An Action Semantics for MML

José M. Álvarez[1], Tony Clark[2], Andy Evans[3], and Paul Sammut[3]

[1] Dpto. de Lenguajes y Ciencias de la Computación.
University of Málaga, Málaga, 29071, Spain
alvarezp@lcc.uma.es
[2] Dpt. of Computer Science, King's College,
Strand, London, WC2R 2LS, United Kingdom
anclark@dcs.kcl.ac.uk
[3] Dpt. of Computer Science, University of York,
Heslington, York, YO1 5DD, United Kingdom
andye@cs.york.ac.uk, pauls@cs.york.ac.uk

Abstract. This paper describes an action semantics for UML based on the Meta-Modelling Language (MML) - a precise meta-modelling language designed for developing families of UML languages. Actions are defined as computational procedures with side-effects. The action semantics are described in the MML style, with model, instance and semantic packages. Different actions are described as specializations of the basic action in their own package. The aim is to show that by using a Catalysis like package extension mechanism, with precise mappings to a simple semantic domain, a well-structured and extensible model for an action language can be obtained.

1 Introduction

The UML actions semantics has been submitted by the action semantics consortium to "extend the UML with a compatible mechanism for specifying action semantics in a software-independent manner" [1]. The submission defines an extension to the UML 1.4 meta-model which includes an abstract syntax and semantic domain for an action language. This language provides a collection of simple action constructs, for example write actions, conditional actions and composite actions, which can be used to describe computational behaviours in a UML model. A key part of the proposal is a description of the semantics of object behaviour, based on a history model of object executions.

Unfortunately, the action semantics proposal suffers from a problem commonly met when developing large meta-models in UML - how to structure the model so as to clearly separate its different components. Failure to achieve this results in a meta-model that is difficult to understand and to modify, particularly, to specialize and extend. In addition, meta-models based on the current UML semantics suffer from a lack of a precisely defined semantic core upon which to construct the meta-model. This means that it is often hard to ascertain the correctness of the model, and to overcome this, significant work must be invested in clarifying the semantics before any progress can be made. On the positive side, the basic semantic model used in the action semantics, with its notion of snapshots and history and changes seems quite appropriate to define the different changing values of a system. In addition, the actions defined in the submission thoroughly cover the wide range of actions necessary for a useful action language. Thus, if a way can be found to better restructure what is a significant piece of work, then clearly there will benefits to all users and implementors of the language.

The purpose of this paper is to show how the definition of a precise semantic core and the use of a Catalysis [2] like package extension mechanism can result in a better structured and adaptable definition of the action semantics. The work is based on an extension of the Meta-modelling Language (MML), a precise meta-modelling language developed to enable UML to be re-architected as a family of modelling languages.

1.1 The basics of the MML model

MML is a metamodeling language intended to rearchitect the core of UML so it can be defined in a much simpler way and it can be easily extended and specialised for different types of systems. Among other basic concepts, MML establishes two orthogonal distinctions, the first one being a mapping between model concepts (abstract syntax) and instances (semantic domain). The second is the distinction between model concepts and concrete syntax and applies both to models and instances. The syntax is the way concepts are rendered. Models and instances are related by a semantic package that establishes the satisfaction relationship between instances and models. A similar relationship is defined between between model concepts and concrete syntax.

These distinction are described in terms of a language definition pattern, see Figure 1. Each component in the pattern is defined as a package. As in UML, a package is a container of classes and/or other packages. In addition, packages in MML can be specialised. This is the key mechanism by which modular, extensible definitions of languages are defined in terms of fundamental patterns, and is similar to the package extension mechanism defined in [2]. Here, specialization of packages is shown by placing a UML specialization arrow between the packages. The child package specializes all (and therefore contains all) of the contents of the parent package.

Another important component of the MML is its core package, which defines the based modelling concepts used in UML: packages, classes and attributes. Currently, the MML model concepts package does not provide a dynamic model of behaviour. Thus, in order to define a semantic model for the action semantics in the MML, an extension must be made to the core MML package. This is described in the following sections.

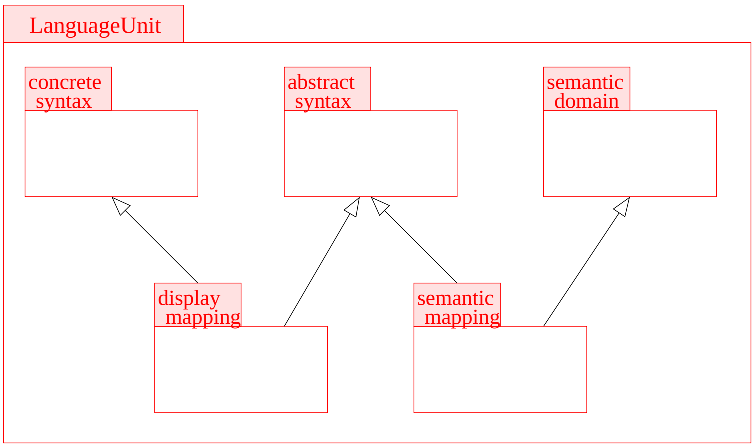

Fig. 1. The MMF Method

2 Principles of the new action semantics

Two basic goals have led to the redefinition of the action semantics. The first one is to include the action semantics as the dynamic core of MML, and possibly substitute the static core. This implies the definition of model and instances views and separation between concepts and syntax. The second goal is to have this action semantics as simple as possible and as easy to extend and specialize as possible. For the goal of simplicity, it is necessary to define as few new concepts as possible. One of the ways to do this is to reuse whenever possible the concepts already defined in the other packages of MML. It is also important to abstract out the fundamental concepts common to all actions and to be as removed as possible from the implementational aspects of actions. MML is designed to be easily extensible, as will the action semantics if we include it as another part of MML. As the actions semantics will be another package in MML, it has to follow the structure of the rest of packages, that is, the package should be composed of model, instance and semantics packages, with the model and instance packages further divided in packages for concepts, syntax and the mapping between concepts and syntax.

2.1 New basic concepts

The dynamic core tries to model the evolution of the values of the objects in the system with time. This is in contrast with the view of the static model that considers instances to be attached to a single value. The approach taken to define the dynamic model considers a history as a sequence of values, often called snapshots, being the execution of the actions responsible for the progression from one value to the next. A snapshot can be related to the whole system, as it is in the first approach to actions in the MMF document, or to a single object as it is in this approach. Only those acts causing the change of a value will be considered to be actions. For example, to write a new value in an instance slot will be an action as the value of the instance slot is different before and after the execution of the action. However, the reading of the value of a variable will not be considered as an action as no element in the system changes its value. These kind of acts will be considered to be expressions. In the current definition of MML, expressions are defined to model the OCL. There is also a subclass of expression called method, which is used to model the static methods of classes. Thus, the basic notion of an action is of a model element that relates a series of values with a series of elements. These input values will be used in the action execution to update some of the values of the elements associated with the action. The specific semantics of every action will be described in every subclass action in terms of its class diagram and well-formedness rules. Unlike in the previous proposal for action semantics, there is no concept of action execution history with a step for every state in the execution. In this model, the action execution is simply the occurrence of the action. If it is a compound action, it can be decomposed into simpler actions.

2.2 Time

Time was introduced in the previous action semantics definition to define a timing order in the dynamic model. In our point of view, time is not a concept general to every type of systems, so will not be used in the basic dynamic model. However, particular systems as real-time systems can easily extend this model to cope with the notion of time as best fitted to its purposes.

3 Actions

An Action represents a computational procedure that changes the state of an element in the system. In order to execute, an action requires some input values that will be used to compute the new values for other elements in the system.

Methods in MML are also used to define computational procedures. Methods are side-effect free - they simply evaluate a set of parameters against an OCL expression to obtain a result. Any method can be defined just by changing the body expression.

Actions are not side-effect free since they change the value of an element. Actions will not have a body expression that specifies what the action does. However, new action classes that specialize the basic Action class will be defined. Their particular behaviour will be described by means of well-formedness rules.

With this approach, a new action cannot be defined by just changing the expression that defines it, but there is a set of standard basic actions on top of which new actions are constructed.

The actions package specializes the staticCore package.

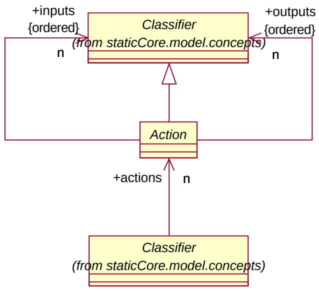

Fig. 2. actions.model.concepts package

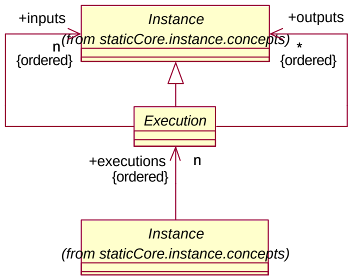

Fig. 3. actions.instances.concepts package

3.1 Concepts

The abstract class Action specializes Classifier. Every action has a set of input parameters and can produce output values. As the order of the parameters and results is significant, these associations are ordered. An action will be executed on the behalf on an object. Therefore there is another association between Action and Class, describing the Class that the host object belongs to.

Methods

[1] The method allActions() returns the set of all actions of Class, including those of its parents.

```
uml.staticCore.model.concepts.Class
allActions() : Set(Action)
  parents->iterate(p s = actions | s -> union(p.allActions()->
  reject(c | actions->exists(c' | c'.name = c.name)))
```

[2] This method returns the set of immediate subactions of this action. The actual set returned is defined in concrete descendants of Action.

```
context uml.actions.model.concepts.action
subactions() : Set(Action)
```

[3] This method returns all subactions of an action, nested to any depth.

```
context uml.actions.model.concepts.action
allSubactions() : Set(Action)
  self.subactions()->
  union(self.subactions().allSubactions()->asSet)
```

[4] This method returns true if the action is a subaction, at any depth, of another given action.

```
context uml.actions.model.concepts.action
```

```
isSubactionOf(otherAction: Action):Boolean
  otherAction.allSubactions()->includes(self)
```

3.2 Instances

The Execution class is defined in the Instances package. An Execution instance represents the actual execution of an action. The execution is associated with the actual inputs values used to execute the action and the actual output results.

Though there is no notion of time, and consequently, time ordering in the dynamic model, there is a causal relationship between the execution of the action and the values used in it. As the action cannot execute until all the input values are calculated, the values after the action execution cannot be accessed for the calculation of the input parameter values.

An Execution also has an association with the object on whose behalf it is executed.

Well-formedness Rules

[1] A calculation used to generate a parameter value for an execution cannot access an instance that is the output of that execution or follows that output in the element history.

```
context uml.actions.instance.concepts.Execution inv:
  To be formalized.
```

4 Primitive Actions

A primitive action is an action that cannot be decomposed into simpler actions. There are several subclasses of primitive action, each tailored to the kind of elements they act upon: null action, variable actions, object actions and slot actions.

These actions specializes the Action class. Their different behaviour will be determined by means of well-formedness rules.

Well-formedness rules

[1] A primitive action has no subactions.

```
context uml.primitiveActions.model.concepts.PrimitiveAction
subactions(): Sequence(Action)
  Set{}
```

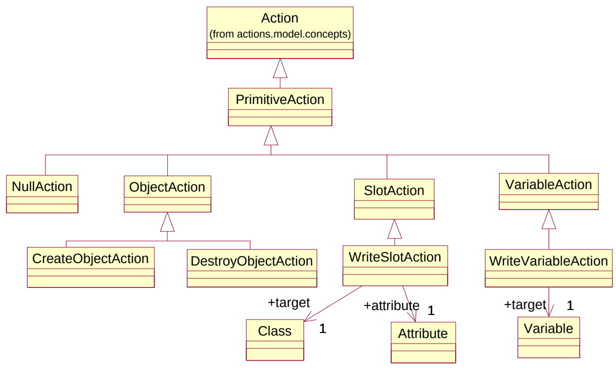

Fig. 4. primitiveactions.model.concepts package

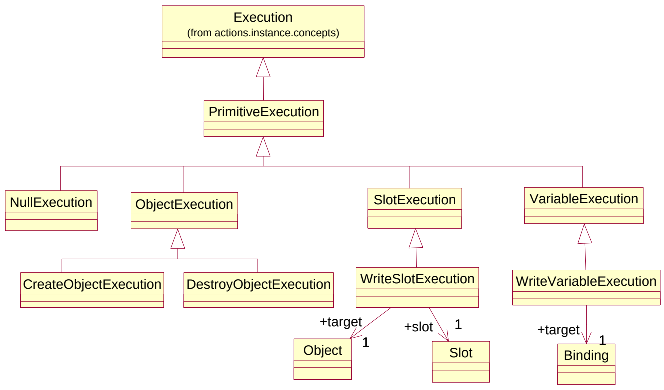

Fig. 5. primitiveactions.instances.concepts package

4.1 Null Action

The null action, as its name states, makes no change to the system. It is included because the compound actions have to have at least one subaction.

Well-formedness rules

[1] A NullAction has no input nor output elements.

```
context uml.primitiveActions.model.concepts.NullAction inv:
  self.inputs->size = 0 and
  self.outputs->size = 0
```

[2] A NullExecution has no input parameters nor output values.

```
context uml.primitiveActions.model.concepts.NullExecution inv:
```

```
self.inputs->size = 0 and
self.outputs->size = 0
```

4.2 Variable Actions

The only possible action on variables is to write a new value to that variable. A variable must be accessible from the action so that it can be used. Accessibility will be discussed in the group actions section.

The variable whose value is updated is the target of the action. The only input parameter to the action is the value to be assigned to the variable.

The input value for the execution of a *WriteVariableAction* will be the same as the value of the variable after the execution of the action.

Well-formedness rules

[1] A WriteVariableAction has a single input parameter.

```
context uml.primitiveActions.model.concepts.WriteVariableAction inv:
  self.inputs->size = 1
```

[2] The type of the updated variable must be the same as the type of the input value.

```
context uml.primitiveActions.model.concepts.WriteVariableAction inv:
  self.inputs->at(1).oclIsKindOf(self.target)
```

[3] The variable must be accessible by the action.

```
context uml.primitiveActions.model.concepts.WriteVariableAction inv:
  self.target.isAccesibleBy(self)
```

[4] A WriteVariableExecution has a single input parameter.

```
context uml.primitiveActions.instance.concepts.WriteVariableExecution inv:
  self.inputs->size = 1
```

[5] The value of the updated variable after a WriteVariableExecution is the input value.

```
context primitiveActions.intance.concepts.WriteVariableExecution inv:
  self.inputs->at(1) = self.outputs->at(1)
```

[6] The output value of the execution belongs to the history of the variable.

```
context primitiveActions.intance.concepts.WriteVariableExecution inv:
  self.target.history->includes self.outputs->at(1)
```

Methods

[1] This method checks whether the given action is within the scope of this variable.

```
context uml.constraints.model.concepts.Variable
isAccesibleBy(a : Action) : Boolean
  self.scope.subactions()->include(self)
```

4.3 Object Actions

An object cannot change its value, but it can be dynamically created and destroyed. There are two actions for Objects, one to create a new object of a given class and another one to destroy an object. Some languages include dynamic object typing, allowing the Class of an object to be changed in execution time. If the reclassify action is to be included in the action semantics, then when an object is reclassified as belonging to another class, the element in the system whose value changes must be known. The reclassify action is not considered further here.

Both the create and destroy object actions have a single input parameter, which is the class the object belongs to. Neither of them have output values.

The execution of a create object action has a result, the created object. This action does not execute any further job, that is it does not execute any initialization on the slots of the new object. The execution of a destroy object action has no output values. An element value cannot be accessed by any other action execution or expression calculation after it has been destroyed.

Well-formedness rules

[1] The element created by a CreateObjectAction must be an object.

```
context uml.primitiveActions.model.concepts.CreateObjectAction inv:
  self.outputs->size = 1 and
  self.target.oclIsKindOf(Class)
```

[2] A CreateObjectAction has a single input.

```
context uml.primitiveActions.model.concepts.CreateObjectAction inv:
  self.outputs->size = 1
```

[3] The input of a CreateObjectAction is the class of the created object.

```
context uml.primitiveActions.model.concepts.CreateObjectAction inv:
  self.inputs.olcIsKindOf(Class)
```

[4] The element destroyed by a DestroyObjectAction must be a object.

```
context uml.primitiveActions.model.concepts.DestroyObjectAction inv:
  self.outputs->size = 1 and
  self.outputs.oclIsKindOf(Class)
```

[5] A DestroyObjectAction has a single input.

```
context uml.primitiveActions.model.concepts.DestroyObjectAction inv:
  self.inputs->size = 1
```

[6] A CreateObjectExecution has no prevalues and one postvalue.

```
context uml.primitiveActions.instance.concepts.CreateObjectExecution inv:
  self.inputs->size = 0 and
  self.outputs->size = 1
```

[7] A CreateObjectExecution has a single input parameter.

```
context uml.primitiveActions.instance.concepts.CreateObjectExecution inv:
  self.parameter->size = 1
```

[8] The output of a CreateObjectExecution is a new object of the specified class.

```
context uml.primitiveActions.instance.concepts.CreateObjectExecution inv:
  self.outputs->at(1).isTypeOf(self.value)
```

[9] A DestroyObjectExecution has one input value and no output values.

```
context uml.primitiveActions.instance.concepts.DestroyObjectExecution inv:
  self.inputs->size = 1 and
  self.outputs->size = 0
```

[10] A DestroyObjectExecution has a single input parameter.

```
context uml.primitiveActions.instance.concepts.DestroyObjectAction inv:
  self.inputs->size = 1
```

[11] The input value of a DestroyObjectExecution is an object.

```
context Class uml.primitiveActions.instance.concepts.DestroyObjectExecution
inv:
  self.inputs->at(1).isTypeOf(Object)
```

[12] An object cannot be accessed by any other action execution or expression calculation after it has been destroyed.

```
context uml.primitiveActions.instance.concepts.DestroyObjectAction inv:
  To be formalized
```

4.4 Slot Actions

As for variables, the only available action on slots is to write a new value on them. In MML, an attribute can be of any type including Classifiers. Therefore, the allowable actions on slots should be the same as those on instances of the classifier to which the attribute belongs.

Well-formedness rules

[1] A WriteSlotAction has a single input parameter.

```
context uml.primitiveActions.model.concepts.WriteSlotAction inv:
  self.inputs->size = 1
```

[2] The attribute must be a valid attribute for the class.

```
context uml.primitiveActions.model.concepts.WriteSlotAction inv:
  self.target->attributes->include(self.attribute)
```

[3] The type of the updated slot must be the same than the type of the input value.

```
context uml.primitiveActions.model.concepts.WriteSlotAction inv:
  self.inputs->at(1).oclIsKindOf(self.slot)
```

[4] A WriteSlotExecution has a single input parameter.

```
context uml.primitiveActions.instance.concepts.WriteSlotExecution inv:
  self.inputs->size = 1
```

[5] The slot must be a valid slot for the object.

```
context uml.primitiveActions.model.concepts.WriteSlotExecution inv:
  self.target->slots->include(self.slot)
```

[6] The value of the updated variable after a WriteSlotExecution is the input value.

```
context primitiveActions.intance.concepts.WriteSlotExecution inv:
  self.inputs->at(1) = self.outputs->at(1)
```

[7] The output value of the execution belongs to the history of the variable.

```
context primitiveActions.intance.concepts.WriteSlotAction inv:
  self.target.history->includes self.output->at(1)
```

5 Compound Actions

In contrast to the primitive actions, other actions are complex and they can be divided into subactions. There are three types of compound actions, group actions, conditional actions and loop actions, each of them defining one of the basic language constructors.

Well-formedness rules

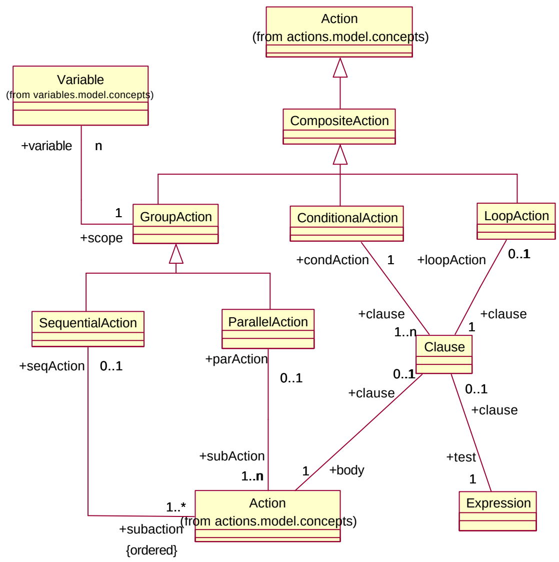

Fig. 6. compositeactions.model.concepts package

[1] A composite action does not update any value.

```
context uml.compositeActions.model.concepts.compositeAction inv:
    self.target->size = 0
```

[2] A composite action has no input parameters.

```
context uml.compositeActions.model.concepts.compositeAction inv:
    self.parameter->size = 0
```

[3] A composite execution has neither a prevalue nor a postvalue.

```
context uml.compositeActions.model.concepts.compositeAction inv:
    self.preValue->size = 0 and
    self.postValue->size = 0
```

[4] A composite execution has no input parameters.

```
context uml.compositeActions.instance.concepts.compositeExecution inv:
    self.parameter->size = 0
```

5.1 Group Actions

A group action is simply intended to give a scope to a number of subactions. The scope includes the variables the actions have access to and the kind of ordering

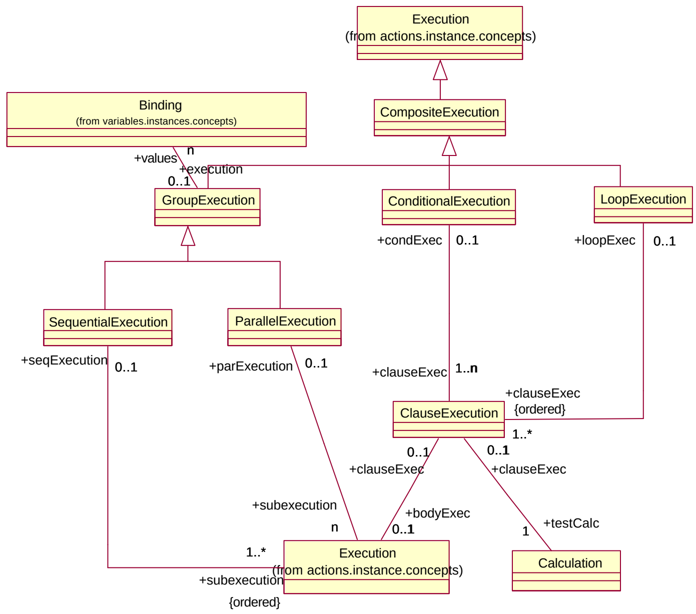

Fig. 7. compositeactions.instances.concepts package

in the action execution. A group action is associated with a number of variables. These variables are accessible to all the subactions in the group action and not accessible by any action outside of it.

There are two types of group action: sequential actions and parallel actions. The subexecutions in a sequential execution have to execute in the order that they are included in the corresponding sequence. This implies that an execution, or a value calculation for that execution cannot access a value modified by a previous execution.

In a parallel execution all the subexecutions can happen concurrently and the same value can be accessed to calculate values for different execution, but the value can only be modified by a single execution.

Well-formedness rules

[1] Subactions executions in a group execution happen sequentially.

```
context uml.compositeActions.instance.concepts.sequentialAction inv:
  To be formalized.
```

Methods

[1] The subactions of a parallel action are its subactions.

```
context uml.compositeActions.model.concepts.parallelAction
subactions(): Set(Action)
  self.subAction
```

[2] The subactions of a sequential action are its subactions.

```
context uml.compositeActions.model.concepts.sequentialAction
subactions(): Sequence(Action)
  self.subaction
```

5.2 Clauses

A clause comprises a test and an body. The test is an expression that returns a boolean value and the body is an action.

When a clause executes, the test is always calculated, but the action may not execute, depending on the context of the clause and the value yielded by the test.

5.3 Conditional Actions

A conditional action is composed of a set of clauses. In the execution of a conditional action, all the clause tests are calculated concurrently. As they are side effect free, the order is not relevant to the final calculations. Only one of the clauses whose test has yielded true in its calculation is chosen to execute its action.

Well-formedness rules

[1] The subactions of a conditional action are its body subactions.

```
context uml.compositeActions.model.concepts.conditionalAction
subactions(): Set(Action)
  self.clause.body
```

[2] A conditional execution has as many test calculations as its corresponding action has text expressions.

```
context uml.compositeActions.instance.concepts.conditionalExecution inv:
  self.clauseExec->forAll(c | c.testCalc->size = 1)
```

[3] A conditional execution only executes one of its clause's bodies, and the test for that clause must yield `true`.

```
context uml.compositeActions.instance.concepts.conditionalExecution inv:
  self.clauseExec->collect(c | c.bodyExec->size = 1)->size = 1
  and
  self.clauseExec->forAll(c | c.bodyExec->size = 1 implies
  c.testCalc = true)
```

5.4 Loop Actions

A loop action has a single clause with its test expression and body action. The loop execution consists of successive clause calculations. For a clause calculation, the test is evaluated and if it yields true, the action is executed. After the execution of action, there is another clause calculation. When the test yields false in a clause calculation, the action is not executed and the loop execution is terminated.

Well-formedness rules

[1] The only subaction of a loop action is its body action.

```
uml.compositeActions.model.concepts.loopAction subactions(): Set(Action)
  self.body
```

[2] A loop execution has one test calculation more than body executions.

```
context uml.compositeActions.instance.concepts.loopExecution inv:
  self.testCalc->size = self.bodyExecution->size + 1
```

[3] The test only yields `false` on the last iteration of a loop execution.

```
context uml.compositeActions.instance.concepts.loopExecution inv:
  self.clauseExec->subSequence(1, self.clauseExec->size - 1)
  ->forAll(c | c.testCalc = true) and
  self.clauseExec->at(self.clauseExec->size).testCalc = false
```

6 Changes in current MML

Some changes to the current definition of MML have to be done to include the dynamic model.

In first place, an instance is no longer related to a single value, but to a number of values, each of them denoting the new value after the execution of an action on the instance. To include this new point of view, in the instance package of the staticCore, the association between Instance and Value is replaced by a one to one association between Instance and a new class, History. The History is in turn associated with a number of values that reflect the successive changes on the instance.

The current concept of Expression only covers those used for OCL. It has to be extended to include subclasses that calculate non- boolean values or return the value of elements in the system that actions need to execute. The reading of the value of a variable is an example of these new classes.

7 Conclusions and Future Work

In this paper we have proposed a restructured meta-model architecture for the action semantics, based on the meta-modelling language (MML). The focus of the work has been to develop a consistent approach to constructing the meta-model, in which abstract syntax and semantics are clearly delineated. The result is a definition that supports a simple methodological approach to extension. Whenever a new action is added to the abstract syntax of the language, a structure preserving addition is made to the semantic domain. This *pattern* of construction is a vital weapon in the meta-modellers armory, and is essential if large meta-models of language are to be constructed successfully.

With respect to other work that has been done in this area, Kleppe and Warmer [4] have concurrently a similar approach to refactoring the action semantics, based on the MML. Our work differs from theirs in that we have attempted to define a more generic model of behaviour, which makes no assumptions about the ordering of actions. Thus, our model is more adaptable, and could form the foundation of a number of different languages with alternative semantic models. This ability is essential in order to support families of related UML action languages.

Clearly, there are a number of future directions of work possible. In particular, the development of additional meta-modelling patterns is currently impacting on the MML [6]. These patterns provide a number of fundamental structures that are commonly repeated across language definitions. For example the container/contained pattern is commonly found in many language models, as are patterns which describe properties of generalisable modelling elements and instansiable elements. It is expected that a refactoring of the meta-model using these patterns would further improve its structure and clarity.

In terms of action semantics, issues relating to time are an important and necessary extension to the work. Time can be added as a variable to an execution or an object, and there is significant work in the formal languages field to draw on in deciding on the best semantic model. Once this has been incorporated fundamental issues of proof and refinement can be explored within a concurrent and real-time context.

References

1. Action Semantics Consortium: Response to OMG RFP ad/98-11-01. Action Semantics for the UML. Revised September 5, 2000 (2000) www.umlactionsemantics.org
2. D'Souza D., Wills A. C.: Object Components and Frameworks with UML – The Catalysis Approach. (1998) Addison-Wesley.
3. Clark T., Evans A., Kent S., Brodsky S., Cook S.: A Feasibility Study in Rearchitecting UML as a Family of Languages using a Precise OO Meta-Modeling Approach. (2000) www.puml.org
4. Kleppe A., Warmer J.: Integration of static and dynamic core for UML: A study in dynamic aspects of the pUML Object-Oriented meta modelling approach to the rearchitecting of UML, (2001) TOOLS Europe 2001
5. Brodsky S., Clark A., Cook S., Evans A., Kent S. (2000) A feasibility Study in Rearchitecting UML as a Family of Languages Using a Precise OO Meta-Modeling Approach. Available at `http://www.puml.org/mmt.zip`.
6. Clark A., Evans A., Kent S. (2000) Engineering Modelling Languages: A Precise OO Meta-Modeling Approach. Available at `http://www.puml.org/mml`