a set of actions. Asynchronous actions and synchronous actions are disjoint sets.

When an object executes an asynchronous action with a message as parameter, it starts an asynchronous invocation. The action returns no value and the requesting object does not wait for a reply from the requested object. When the requesting object and the requested object are on different machines, the requesting one even does not wait until the other object actually receives the message.

Upon receiving the message, the requested object might queue it according to a specific strategy, which is not in the scope of the basic action semantics. When it is free, the sequential object processes the message[6]. Processing the message firstly means a decision on its type is made. The run-time environment must be capable of dynamic class lookup. When the type is determined, the environment can thus match the message with a specific behavior. The execution of this behavior usually results in a procedure call.

The requested object may or may not return some values to the requesting one. If values are to be returned, the requested object must execute another asynchronous or synchronous action.

In contrast, if the requesting object executes a synchronous action, it is blocked until a reply is received from the requested object. Even if no return value is required for the invocation, the requested object must send back a message indicating the processing is complete, and the requesting object is allowed to proceed with its jobs.

3.6.2 Consideration for distributed systems

For distributed systems, references become meaningless. From this consideration, messages are uniformly transmitted *by value*. So the identifier of a message is unimportant. The message must be duplicable, so that transmission does not cause a loss of information.

The transmission of messages via the network is time-consuming and the message may be lost on the way. For instance, when an object issues a synchronous invocation to another object, but the message is lost, the requesting object would be blocked forever. The basic action semantics does not give a solution to this problem. It is elaborated in other real-time profiles.

3.6.3 Procedures

Procedures themselves are not actions, but rather combination of actions. A requested object receiving a message usually implies a procedure is to be executed. Note that an invocation can not directly execute an action, but only a procedure.

The concept of procedures in action semantics provides a certain level of modularity. Procedures are self-contained. The execution of a procedure is not allowed to access any *global variables* or *static variables*. Instead, all the variables are contained in an object in the form of attributes.

There is no means to explicitly execute part of a procedure from the requesting object. A procedure is regarded as an entity, which accepts input and produces output (like actions). The input is taken from the attributes of the request message. For synchronous invocation, the output serves as the return value; for asynchronous invocation, the output is simply ignored.

3.6.4 Procedure lookup

At run-time, a request must be matched to a procedure on the receiver side. In the context of an object, this mechanism is called *operation lookup*. It requires certain kinds of maps to look up the type of a message, and locate the appropriate procedure of the request object.

A map may be global or associated with a class or an object. A global map records all the procedures and their signature. The look-up in this map is very like a traditional procedure call. A map associated with a class only records the methods of the class. If no method in the class matches the request, another class is searched, which is linked to it with a pointer. A map associated with an object records the methods of the object[7]. If no method is matched, another object linking with it is then searched.

A matching from a request type to a procedure in the context of a state machine is called *transition triggering*. In a state machine, each state is a state of the requested object. A map is associated with

[6]This does not mean the requested object is completely idle. It may process multiple messages at the same time, provided that its implementation allows this.

[7]Be reminded that the classifier of the object may change at run-time.

the object or the class of the object. Similarly, if no procedure is matched in the map, another object or class which is pointed to is then searched.

3.7 Exception actions

When the execution of an action enters an abnormal state, another action takes the control flow and the original action is aborted.

There are two kinds of exception actions: exceptions and interrupts. When an error occurs in an action, an exception of a certain type is generated, and the run-time system looks for a *handler* to handle it. If the action has an exception handler table, the system matches the exception in the table to find an action to handle it; if no action matches, the exception is propagated to the outer context, until it is handled.

Interrupts are signals that force the current action to stop and change the control flow to another action. It is possible to base the choice of new control flow on the current state of the object. Interrupts do not propagate.

3.8 Timing issue

Time is defined in action semantics to precisely express the behavior of models, which may be sequential or concurrent.

For sequential models or parts of models, the ordering of action executions conforms to the timing causality. A sequence of actions is executed one by one.

For concurrent models or parts of models, the ordering of action executions is unimportant, unless different actions are required to be synchronized to access shared variables. Without explicit synchronization, actions may be executed concurrently or in a meta-model-dependent order.

3.8.1 Terminology

The behavior of a model contains a number of *execution entities*, each of which has its own life cycle from the creation time till the destruction time. In its execution, each entity is only interested in its own behavior and time, unless it must synchronize with other entities for some reason. An entity has an *identity*, which is unique and does not change over time.

The dynamic behavior of an entity is a sequence of *snapshots*, each of which represents a stable state of the entity. They are very like the frames in a movie. Note that a snapshot represents a stable state of an entity at a certain point in time instead of the state of the model. A model may be mutable during the execution of an action. Different entities in the model have their own snapshots over time, and when entity E_1 has a snapshot, it doesn't mean another entity E_2 is also in a stable state at that time.

A *change* causes a transition between two states of an entity, so it is associated with two snapshots: a predecessor and a successor. Both of the states are stable, but the change may take some time during which the state is volatile. Other entities are not allowed to access the variables of this entity if it is in a change. A number of changes are associated with a specific entity.

Time is an important property of a change, which specifies when a change occurs. It is usually a nonnegative integer with 0 representing the creation time of the entity. Because of this, the time is relative to the life of the entity. The execution environment may maintain a global time for internal use, but entities in an execution of the model only have the knowledge of their local time.

A *history* is constituted by a sequence of changes of an entity over time, which starts from the creation of the entity and ends with its destruction. Because the history is for a single entity, every change in the history is associated with the same entity. History is a sequential chain of changes. The successor snapshot of a change (except the last one) is the predecessor of the next change.

The execution of an entity can be viewed as an entity in its own right, which allows us to focus more on the dynamic behavior of the entity. It has its own identity and history. A change in the history of such an execution is a *step* in the execution. An *execution snapshot* is a stable stage between steps. Step is the subclass of change, and execution snapshot is a subclass of snapshot. Steps can access other snapshots, but the access must be synchronized; so each of them keeps a record of referents —

the snapshots that are to be synchronized with it. And of course, the accessed snapshots also keep records of these accessors in attributes.

3.8.2 Timing

As mentioned, the occurrence time of a change is treated as its attribute. The change is placed in a history, which is a sequence of changes for an entity. The order of those changes conforms to the incremental order of their time attributes. In this way, changes are carried out one after another, with the invariant that the first change (always at time 0) is the creation of the entity, and the last change is its destruction.

As time is an attribute of changes, one can imagine that even the time (or, the sequence) of changes is mutable at run-time. The basic action semantics gives no narration on this issue. If this is permitted, interesting features will exhibit themselves in various models, because the execution of a model is capable of dynamically modifying the structure of the model itself, which affects the execution in return. Modifying the time attribute of another change to a value larger than the current local time is just like scheduling an event in the future; modifying it to be smaller makes no sense, because the time has already elapsed and the past is cannot be altered.

3.9 Action description

This section gives the look and feel of the action description.

Associations

Describes the associations with this action, in particular, defines the types and cardinalities of input and output.

Inputs

Defines the input pins and their multiplicity. Inherited pins are listed and marked with the modifier <inherited>. If the value of an input pin is obtained via more than one association, the OCL dot-notation is used to specify the path to navigate (i.e. end1.end2.inputName : type [multiplicity]*).*

Outputs

Defines the output pins and their multiplicity. The syntax is the same as the input section.

Well-formedness rules

Like many of the other action semantics descriptions, action description can be associated with a set of well-formedness rules.

Execution Semantics

Describes the semantics of the action. It can have one optional Assertion *section and one or more* Production *sections. The assertion section uses OCL to define the global invariant of the execution. A production section contains three parts: a precondition and a postcondition use OCL description to specify the precondition and post condition of the action, an optional assertion specifies the assertion for this action. In both production and assertion sections, there can be English descriptions for better understandability.*

Table 3: Action description

An action description describes the semantics of an action. It reuses the OCL language, but is not limited to specifying constraints. Its basic components are shown in Table 3, some of which are optional, while the others must be provided, even if their body is just *none*.

An example of action description is given in Table 4 [AILKC^{+}00]. It describes the CreateObjectAction, which is statically associated with a class and creates an object at run-time. No input is required. The only output is the new object on the output pin. No assertion section is needed. The well-formedness rules assure the well-formedness of the execution.

Associations

• class : Classifier [1..1]	Classifier to be instantiated.
• <derived> result : OutputPin [1..1]	Derived from Action:outputPin. Gives the output pin on which the result is put.

Inputs

None

Outputs

• result : ObjectIdentity [1..1]	The created object. The type of identity must be the classifier specified for the action.

Well-formedness rules

[1] The classifier cannot be abstract.
not (self.classifier.isAbstract = true)

[2] The classifier cannot be an association class.
not self.classifier.oclIsKindOf(AssociationClass)

[3] The classifier of the result pin must be the same as the classifier of the action.
self.result.type = self.classifier

[4] The multiplicity of the output pin is 1..1.
self.result.multiplicity.is(1,1)

Execution Semantics

context self: CreateObjectActionExecutionSnapshot

Production 1: The result has exactly one classifier in the successor snapshot, which is the same as the classifier specified as a parameter of the action. The result has no attribute values in the successor snapshot and does not participate in any associations.

Precondition:

self.status = ready

Postcondition:

```
let newidentity : ObjectIdentity =
    self.pinValue → select(pin = self.executionID.action.result).value in
        self.status = completed and newidentity.new()
        and newidentity.classifier → size() = 1
        and newidentity.classifier → includes(self.executionID.action.classifier)
        and newidentity.linkEndValue → size() = 0
        and newidentity.attributeValue → size() = 0
```

Table 4: Action description for CreateObjectAction

4 Action semantics at the meta-model level

Instead of merely used to specify the dynamic behavior of the model execution, action semantics, similar to a programming language in some sense, is also capable of restructuring the model design at the meta-modelling level.

When action semantics descriptions are employed at design time, they are executed in the meta-model instead of the model. The model may not be finished at that time, so it is impossible to execute the model. The action semantics description running on the meta-model is to automatically restructure the design and possibly make it conform to a certain *design pattern* or *refactor* it [SPH+01].

Using action semantics at this level enhances the functionality of the meta-model. However, it is just introduced to facilitate the design work, without which designers are still able to make the same modification manually.

4.1 Object-oriented refactoring

Refactoring was introduced in [Opd92] to make the source code more understandable and readable. This is a behavior-preserving transformation, but it helps programmers to manipulate common language constructs, such as class, method and variable, of an existing application, without modifying the way it works [SPH+01].

The same idea is used in the UML model design. A good example is to refactor a class diagram, which happens to be ill-designed.

Ill designs are usually found in models, i.e. identifying a composition relationship with an inheritance (or generalization) relationship and thus an invalid inheritance is formed. In this case, an action defined in action semantics is used to discover all these design flaws and, if found, correct them. (Of course this process is very limited and also requires a lot of human work.) As mentioned above, action semantics uses the OCL dot-notation to navigate among all the objects in a model, it is able to look into every detail of the model design.

```
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)
```

Table 5: An example of refactoring

Table 5 shows another example of removing all the useless classes. Such classes are not referenced by any other class and they have no subclasses (or no features) [SPH+01]. Two functions, `addSuperClass` and `delete` are not defined in action semantics. They are defined by the programmer to add a generalization to a subclass and delete a class accordingly.

More examples on refactoring class diagrams and statecharts can be found in [SPTJ01].

4.2 Design patterns

Design patterns summarize and formalize good software design experiences, which can be reused in different contexts to increase modularity, stability, reusability, an so on [GHJV95].

Applying design patterns to the model design is desirable, because they are proved (either theoretically or practically) to be very good designs. The new model designs benefit from reusing these good designs.

However, an initial design may not conform to a design pattern even if it could. This is not always because of a simple-mindedness: the designer may choose to first design the model in a more straightforward way and test it; if the test result is satisfactory, design patterns are considered to improve the performance, modularity, and so forth.

Applying a design pattern to multiple models can be achieved in one step and automatically. The designer first identifies a design pattern or a variant of a design pattern, which improves the current model(s). Then a small program is written in action semantics to reconstruct parts of the model(s). For class diagrams, this reconstruction results in adding or deleting classes or associations. The difference between applying a design pattern and refactoring is the former approach also changes the model behavior. For instance, when applying the *proxy* design pattern, proxy classes are added between external access and the real objects. The proxy intercepts the access requests made to the real objects, relays refined requests to the real objects, and then sends back the results. This pattern makes the external design less dependent on the object's internal implementation.

For statecharts, new states may be automatically created, and the existing state hierarchy is also subject to change. An example is, in an initial statechart design, many integer variables are used to preserve states. However, to achieve a pure statechart design, only states preserve states and no variables are allowed, since they make it difficult if not impossible to analyze the model. Then an *integer* design pattern can be automatically applied to restructure the model by first discovering all the integer variables in use and then removing these variables and inserting new states in appropriate places to provide the same functionality. If the very large amount of additional states are written manually, it is an extremely boring and error-prone process.

4.3 Aspect weaving

Different aspects of an application are separated at design time. They are only woven together when necessary.

Separating different aspects at an early time is desirable, because it makes the design more modular by limiting the dependence between aspects. The interface is thus very clear and changing one aspect usually does not require any modification on other aspects.

When aspects are to be woven together for test or distribution purposes, an automatic process is required. This is accomplished by an action semantics description representing the aspect weaving rules. The description is run (or interpreted) by the modelling environment.

An example of aspect weaving is a computation system originally designed to run on a single computer, while later requirements require to run it in a distributed system. If the information sharing and passing aspect is separated from the others at an early design phase, the porting from a single computer to the distributed system is just to redesign this aspect and run the same action semantics description again to weave all the aspects together.

5 Conclusion

Action semantics is a very powerful tool to formally specify the behavior of UML models. When it is standardized, it will form a new package for UML, which is capable of rigorously modelling behavior and at the same time interacts with other components harmoniously. Model design at both the modelling level and the meta-modelling level will then change dramatically.

As all kinds of actions defined in the basic action semantics are discussed above, it is obvious that action semantics still has flaws, and disputes still exist on some critical subjects. This is natural for such a new development thing in such an evolving field.

References

[ÁCES01] José M. Álvarez, Tony Clark, Andy Evans, and Paul Sammut. An action semantics for MML. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 4th International Conference, Toronto, Canada, October 2001, Proceedings*, volume 2185 of *LNCS*, pages 2–18. Springer, 2001.

[AILKC+00] Alcatel, I-Logix, Kennedy-Carter, Inc. Kabira Technologies, Inc. Project Technology, Rational Software Corporation, and Telelogic AB. *Action Semantics for the UML*. Document ad/2001-03-01. OMG, 2000. Available from World Wide Web: `http://cgi.omg.org/cgi-bin/doc?ad/01-03-01`.

[BB01] Erwan Breton and Jean Bézivin. Towards an understanding of model executability. In *Proceedings of the international conference on Formal Ontology in Information Systems*, pages 70–80. ACM Press, 2001. Available from World Wide Web: `http://www.sciences.univ-nantes.fr/info/lrsg/Pages_perso/Publications/fois.pdf`.

[Fen02] Thomas Feng. Statechart virtual machine, 2002. Available from World Wide Web: `http://moncs.cs.mcgill.ca/people/tfeng/?research=svm`. MSDL, McGill.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[Mos96] Peter D. Mosses. Theory and practice of action semantics. In *MFCS '96, Proc. 21st Int. Symp. on Mathematical Foundations of Computer Science (Cracow, Poland, Sept. 1996)*, volume 1113, pages 37–61. Springer-Verlag, 1996. Available from World Wide Web: `http://www.brics.dk/RS/96/53/BRICS-RS-96-53.pdf`.

[Mos02] Peter D. Mosses. Action semantics and ASF+SDF - system demonstration, 2002. Available from World Wide Web: `http://www.cwi.nl/ftp/markvdb/entcs65.3/65.3.003.pdf`.

[MTAL98] Stephen J. Mellor, Steve Tockey, Rodolphe Arthaud, and Philippe LeBlanc. Software-platform-independent, precise action specifications for UML. In Jean Bézivin and Pierre-Alain Muller, editors, *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998*, pages 281–286, 1998. Available from World Wide Web: `http://www.kc.com/as_site/download/UML_AS_paper.pdf`.

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, UIUC, Urbana-Champaign, IL, USA, 1992. Available from World Wide Web: `http://www.cs.uiuc.edu/Dienst/Repository/2.0/Body/ncstrl.uiuc_cs/UIUCDCS-R-92-1759/postscript`.

[RFBLO01] Dirk Riehle, Steven Fraleigh, Dirk Bucka-Lassen, and Nosa Omorogbe. The architecture of a UML virtual machine. In *Conference on Object-Oriented*, pages 327–341, 2001.

[SGJ02] Gerson Sunyé, Alain Le Guennec, and Jean-Marc Jézéquel. Using UML action semantics for model execution and transformation. *Information Systems*, 27:445–457, 2002. Available from World Wide Web: `http://www.sciencedirect.com/science/article/B6V0G-45J8WV9-1/1/8622e2f14c9a518a5241431af02e27d8`.

[SPH+01] Gerson Sunyé, François Pennaneac'h, Wai-Ming Ho, Alain Le Guennec, and Jean-Marc Jézéquel. Using UML action semantics for executable modeling and beyond. In *Advanced Information Systems Engineering. 13th International Conference, CAiSE 2001, Interlaken, Switzerland, June 4-8, 2001, Proceedings*, volume 2068 of *LNCS*, pages 433–447. Springer, 2001. Available from World Wide Web: `http://www.irisa.fr/triskell/publis/2001/Sunye01a.pdf`.

[SPTJ01] Gerson Sunyé, Damien Pollet, Yves Le Traon, and Jean-Marc Jézéquel. Refactoring UML models. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 4th International Conference, Toronto, Canada, October 2001, Proceedings*, volume 2185 of *LNCS*, pages 134–148. Springer, 2001.