RC 22132 (W0107-25) 30 July 2001 **Computer Science**

IBM Research Report

A Test-Ready Meta-Model for Use Cases

Clay E. Williams
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g. , payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

A Test-Ready Meta-model for Use Cases

Clay E. Williams

IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598, USA
clayw@us.ibm.com

Abstract: In the UML, use cases are used to define coherent units of functionality associated with classes, subsystems, or systems. Two principal purposes that use cases serve are specifying the functionality the system will provide and providing a basis for developing test cases for the system being specified. This paper discusses issues that arise when using use cases as the basis for model-based testing. A basic use case meta-model is developed that is consistent with the UML 1.3 specification but based on meta-modeling techniques developed to provide precise semantics for modeling languages. This meta-model is then extended to derive a test enabled meta-model for use cases. The paper closes by discussing future research.

1 Introduction

The Unified Modeling Language (UML) provides use cases as a mechanism for specifying the functionality that will be provided by a system. Use cases are useful for three purposes [BRJ99]:

(1) Use cases provide a mechanism for describing the system's functionality in a way that is understandable by domain experts.

(2) Use cases provide a starting point for developers to understand and implement the required functionality.

(3) Use cases serve as the basis for testing the system as it evolves.

In the first case, domain experts exploit use cases to ascertain that the system being constructed provides complete and correct functionality. No additional UML artifacts are created as a result of this step. In the second situation, use cases serve as the basis for deeper analysis and design, in which the use cases are realized using object-oriented methods. The realization process typically involves exploring a domain model developed during analysis and determining how each use case can be achieved using the classes from the model [Ja92]. The expansion from 'what' (use cases) to 'how' (collaborations of classes) requires a well-defined meta-model for classes, associations, and related concepts. Such models have received considerable attention, for example see [Cl00] and [Cl01]. The third advantage of using use cases is to aid in the development of test cases, which is unique among the three purposes listed above in that it is the only one whose goal is to utilize use cases to produce an artifact that is external to the UML. Thus, just as classes and their associated concepts require a well defined meta-model in order to produce sound code, use cases need a precise meta-model to produce valid and robust test cases. This paper explores the meta-modeling issues that arise when seeking to exploit use cases for testing purposes.

It is important to note that testing is an important and costly step in the software development process. Testing typically takes anywhere from 30%-50% of the total budget of a software project. Improvements in testing can lead to significant savings for a development organization. In fact, in most organizations, testing consumes a larger fraction of resources than the coding phase of the life cycle. Thus, automation of portions of the testing process offer some of the most significant opportunities for enhancing the efficiency of the development process.

The remainder of this paper proceeds as follows. First, I examine the basic requirements for test case generation from use cases. Next, I develop a generic meta-model for use cases based on concepts from the UML 1.3 meta-model. This meta-model is then extended with new testing related elements to develop a meta-model that is amenable for test

case generation. This extension shows the power and flexibility of developing UML as a family of modeling languages as proposed in [Cl00]. Finally, I present conclusions and opportunities for further work.

2 Basic Tenets for Testing with Use Cases

Before proceeding to develop a meta-model for use cases, it is important to step back and explore what is required for use cases to be used in conjunction with model-based testing techniques. These tenets arose as a result of developing a tool for use case based testing. This tools runs within one of the popular modeling tools and allows testers to add additional annotations to the use case model. The output of the tool is a suite of test cases optimized for coverage of the input parameters and test suite size. An early paper on the technique is [WP99]; the tool has been enhanced with additional capability based on the meta-model described in section 4.

2.1 Use Cases define Classifier Functionality

In [RJB00], a use case is described as a "coherent unit of functionality provided by a classifier." Thus, the first requirement for a meta-model for use case based testing is that it should support the notion that there is a classifier against which a use case is developed. The classifier is what is being tested, with the use cases describing how to test it. Rather than focusing on a use case as a classifier with attributes and operations, test generation is best served when use cases are considered as having the classifier being tested "behind" them, with this classifier being the repository of state. The use case is then a set of action sequences which involve either the classifier or actors associated with the use case. This viewpoint differs significantly from earlier attempts to formalize use cases [OP98][S01]. The justification for this "stateless" view of use cases is described below; however, the two viewpoints may be reconcilable. The basic approach to reconciliation is as follows: use cases could be considered to provide a projection onto a subset of the attributes of the classifier against which they are defined. Then, the action sequence notions are directly mappable to the labeled transition system formalism used in [S01].

In the description of the features that may belong to a use case, [RJB00] states that a use case may have attributes and operations. It is states that the attributes may represent the state of the use case, or progress of executing it. However, the state of the use case and the progress of executing it are two very different things. The state would simply be the aggregate values of any attributes belonging to the use case, whereas progress would indicate where we are (i.e. which action is the current action) in terms of executing an instance of the use case. For effective test generation, location in the action sequence of the use case is all of the state that we require.

From a testing viewpoint, the state based formalization of use cases [S01] raises questions of parsimony. The first issue is the definition of the state space S for a use case u. If u is considered to be associated with an entity with state space E and actors with state spaces $A_1,...,A_n$ u is constrained to include a quotient map, which is a surjection $h{:}E\times A_1\times...\times A_n\rightarrow S$. The problem is that the state space of the actors involved with a use case should be irrelevant to the use case, as the actor represents an entity external to the system's boundary. This issue is seen again in the notion of pre- and postconditions, which are defined as a relation $PP\subseteq(E\times A_1\times\ldots\times A_n)\times(E\times A_1\times\ldots\times A_n)$. Our experience in automated test case generation indicates that the state of the entity E is sufficient to correctly construct a precondition to determine when a use case may be invoked, thus capturing the state of the actors is unnecessary. Finally, [S01] discusses the need for formalizing notions of use case "interference." In the framework in which state is a property of the classifier being tested rather than the use case, interference can be defined as the situation in which different use case instances update the same attribute on the classifier. Given this definition, techniques based on data flow analysis can be used to generate interesting test cases based on interference.

A argument similar to that for attributes can be constructed concerning operations on use cases. [RJB00] states that "an operation represents a piece of work the use case can perform. It is not directly callable from the outside, but may be used to describe the effect of the use case on the system. The execution of the operation may be associated with the receipt of a message from an actor. The operations act on the attributes of the use case, and indirectly on the system or class that the use case is attached to." This quote points to several ambiguities concerning operations. First, they are not invocable from outside, but somehow respond to messages from actors. Second, their primary purpose is to act on the attributes of the use case, with changes to the classifier's attributes being indirect

effects. As our representation has no attributes beyond a current location in the action sequence for the use case, operations are unnecessary.

Thus, the first requirement on our meta-model is that the use case be a stateless entity (beyond the need to keep track of where we are within a use case instance) that describes a set of sequences of actions, each of which involves an actor or the classifier against which the use case exists. Extension and inclusion should be supported in the form defined in the 1.3 specification.

2.2 Use Cases consist of Actions

Since our representation of use cases is that of a sequence of actions without attributes or state, developing the appropriate action representation is essential. Action semantics have been considered in [ASC01] and [Al01]. The MML based semantics developed in [Al01] serve as a basis for action sequences within use cases.

As they relate to use cases, actions always describe one of two situations: interaction with an actor or the classifier to which the use case is associated. The actions need to be robust enough to specify what happens to the actor/classifier, without specifying how it happens. A key point is that actions should be simple to record in the use case, but regardless of the appearance they represent syntactic sugar for specifying interactions between actors and classifiers.

2.2.1 Actor Related Actions

There are three types of actions that a use case can specify concerning an actor. In all three, the entity implicitly involved is the classifier owning the use case, not the use case itself. The actions are: receiving input from an actor (Actor Input), writing output to an actor (Actor Output), and invoking a computation on an actor (Actor Computation). Computation differs from the others in that output is passed to the actor, resulting in a new set of input from the actor that is dependent on the original output. To illustrate the possible actions, consider the following fragment of a use case model shown in Figure 1.

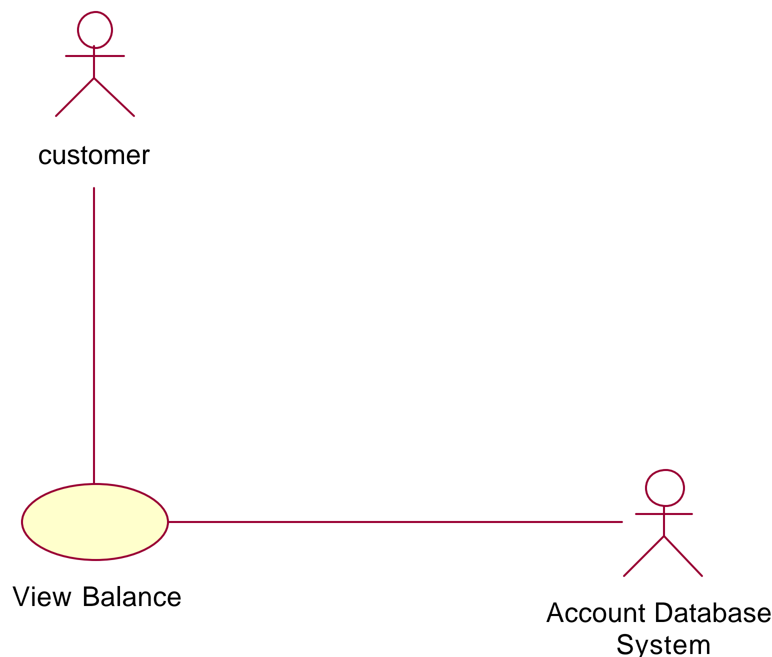

Figure 1. Actor related Actions.

In this instance, when the customer requests a balance inquiry from the ATM system, the ATM system sends a request to the Account Database Actor, passing it the customers account number as a parameter. The Account Database System actor returns the balance in the customers account. The balance is then displayed to the customer. All three of the actor related action types are used in this simple scenario, as shown in Table 1.

Step	Action	Action Type
1	Customer requests balance inquiry.	Actor Input
2	ATM issues request to account DB system. Balance is returned.	Actor Computation
3	ATM displays amount to customer	Actor Output

Table 1. Action Scenario illustrating Action Types

The meta-model for testing with use cases must support the three actions types (Actor Input, Actor Output, Actor Computation) between actors and the classifier.

2.2.2 Classifier Related Actions

A use case must also be capable of expressing actions that occur within the system as a result of an interaction with an actor. These actions specify state changes that the classifier undergoes as a result of the execution of the use case. The action language defined in [Al01] provides the necessary action types required to update the classifier state, so no further requirements for action types against classifiers are needed.

2.3 Actors drive Use Cases

In the later phases of testing a system, testers are often interested in executing sequences of use cases. These are selected because they represent interesting transactions that actors perform using the system. In order to specify these sequences, we require the notion of "flows" between use cases, where a flow defines that a test case should be considered in which one use case directly follows another. For example, suppose we have a set of use cases $U=\{u_1,u_2,\ldots,u_4\}$ and an actors *A*. If *A* was associated with u_1, u_2, u_3, and u_4, then the flows for *A* in which u_1 occurs first, followed by any number of either u_2 or u_3, followed by u_4 can be represented by the *use case flow graph* in Figure 2. Note that these edges are not associations, but temporal flows through the system performed by an actor. The meta-model should support the definition of a set of use case flow that can be associated with an actor.

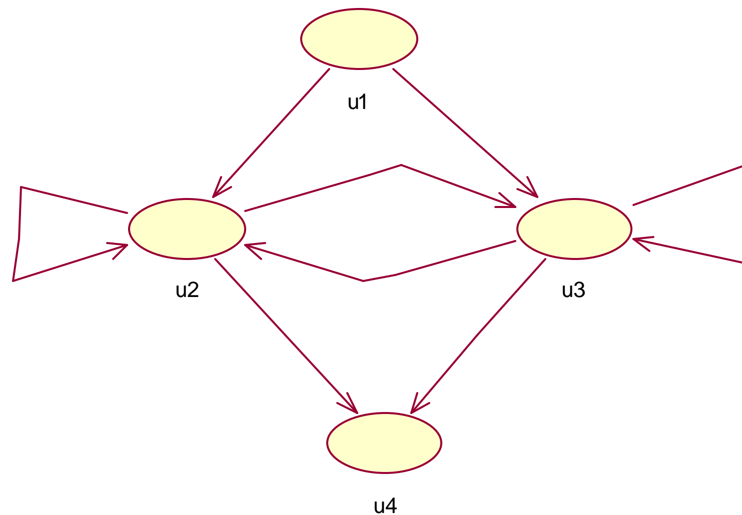

Figure 2. Use Case Flow Graph

2.4 Testing requires Test Data

The final set of requirements for a meta-model that supports testing is that a use case should be capable of specifying the inputs that flow from an actor to the classifier, as well as the output that flows the other way. The set of input data values required by a use case can be represented as *parameters,* which consists of *logical partitions* that categorize the actual test data [WP99][P00]. For example, if the parameter is **password**, the two logical partitions the parameter could have are **valid** and **invalid**. Given these partitions for the parameter, it is possible to associate actual data with them. For example, there will be one valid password for a given user, but there could be an infinite number of invalid passwords. A meta-model that supports testing from use cases will support the notions of parameters, logical partitions, and actual data that belongs to a given logical partition. Another requirement is that test data is often related in complex ways. The meta-model for test data needs to be capable of expressing these complex relationships, using entities such as collections and objects.

2.5 Requirements Summary

The following summarizes the requirements discussed in the above sections.

(1) Use cases should be represented as a sequence of actions written against a classifier.

(2) A set of three actor related actions should be defined for use cases: actor input, actor output, and actor computation.

(3) Transactions for actors should be definable based on sequences of use cases known as flows.

(4) The meta-model needs to incorporate a rich notion of test data involving parameters, partitions, and actual data. Complex constructs such as collections and objects should be supported.

3 A Use Case Meta-model

The meta-model is constructed using the MMF (Meta-modeling Facility) described in [Cl01]. The MMF method is performed using the Meta-modeling Language (MML), which is a language developed to enable UML to be re-architected as a family of modeling languages [Cl00]. MML has a well defined semantics provided using the ς-calculus [CEK01]. In the MMF method, MML is used to provide two types of mappings. The display mapping maps concrete syntax (human viewable layout of UML diagrams) to the abstract syntax (machine processable representation). The semantic mapping maps the abstract syntax to the semantic domain, which describes how model concepts are realized in model instances. Figure 3 [Al01] illustrates these relationships. Modeling using MML is supported by the meta-modeling tool (MMT.)

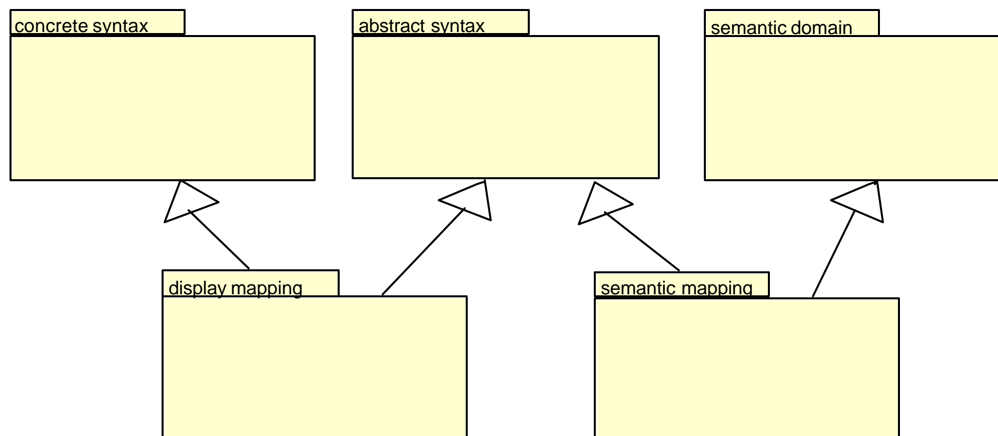

Figure 3. The MMF Method

The focus in this paper is the semantic mapping between abstract syntax and the semantic domain; I do not consider issues associated with concrete syntax and screen layout. The packages described below use elements from the mml package described in [Cl00] and the actions package described in [Al01]. Thus, they extend both of these models,

as is shown in Figure 4. The dashed arrows represent generalization rather than dependency. The testing package is shown outside of the UML package because it uses elements that are not part of standard UML. These are added as annotations to a standard UML model by the tester. This is an excellent example of the extensibility that comes from the meta-modeling approach of [Cl00]. The Object Constraint Language (OCL) [WK99] is use to specify additional constraints to guarantee that the meta-models are well formed.

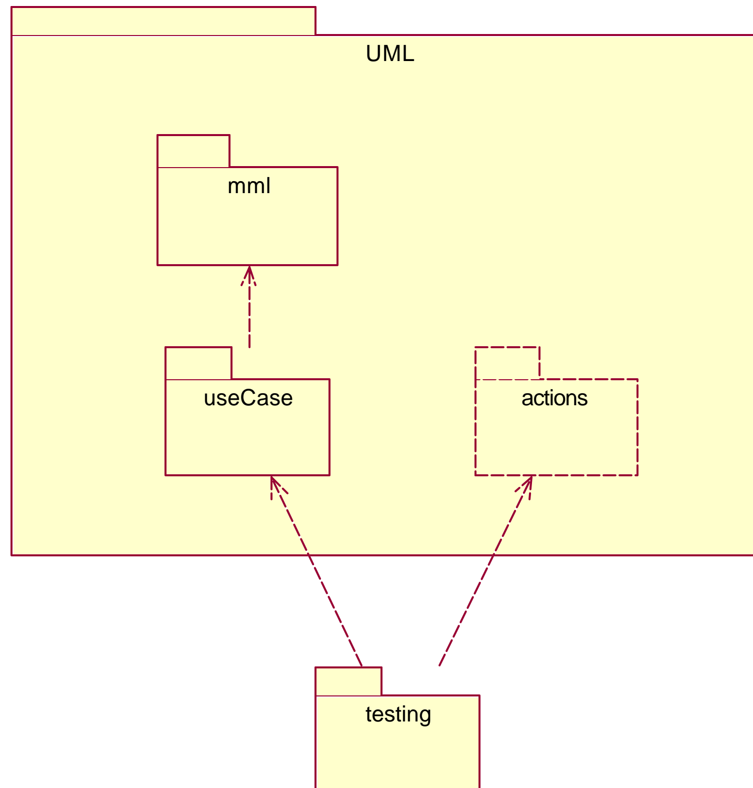

Figure 4. Generalization relationships between packages.

The use case package uses MMF techniques to describe the basic modeling structures required to build use case models. The basic functionality defined in UML 1.3 is provided. The key elements in developing use case models are use cases and actors, which are related by associations (see Figure 5).

3.1 uml.useCase.model.concepts

Use cases are generalizations of Classifier, and as such may be related to other use cases via generalization. Generalization between use cases will not be covered in this paper, but deserves further study. The basic ideas involving generalization based on the points of views of the actors involved [S01] is the approach that would be most amenable to the meta-model defined here, with the appropriate changes to allow classifiers to be the holder of state. Several issues remain to be explored, [S01] provides an excellent overview of the challenges. In practice, generalization has been used with our test case generation approach, but always from abstract to concrete use cases. Thus, the semantic requirements are less stringent. The other relationships between use cases are the ≪extend≫ and ≪include≫ relationships. These are addressed in the meta-model, as shown in Figure 6.

The ≪extend≫ relationship is based upon the description given in [RJB00]. In [RJB00], ≪extend≫ relationships are composed of three entities: the base use case, the extension use case, and the extends relationship between them. The base use case defines a set of extension points, and each extension point corresponds to a location in the base use case where additional behavior can be added. The extend relationship refers to these extension points as a set of

names. The relationship also has associated with it a constraint, which is a condition that must be true for the extension to take place. The constraint will be based on either the classifier or an input or result from an actor associated with the use case. The extension use case consists of a set of insertion segments, each of which specifies a sequence of actions that can be inserted when the segment is included in the base use case via extension.

The ≪include≫ relationship is similar to the extend relationship, but simpler. Inclusion always occurs at the point specified, and all of the behavior is included rather than just an insertion sequence.

Actors comprise the other key element for developing use case models. They are related to use cases via associations. The representation for these elements is shown in Figure 6.

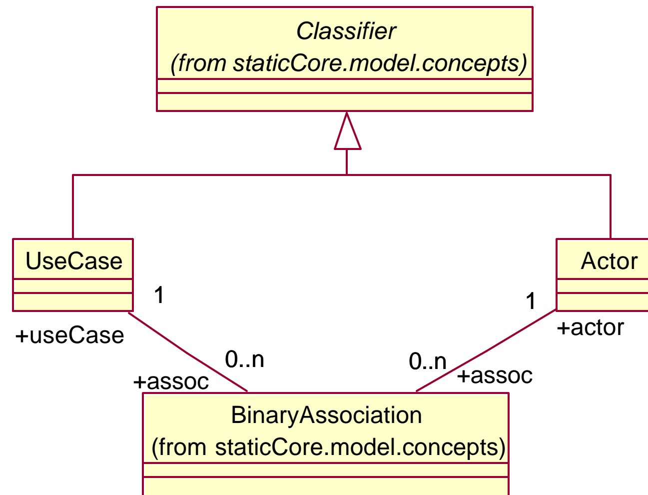

Figure 5. Use Cases and Actors in the uml.useCase.model.concepts Package

Well-formedness rules

[1] An ≪extend≫ relationship must be between distinct use cases.

```
context uml.useCase.model.concepts.Extend inv:
   not(self.base = self.extension)
```

[2] An ≪include≫ relationship must be between distinct use cases.

```
context uml.useCase.model.concepts.Include inv:
   not(self.base = self.addition)
```

[3] Actors cannot contain other classifiers.

```
context uml.useCase.model.concepts.Actor inv:
   self.elements->isEmpty
```

[4] The elements of a use case contains its extension points.

```
context uml.useCase.model.concepts.UseCase inv:
   elements->exists(g |
         g.name = "extPt" and g.elements = extensionPts())
```

[5] The elements of a use case contain its insertion segments.

```
context uml.useCase.model.concepts.UseCase inv:
   elements->exists(g |
         g.name = "segments" and g.elements = segments())
```

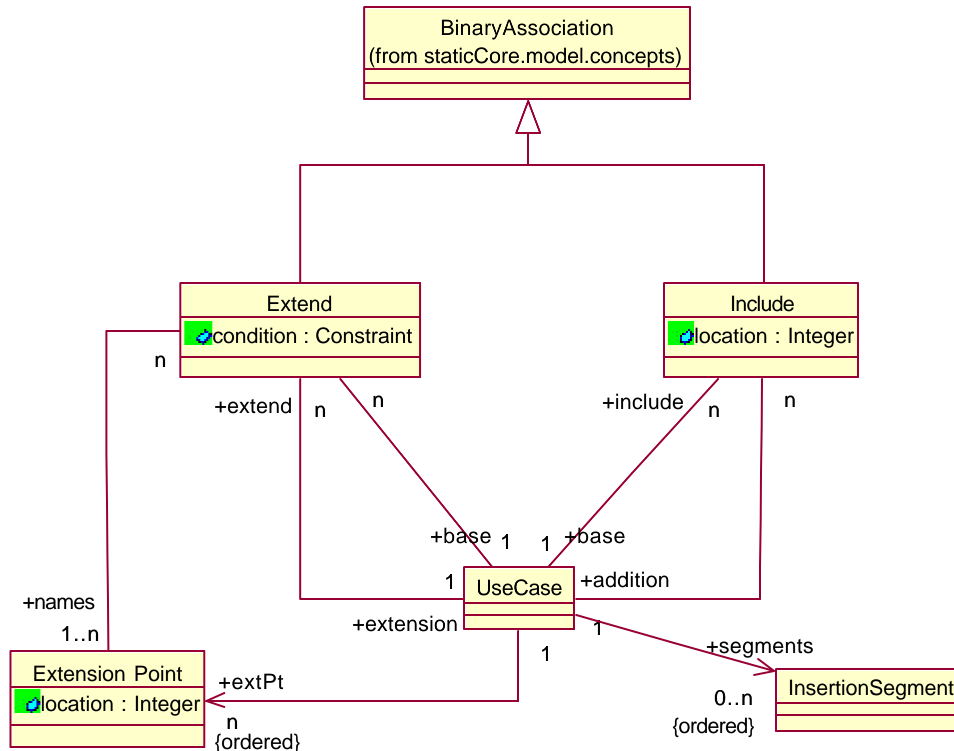

Figure 6. Actors and their Relationships in the uml.useCase.model.concepts Package

Methods

[1] The method extensionPts() returns a set containing all of the extension points of a use case.

```
context uml.useCase.model.concepts.UseCase
   extensionPts() : Set(ExtensionPoint)
   extPt->iterate(p s = Set{} | s->union(p))
```

[2] The method segments() returns a set containing all insertion segments of a use case.

```
context uml.useCase.model.concepts.UseCase
   segments() : Set(InsertionSegment)
   segments->iterate(p s = Set{} | s->union(p))
```

3.2 uml.useCase.instance.concepts

Use case instances are specializations of Instance. They are related to one another using instances of extend and include relationships. These relationships are shown in Figure 7. Actor instances are related to use case instances via association instances, as shown in Figure 8.

Well-formedness Rules

[1] The extend relationship must be between distinct use case instances.

```
context uml.useCase.instance.concepts.ExtendInstance inv:
   not(self.base = self.extendUCInstance)
```

[2] The include relationship must be between distinct use case instances.

```
context uml.useCase.instance.concepts.IncludeInstance inv:
   not(self.base = self.includeUCInstance)
```

[3] The number of extension point instances associated with an extend instance must be equal to the number of insertion blocks on the extending use case.

```
context uml.useCase.instance.concepts.ExtendInstance inv:
   self.extensionPoint->size = self.extendUCInstance.blocks->size
```

[4] Use case instances contain their insertion blocks.

```
context uml.useCase.instance.concepts.UseCaseInstance inv:
   elements->exists(g |
         g.name = "segments" and
         g.elements = blockSet())
```

[5] Use case instances contain their extension point instances.

```
context uml.useCase.instance.concepts.UseCaseInstance inv:
   elements->exists(g |
         g.name = "extPt" and
         g.elements = extensionSet())
```

Methods

[1] The blockSet method returns a set containing the insertion blocks on a use case instance.

```
context uml.useCase.instance.concepts.UseCaseInstance
   blockSet() : Set(InsertionBlock)
   blocks->iterate(b s=Set{} | s->union(b))
```

[2] The extesionSet method returns a set containing the extension point instances for a use case instance.

```
context uml.useCase.instance.concepts.UseCaseInstance
   extensionSet() : Set(ExtensionPointInstance)
   extensionPt->iterate(e s=Set{} | s->union(e))
```

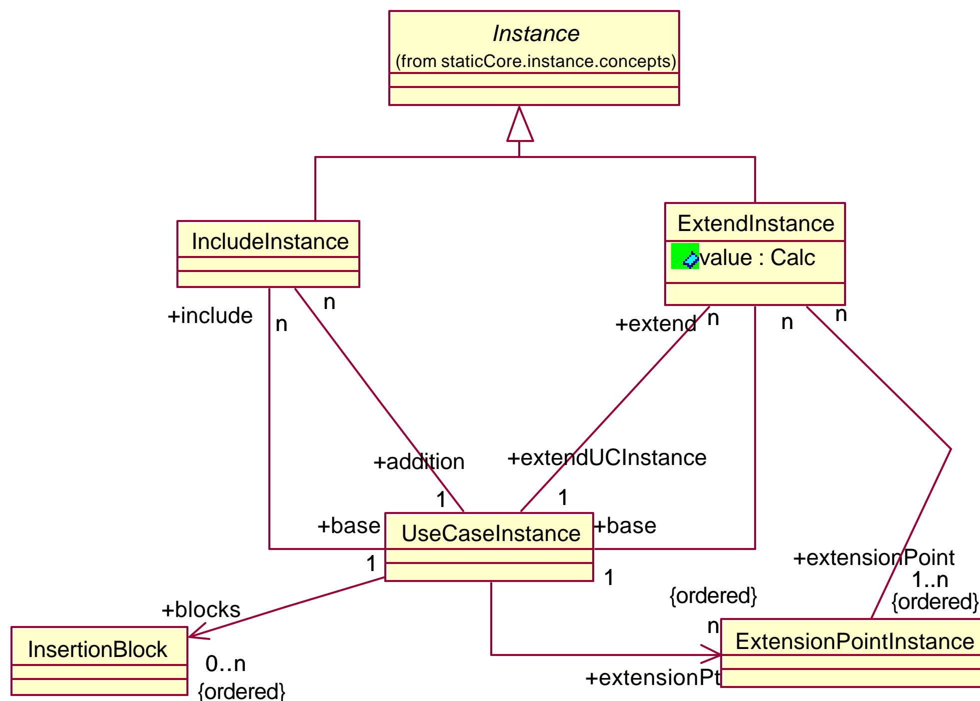

Figure 7. Use Case related Classes in the uml.useCase.instance.concepts Package

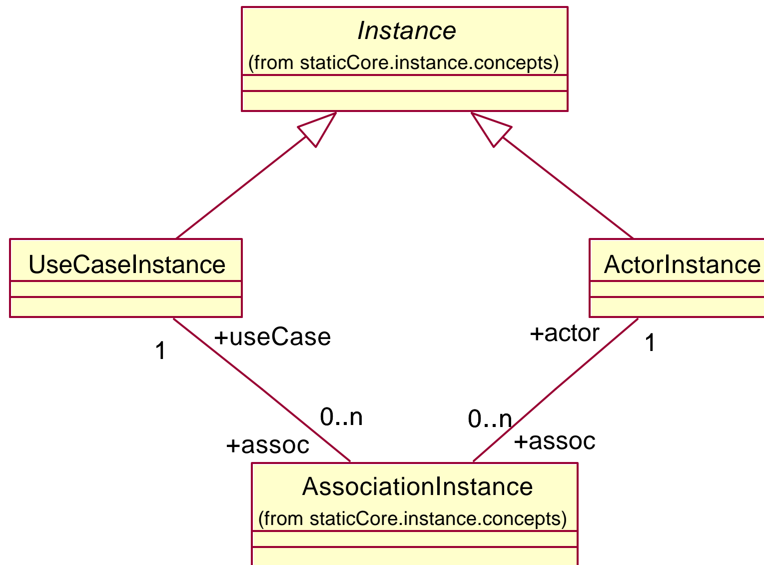

Figure 8. Actor related classes in the uml.useCase.instances.concepts Package

3.3 uml.useCase.semantics

The semantics for the use case package is defined in the useCase.semantics package, which extends both the useCase.model.concepts and useCase.instance.concepts. Semantic relationships are developed by defining a mapping between the model concepts and instance concepts. These are shown in Figure 9.

Well-formedness Rules

[1] Abstract use cases cannot be instantiated.
`Inherited from Classifier.`

[2] Instances must satisfy the properties of their classifier.
`Inherited from Instance.`

[3] The value for an extends instance is calculated correctly for the constraint.
`Inherited from Instance`

One area that requires additional formalization is how extension and inclusion points are correctly mapped from location references to precise points (as an integer) in a sequence of actions. The basic idea is that a sequence of actions exists, with extension points containing location references to places between actions. However, some of these actions may be composite, so an instance of the use case would not necessarily have references in exactly the same order. However, they would increase monotonically in both the use case and its instance. Further work along the lines explored in [S01] could formalize this further in the meta-model.

Figure 9. uml.useCase.semantics Package

4 The testing Package

The testing package extends both the useCase package defined above, as well as the action package presented in [Al01]. The concepts in the testing package are based on work done to explore automatically generating test cases from use cases, early versions of which are described in [WP99]. The requirements outlined in section 2 above will be addressed in the meta-model defined in this package.

4.1 testing.model.concepts

The following formalizations are used to meet the requirements necessary to utilize use cases for testing.

(1) Make actions explicit as constituents of use cases.

(2) Emphasize that use cases express functionality for a classifier

(3) Add pre- and postconditions to use cases.

(4) Enhance action language to support use case specification of communication between actors and classifiers.

(5) Support the notion of flows through use cases for actor based testing.

(6) Support the addition of parameters and partitions for input from actors.

(7) Support associations with test data.

Figure 10 shows the enhancements to the use case portion of the meta-model based on formalizations 1-3 described above.

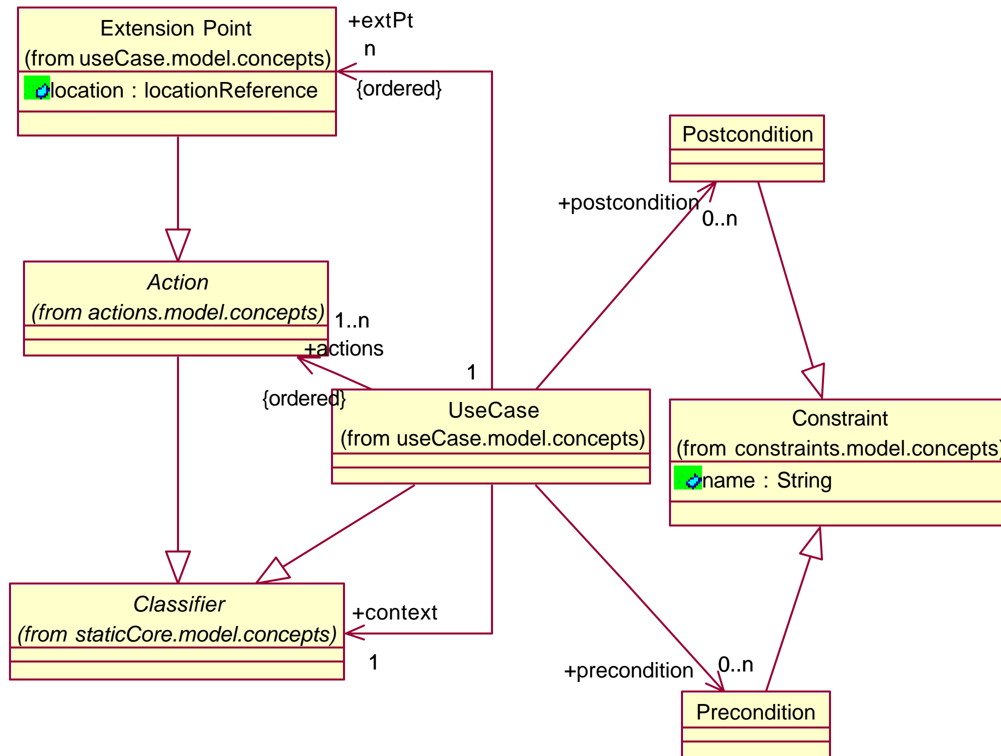

Figure 10. Additions to the Use Case meta-model in the testing.model.concepts Package.

Each use case consists of an ordered sequence of actions, and is associated with a classifier, which provides the context for that use case's actions. Pre- and postconditions have been added, and are constraints written against the context of the use case. In the same spirit as [OP98], extension points are defined as a subclass of Action. This provides consistency with the notion that a use case is simply a sequence of actions.

The actor related portions of the meta-model are also enhanced in the testing package, as shown in Figure 11. Actors are associated with flows through use cases, which defined sequences of use cases that are interesting from a testing point of view. Although it is not shown here, flows specialize Classifier. Actors are defined to communicate with classifiers via message related actions, which are shown in Figure 12.

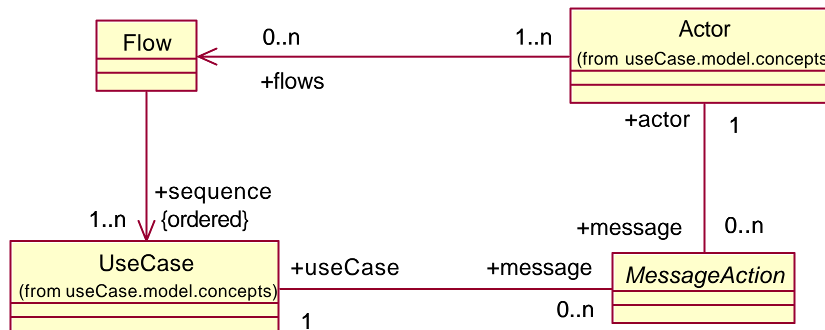

Figure 11. Flows and Messages between Actors and Use Cases.

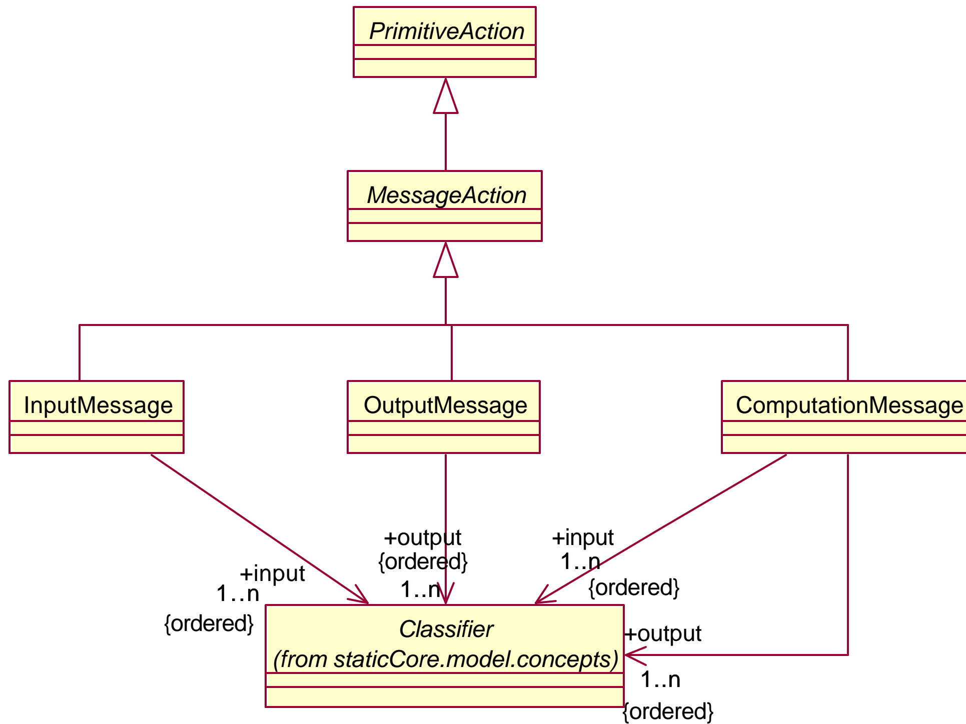

Figure 12. Message Actions in the testing.model.concepts Package

The diagram in Figure 13 formalizes the notions of parameters, partitions, and their associated test data. Test data can be structured in any form, as it is a specialization of classifier. Associated with each use case are parameters which flow from outside of the system (via actors) into the classifier. These parameters are ordered, and each parameter can take on a logical value. Each logical value can be associated with actual test data, providing a link between testing logical portions of functionality and the test data actually used to do the testing.

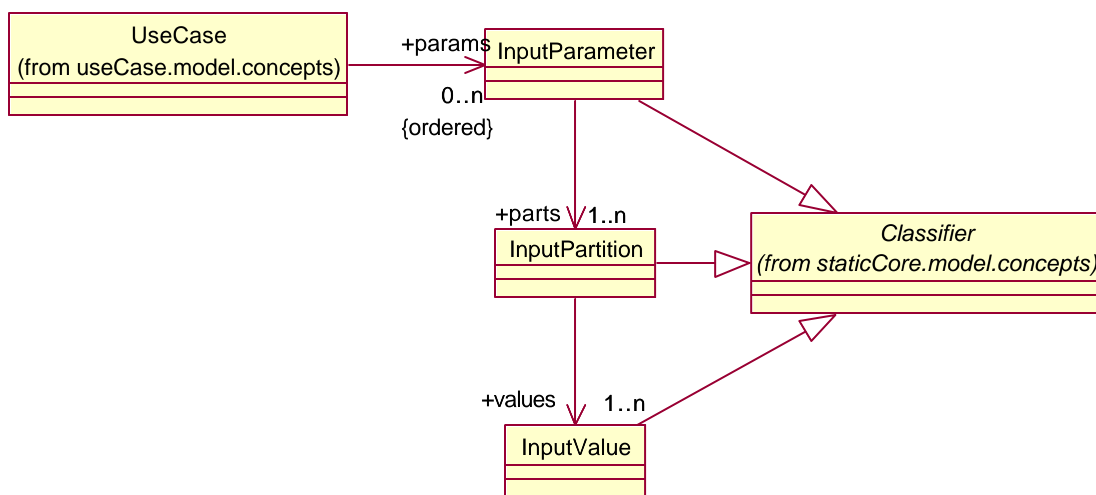

Figure 13. Parameters, Partitions, and Test Data in the testing.model.concepts Package

Given these new classes and relationships, several new constraints must be defined on the model. For reasons of space, I only give a representative sample here taken from use cases, actors, and flows. The various message and data handling classes need to be considered as well, but they are handled in a manner identical to those below.

Well-formedness Constraints

[1] Use case must contain its action set, context, preconditions, postconditions, and input parameters.

```
context testing.model.concepts.UseCase inv:
   elements->
     (exists(g | g.name = "context" and
         g.elements = context) and
     exists(g | g.name = "precondition" and
         g.elements = precondition) and
     exists(g | g.name = "postcondition" and
         g.elements = postcondition) and
     exists(g | g.name = "actions" and
         g.elements = actionSet()) and
     exists(g | g.name = "inputParams" and
         g.elements = paramSet())
   )
```

[2] Actors contain their flows.

```
context testing.model.concepts.Actor inv:
   elements->exists(g | g.name = "flow" and
         g.element = flows))
```

[3] Flows contain their sequence sets.

```
context testing.model.concepts.Flow inv:
   elements->exists(g | g.name = "sequence" and
         g.element = seqSet())
```

[4] The context for a precondition and postconditions is the context for the use case to which it belongs.

```
context testing.model.concepts.UseCase inv:
   self.precondition->forAll(p | self.context = p.context)
   self.postcondtion->forAll(p | self.context = p.context)
```

Methods

[1] The actionSet() method returns a set of all actions within a use case.

```
context testing.model.concepts.UseCase
   actionSet() : Set{Action}
   actions->interate(a s=Set{} | s->union(a))
```

[2] The paramSet() method returns all parameters associated with a use case.

```
context testing.model.concepts.UseCase
   paramSet() : Set(InputParameter)
   params->iterate(p s=Set{} | s->union(p))
```

[3] The seqSet() method returns the set of all use cases contained in a flow.

```
context testing.model.concepts.Flow
   seqSet() : Set(UseCase)
   sequence->iterate(u s=Set{} | s->union(u))
```

4.2 testing.instance.concepts

The semantic domain for testing is enhanced with appropriate execution classes for capturing action instances, value classes for pre- and postconditions, the notion of transactions as instances of flows, and instance classes for the

parameter and partition classes. Execution is covered using the abstract MessageExecution class, which is specialized into three types of executions for messages: InputExecution, OutputExecution, and ComputationExecution. Each of these message types recieves a sequence of Instances as their input and/or output. The message executions are directed toward a particular ActorInstance, which must have an association with the UseCaseInstance. Figures 14 through 17 illustrate the concepts from the testing.instance.concepts package.

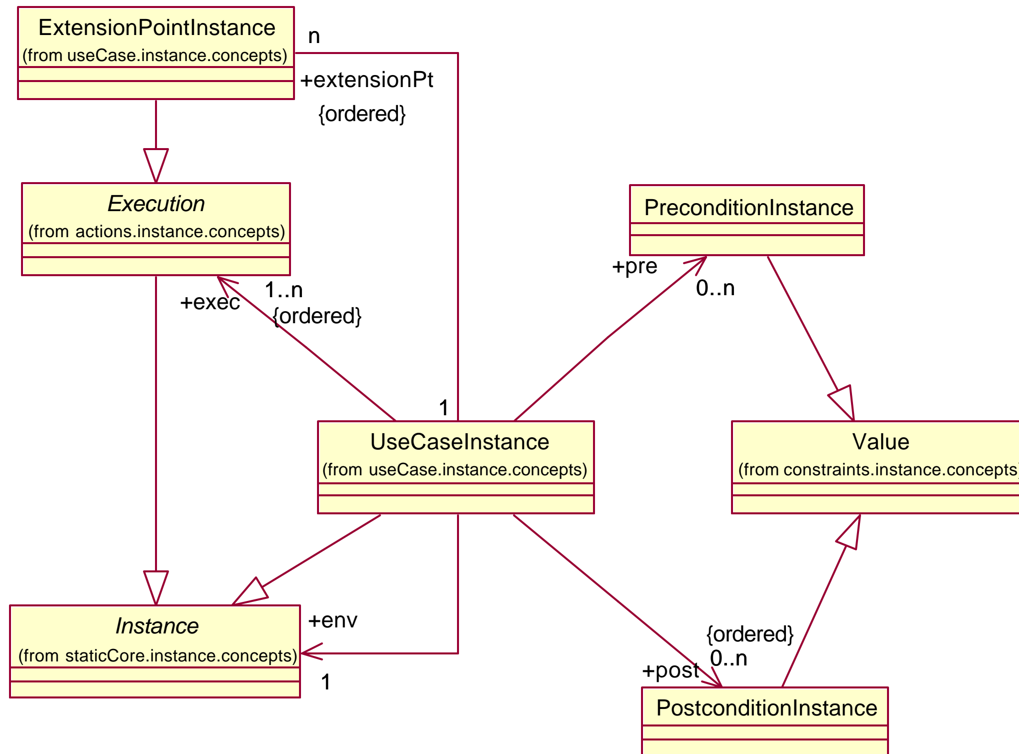

Figure 14. Use Case Instances in the testing.instance.concepts Package

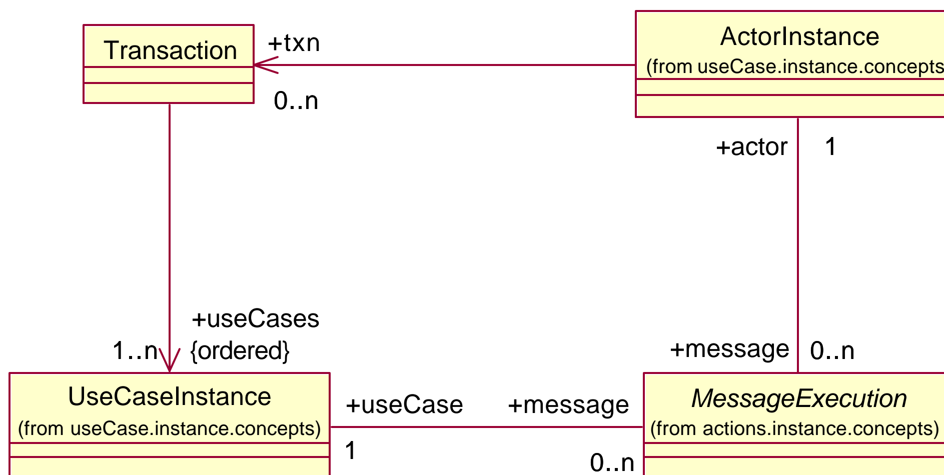

Figure 15. Transactions and Messages in the testing.instance.concepts Package

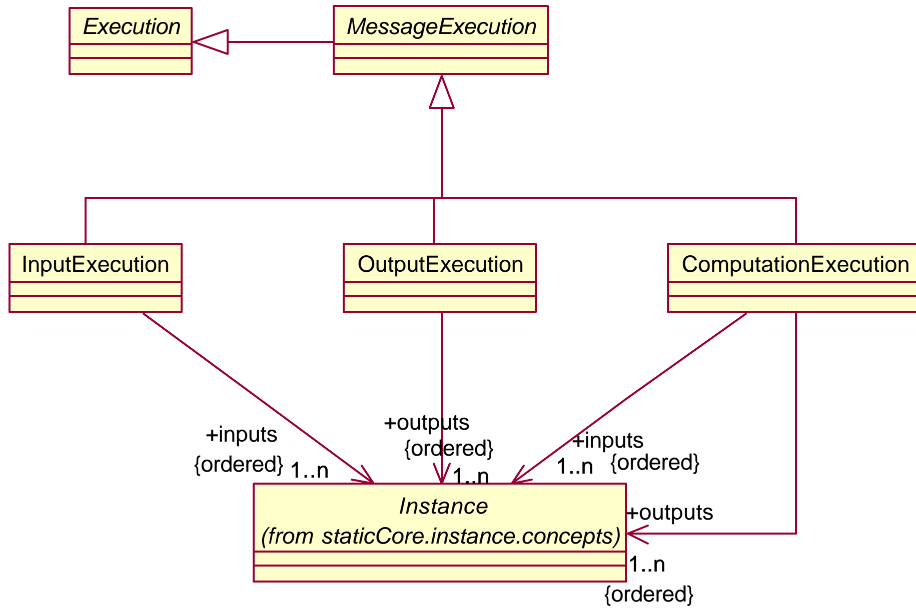

Figure 16. Message instances in the testing.instance.concepts Package

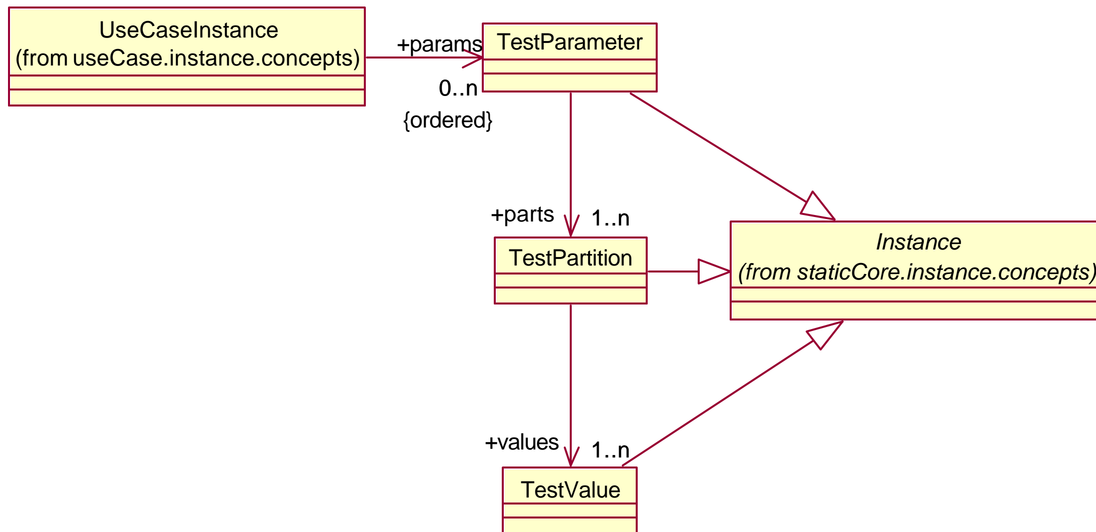

Figure 17. Logical / Physical Representations of Test Data in testing.instance.concepts Package

Well-formedness Constraints

Because many of the constraints are similar or identical to constraints constructed earlier, only two interesting constraints are shown here.

[1] All transactions for an actor must contain only use cases with which the actor is associated.

```
context testing.instance.concepts.ActorInstance inv:
   self.transactions->forall(t |
         t.testSteps->forall(u |
               self.assoc->exists(a | a.useCase = u)
         )
   )
```

[2] Messages can only flow between associated actor and use case instances.

```
context testing.instance.concepts.MessageExecution inv:
   self.actor.assoc->exists(a | a.useCase = self.useCase)
```

4.3 testing.semantics

The package testing.semantics provides semantic mappings. For the sake of space, only a sample of these mappings are shown in Figure 18. The rest are straightforward and follow the same pattern. Two well-formedness rules are constructed for the more interesting semantic properties that must hold.

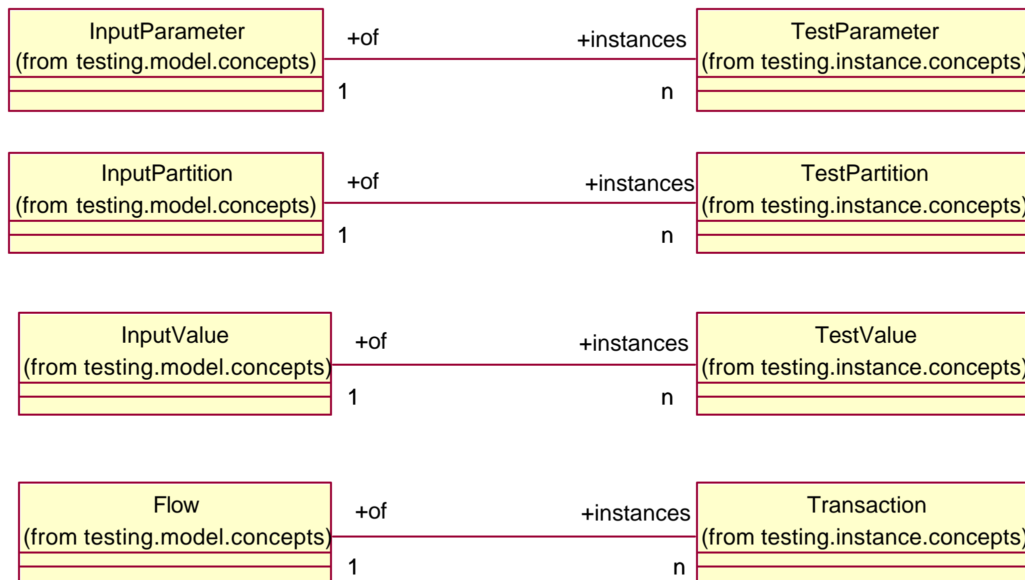

Figure 18. Semantic mappings for testing.semantics Package

Well-formedness Rules

[1] The execution sequence of a use case instance must include Execution elements corresponding to the Action sequence of the use case.

```
context testing.semantics.UseCaseInstance inv:
   Sequence{1,2,...,self.exec->size}->iterate(i o=true |
         (self.exec->at(i).of = self.of.actions->at(i)) and o)
```

[2] The use case instances in a transaction must correspond to the use cases defined in the corresponding flow.

```
context testing.semantics.Transaction inv:
   Sequence{1,2,...,self.useCases->size}->iterate(i o=true |
         (self.useCases->at(i).of = self.of.useCases->at(i)) and o)
```

4 Conclusion and Future Work

UML use cases can serve as the basis for model-based testing, but they require some additional information to be useful. I developed a set of requirements that a meta-model should satisfy in order to facilitate the capture of this additional information. Next, I developed a meta-model for standard use cases based upon UML1.3, but using the techniques defined in the MMF. This standard model was then extended to develop a test-ready meta-model. This meta-model has been used as a foundation for a tool for model-based testing. Early pilots with this tool have produced promising results, indicating that the concepts formalized above are important in test case generation. Several important questions remain concerning model-based testing with use cases. Additionally, there are some broader questions about testing and UML that need to be explored.

As noted earlier, generalization requires a thorough treatment in order to be formalized in a way that is useful from a testing perspective. Similarly, extension and inclusion relationships require additional work to formalize the notion of location references in the base use case. How these different relationships affect one another must also be considered carefully. For instance, generalization indicates that a child use case should inherit the relationships of its parent, including the ≪include≫ and ≪extend≫ relationships. What this means needs to be thoroughly explored and considered in the meta-model.

Utilizing other UML elements in modeling is another important area that requires further work. For example, behind a use case might be a set of sequence diagrams indicating how the use case is realized. This information could be used to enhance test generation algorithms, but it needs to be related to the use case model in a rigorous way. One approach would be to develop a meta-model for formalizing realization, refinement, and composition. These topics are covered in the Catalysis process [DW99], but require more study and formalization in order to gain the maximum advantage by supporting them in an automated environment.

Finally, this work raises a basic question about use cases in general. If it is not essential to consider use cases as having state for test case generation, is state an important or desirable concept to have associated with use cases? The basic impetus for arguing this point is that use cases are (or should be) at their most detailed and robust by test case generation time. If this is true, and state is not required, will it ever be useful for practitioners to think about use cases as having state? I suspect that state should be left to other classifiers such as classes, subsystems, and systems, but I would like to see a broader conversation about use cases as "stateless" classifiers.

Bibliography

[Al01] Álvarez, J. et. al.: An Actions Semantics for MML. Available at http://www.puml.org/mml

[ASC00] Action Semantics Consortium: Response to OMG RFP ad/98-11-01.: Action Semantics for the UML. Revised September 5, 2000 (2000). Available at http://www.umlactionsemantics.org

[BRJ99] Booch, G.; Rumbaugh, J.; Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley, 1999.

[CEK01] Clark, T.; Evans, A.; Kent, S.: The Meta-modeling Language Calculus: Foundation Semantics for the UML. In (Hussmann, H.): Proc. 4th Intl Conf. on Fundamental Approaches to Software Engineering, Genova, Italy, 2001. Springer-Verlag, Berlin, 2001. pp. 17-31.

[Cl00] Clark, T. et. al.: A Feasibility Study in Rearchitecting UML as a Family of Languages using a Precise OO Meta-Modeling Approach. (2000) Available at http://www.puml.org/mml

[Cl01] Clark, T.; et. al.: The MMF Approach to Engineering Object-Oriented Design Languages. Available at http://www.puml.org/mml

[DW99] D'Souza, D.; Wills, A.: Objects, Components, and Frameworks with UML: The Catalysis Approach. Addison-Wesley, 1999.

[Ja92] Jacobson, I.; et. Al.: Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, 1992.

[OP98] Övergaard, G.; Palmkvist, K.: A Formal Approach to Use Cases and Their Relationships. In (Bézivin, J.; Muller, P.): Proc. 1st Int. Workshop on the Unified Modeling Language, Mulhouse, France, 1998. Springer-Verlag, Berlin, 1998. pp. 406-418.

[P00] Paradkar, A.:SALT-an integrated environment to automate generation of function tests for APIs. In Proc. 11th Int. Symp. on Software Reliability Engineering, San Jose, California, 2000. IEEE Press, 2000. pp. 304-316.

[RJB00] Rumbaugh, J.; Jacobson, I.; Booch, G.: The Unified Modeling Language Reference Manual. Addison-Wesley, 2000.

[S01] Stevens, P.: On Use Cases and Their Relationships in the Unified Modeling Language. In (Hussmann, H.): Proc. 4th Intl Conf. on Fundamental Approaches to Software Engineering, Genova, Italy, 2001. Springer-Verlag, Berlin, 2001. pp. 140-155.

[WK99] Warmer, J.; Kleppe, A.: The Object Constraint Language: Precise Modeling with the UML. Addison-Wesley, 1999.

[WP99] Williams, C.; Paradkar, A.: Efficient Regression Testing of Multi-Panel Systems. In Proc. 10th Int. Symp. on Software Reliability Engineering, Boca Raton, Florida, 1999. IEEE Press, 1999. pp. 158-165.