Toward 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 classifiers (classes, subsystems, or systems.) Two principal purposes that use cases serve are specifying the functionality the classifier will provide and providing a basis for developing test cases for the classifier. This paper discusses issues that arise when using use cases as the basis for model-based testing. Based on this discussion, a test-ready meta-model for use cases is developed. Next, I describe a tool constructed using the concepts from the meta-model, and provides data from the initial pilots of this tool. I close by discussing future research.

1 Introduction

The Unified Modeling Language (UML) utilizes use cases as a means to specify 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. 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 for external use. 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.

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

The remainder of this paper explores the issues that must be addressed to support model-based testing with use cases. The foundations of a meta-model is developed to address these issues. The paper closes by discussing a tool built on these ideas and topics for further research.

2 Basic Tenets for Testing with Use Cases

Before proceeding to explore a test-ready use case meta-model, it is important to identify what is required for use cases to be used in conjunction with model-based testing techniques.

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. The use case is simply a set of action sequences which involve either the classifier or actors associated with the use case. All relevant state information is associated with the classifier. The only 'state' that belongs to the use case is the current action being performed, which can be managed with a simple counter. This viewpoint differs significantly from earlier work that formalizes use cases using state based techniques [S01]. The justification for my "stateless" view of use cases is described below. The two viewpoints may be reconcilable using the following approach: 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].

From a testing viewpoint, the state based formalization of use cases presented in [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 an 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 a classifier. Given this definition, techniques based on data flow analysis can be used to generate interesting test cases based on interference.

If use cases do not require a notion of state beyond an action counter, neither do they require the notion of attribute or operation as discussed in [RJB00]. These concepts are best when applied to the classifiers which the use case describes, keeping the use case focus on describing functionality, not structure.

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. All of the other standard relationships (generalization, extension, inclusion, realization, and associations with actors) should be supported as defined in the UML 1.3 standard.

2.2 Use Cases consist of Actions

Since our representation of use cases is that of a sequence of actions without persistent attributes or state, developing the appropriate action representation is essential. Action semantics have been considered in [ASC00] 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 interactions between an actor and 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.

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 one use case directly follows another. For example, suppose we have a set of use cases $U=\{u_1,u_2,...,u_4\}$ and an actors A. 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 1. Note that these edges are not associations, but temporal flows through the system use cases. The meta-model should support the definition of a set of use case flows that can be associated with an actor.

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 pass from an actor to the classifier, as well as the outputs that move 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 with use cases will support the notions of parameters, logical partitions, and actual data that belongs to a given logical partition.

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.

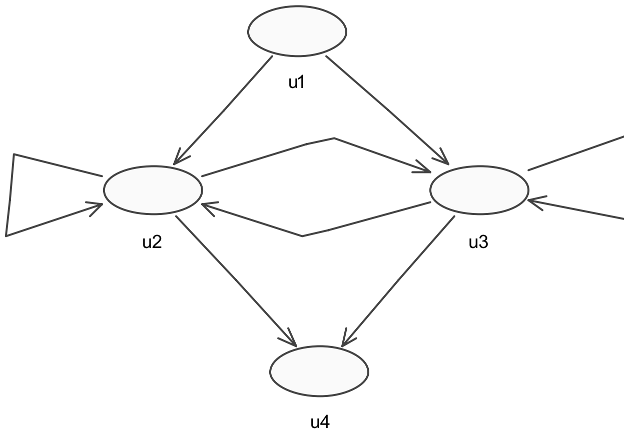

Figure 1. Use Case Flow Graph

3 The Testing 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 (a machine processable representation). The semantic mapping maps the abstract syntax to the semantic domain, which describes how model concepts are realized in model instances. Modeling using MML is supported by the meta-modeling tool (MMT.)

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 testing package described below use elements from the mml package described in [Cl00], the useCase package described in [W01], and the actions package described in [Al01]. It extends these packages, as shown in Figure 2. The 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. The Object Constraint Language (OCL) [WK99] is use to specify additional constraints to guarantee that the meta-models are well formed.

3.1 A Brief view of the useCase Package

The useCase package uses MMF techniques to describe the basic modeling structures required to build use case models. A more detailed description [W01] is available from

the author of this paper by sending an e-mail request. 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. Generalization as well as the ≪extend≫ and ≪include≫ relationships are supported. Some of these concepts are key for test case generation as well. For example, ≪extend≫ and ≪include≫ relationships must be tested to verify that the system supports them correctly.

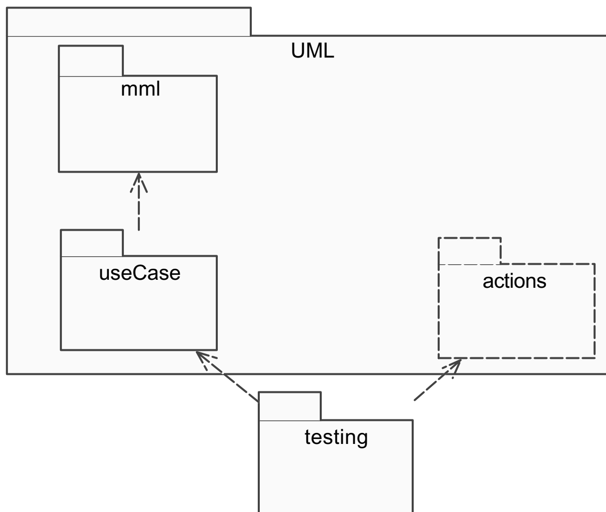

Figure 2. Generalization relationships between packages.

3.2 The testing Package

This package contains the meta-model that addresses the concerns described in section 2. This package contains three subpackages: testing.model.concepts (abstract syntax), testing.instance.concepts (semantic domain) and testing.semantics (the mapping from abstract syntax to the semantic domain.)

3.2.1 testing.model.concepts

The following formalizations are used to meet the requirements from section 2:
(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 3 shows the use case portion of the meta-model based on formalizations 1-3 described above. 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.

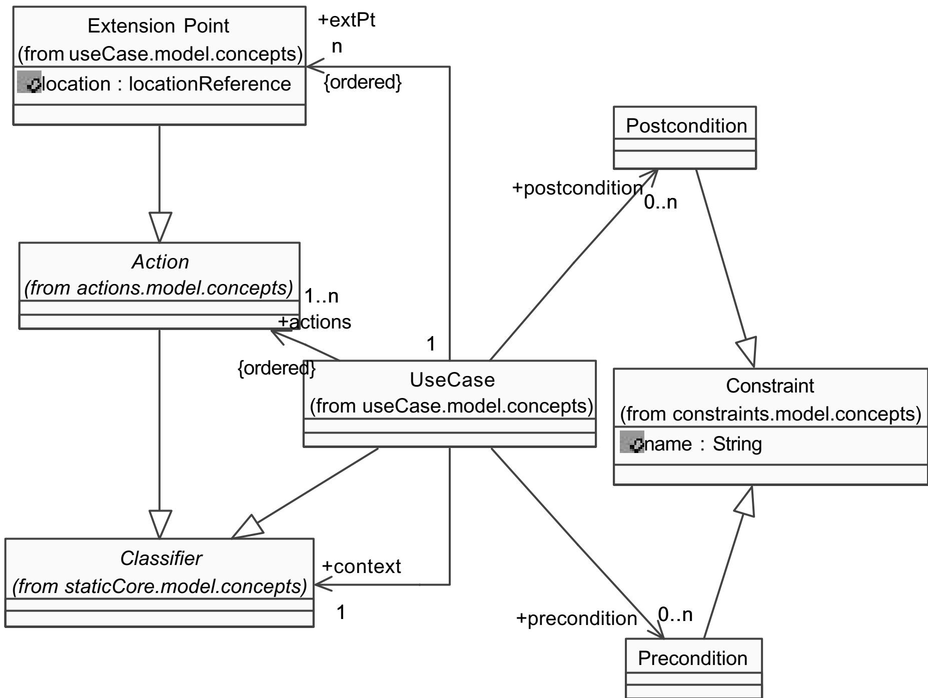

Figure 3. The Use Case meta-model in the testing.model.concepts Package.

The actor related portions of the meta-model are also described in the testing package (Figure 4.) Actors are associated with flows through use cases, which defined sequences of use cases that are interesting from a testing point of view. Actors are defined to communicate with classifiers via message related actions (Figure 5.)

Figure 6 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 represented by InputPartition. Each logical value can be associated with actual test data (as InputValue), providing a link between testing logical portions of functionality and the test data actually used to do the testing.

Given these classes and relationships, several constraints must be defined on the model. For reasons of space, I only give a representative sample. These include constraints that specify that use case must correctly contain references to other elements, as well as constraints that guarantee that the use case specifies actions only on the classifier against which it is written. The constraints not shown will be similar to these or others described in [Cl00].

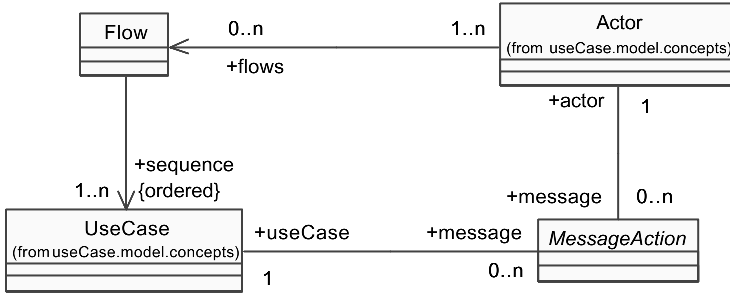

Figure 4. Flows and Messages between Actors and Use Cases.

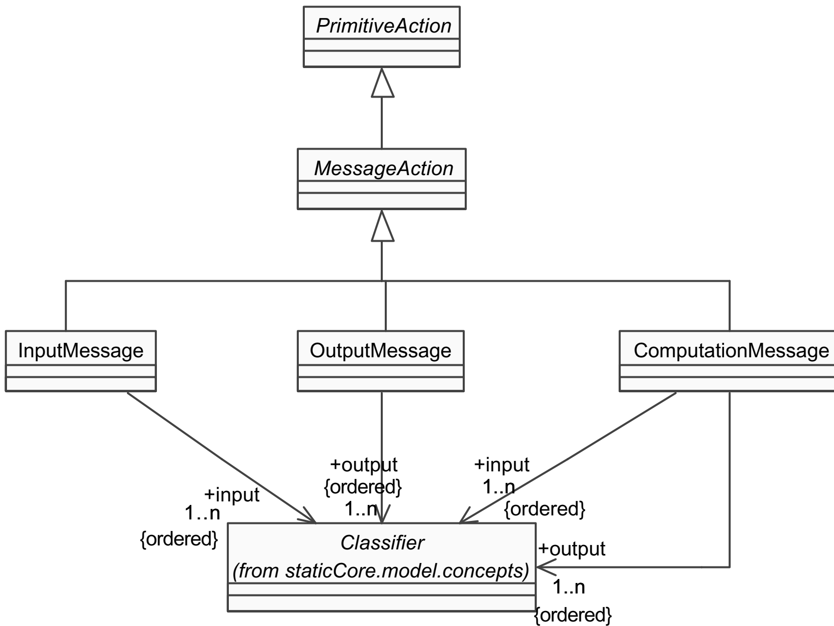

Figure 5. Message Actions in the testing.model.concepts Package

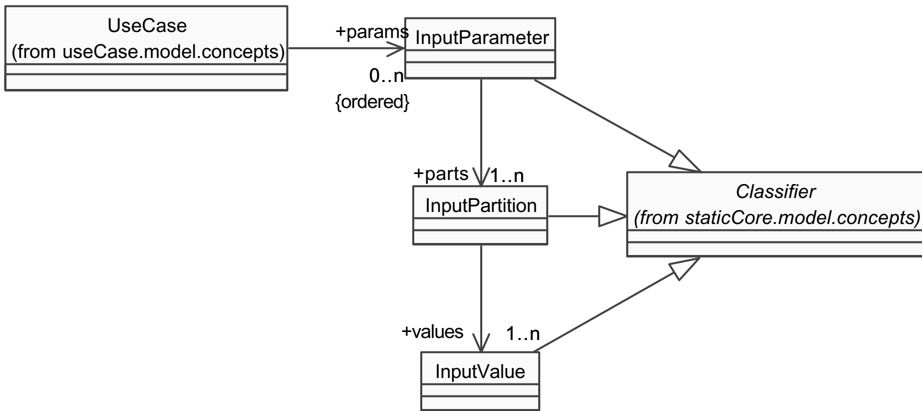

Figure 6. Parameters, Partitions, and Test Data in the testing.model.concepts Package

Well-formedness Constraints

(1) Use case must contain its action set and context. (Preconditions, postconditions, and input parameters will be handled in a similar fashion.)

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

(2) The context for a precondition and postcondition 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))
```

3.2.2 testing.instance.concepts

The semantic domain for testing is enhanced with appropriate execution classes for capturing action instances, instance 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 7 through 10 illustrate the concepts from the testing.instance.concepts package.

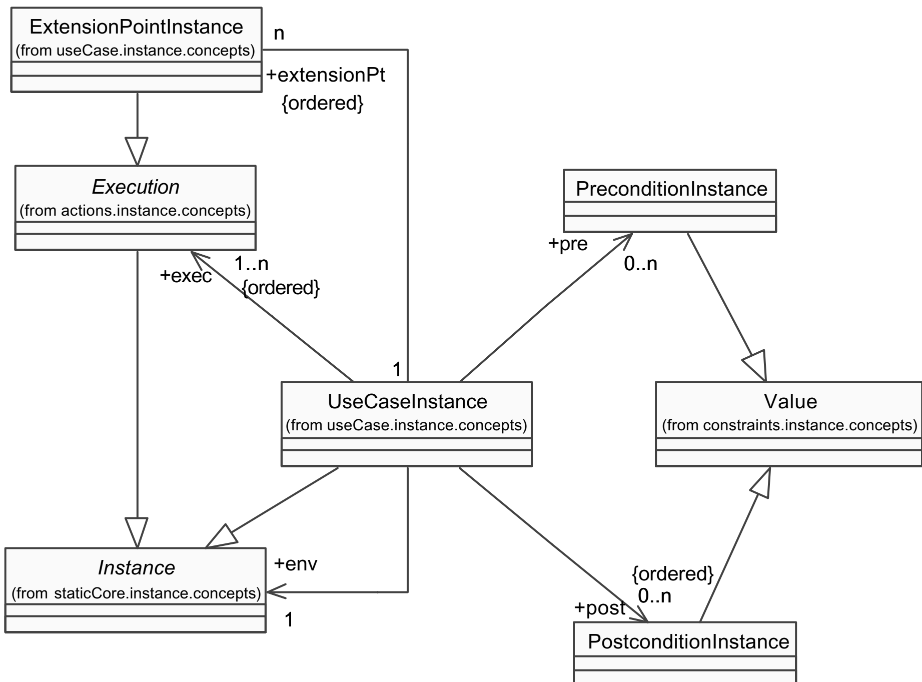

Figure 7. Use Case Instances in the testing.instance.concepts Package

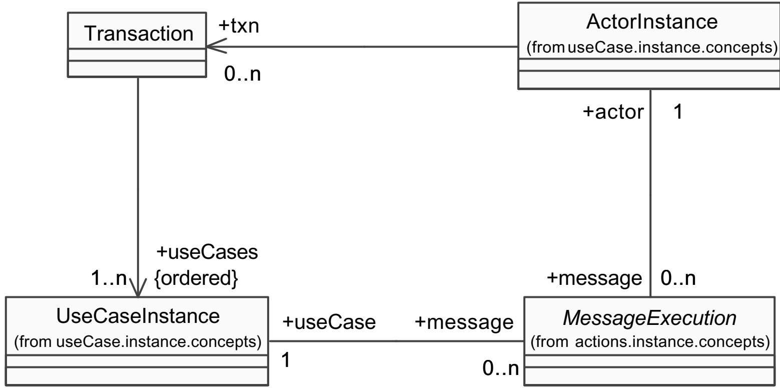

Figure 8. Transactions and Messages in the testing.instance.concepts Package

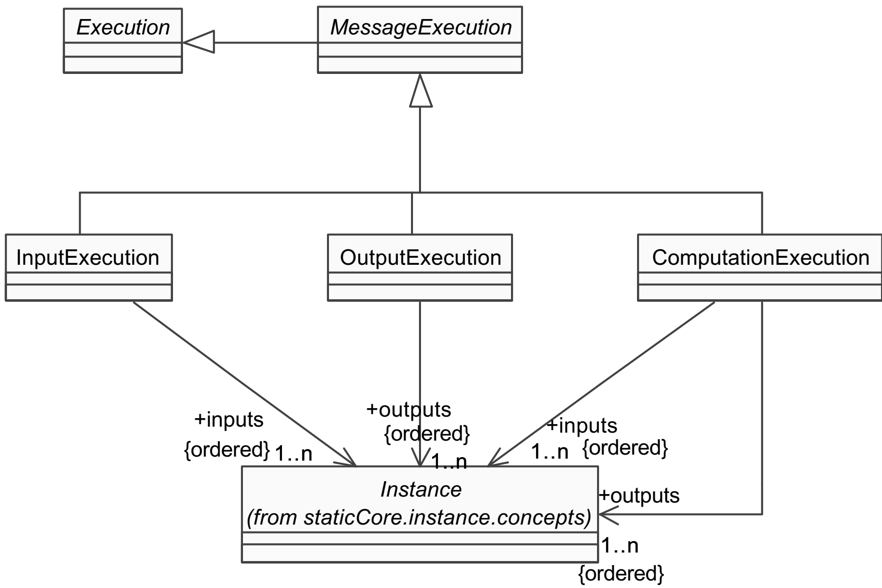

Figure 9. Message instances in the testing.instance.concepts Package

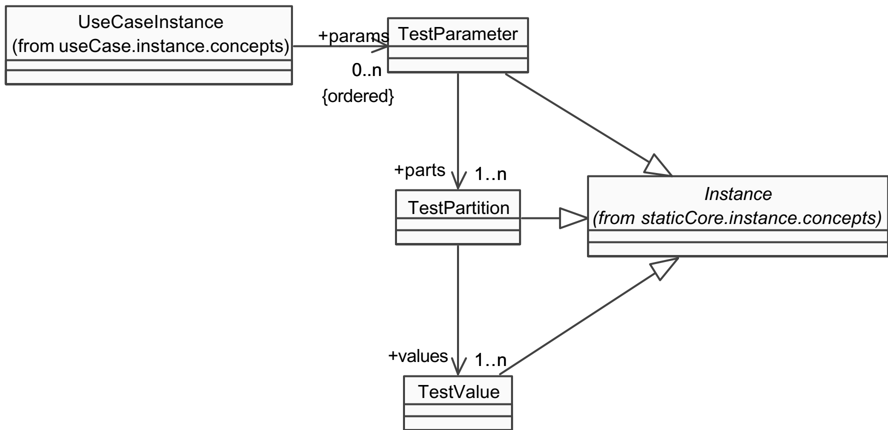

Figure 10. Test Data in testing.instance.concepts Package

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. (Association is defined in [W01].)

```
context testing.instance.concepts.ActorInstance inv:
   self.txn->forall(t |
         t.useCases->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)
```

3.2.3 testing.semantics

The package testing.semantics provides semantic mappings. These mappings follow the same pattern described in the useCase package (many to one mappings between instances and model elements.) A sample set of four mappings is shown in Figure 11. Well-formedness rules for two interesting semantic properties that must hold are also provided.

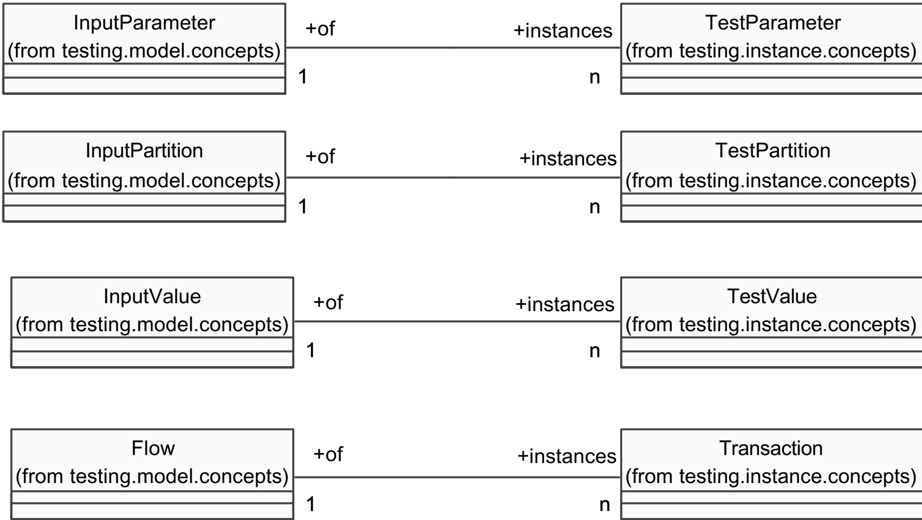

Figure 11. Mappings in the testing.semantics Package

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}->forAll(i |
        self.exec->at(i).of = self.of.actions->at(i))
```

(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}->forAll(i |
       self.useCases->at(i).of=self.of.useCases->at(i))
```

4 UCBT - A Tool for Use Case Based Testing

UCBT is a tool for generating test cases from annotated use case models. It is based on the concepts described in the meta-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]; however, the tool has been enhanced with additional capability not covered in the paper. In this section, I present a brief overview of the tool, some preliminary results of its use, and the ongoing work on the tool.

4.1 An Overview of UCBT

UCBT is a tool that runs inside of one of the popular UML modeling tools. It allows the tester to annotate the use case models derived during the system requirements phase. The annotated models can then be used for test generation. The annotation information is separated into 4 submodels: the system model, the use case model, the data model, and the user model. These models and their relationships are shown in Figure 12.

The system model is a simplified representation of the classifier whose behavior the use cases describe. Currently, this is a collection of parameters, each of which has an enumerated set of possible values. This is a much more simple representation than the full range that UML would provide, and part of our current work is enriching the representation of the system model. Given the current system representation, the tool can use the system model to keep track of the values of parameters of interest. These parameters can also be used as the foundation for pre- and postconditions on the use cases.

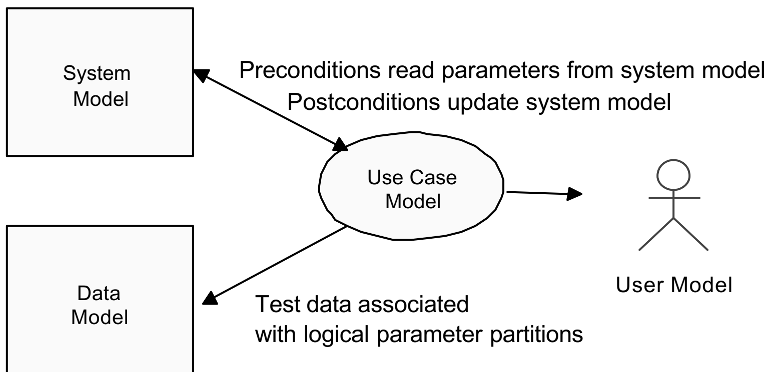

Figure 12. Relationships Between Sub-models in UCBT

The use case model describes the capabilities that the system provides. Associated with each use case is a set of input parameters. Each parameter has a set of logical partition values associated with it. The use case model also provides a mechanism for specifying the results of the use case that are returned to the actor, as well as pre- and postconditions. The use case model supports the definition of constraints on ≪extend≫ relationships. The parameters for these constraints may come from either the system model, or they may be based on the input parameters to the use case. The use case model also provides a way to specify test combinations of interest on the input parameters.

The data model is quite simple. It provides a mechanism for mapping the parameters and logical partitions for a given use case to actual test data values. Currently the model only support simple types. Support for complex types is an area we are investigating.

The user model captures flow sequences that are associated with each actor in the use case model. These are captured as simple use case pairs (A,B), indicating that use case A may be followed by use case B. The user model also contains a workload percentage for each actor. This is used to develop a recommended test configuration given the test cases that are generated by the tool.

Given an annotated model, UCBT generates test cases by performing three specific steps:

(1) Optimization of input combinations for each use case.
(2) Optimization of the flow sequences across use cases. This results in an optimal flow graph.
(3) Generation of the test cases by traversing the flow graph.

The tool generates two types of test cases: human readable and executable. The human readable test suites are suitable for inclusion in a test plan document. The executable test cases can be run against any system that provides a programmatic API. Because of the two-phase optimization, UCBT produces very focused suites of test cases. These suites guarantee that all of the combinations of interest and all of the flows are covered. This means that the tester is assured that the items that they specified will be tested, and the testing will occur in a very efficient manner.

4.2 Preliminary Results for UCBT

UCBT has been evaluated in three small pilots (see Table 1.) The purpose of these pilots was to study how appropriate our representations were and how useful the tool was for application to commercial software systems. The defect detecting quality of the tool was not extensively examined in these pilots; we are currently exploring this further.

4.3 Current Enhancements

Currently, we are focusing on three basic areas for enhancement.

(1) Enhancing the system model to support a richer set of elements that is consistent with the UML notion of classifier.
(2) Implementing a more robust action language for describing use case behavior. The current tool has a very simple action model that makes representing selection and/or iteration structures challenging.
(3) Implementing a richer notion of flow. The pair-wise description of use case flow was not flexible enough to do some of the things that our pilot testers would like to do. We are currently exploring making flow definition more flexible by using flow scenarios rather than pair-wise definition.

Pilot	Pilot Properties	Outcome
1	• Test cases developed by research. • Handed off to system owner for execution. • Compared with the current, manually developed suite by running both.	• Fivefold reduction in test suite size (30 vs. >150 test cases.) • 68% decrease in time to setup/run/verify test suite • Major defect detected by new suite that escaped manual suite.
2	• Modeling and generation done in testing organization. • Tester built UML use case model and generated test cases. • Comparison against a manual suite developed concurrently by a different tester.	• Coverage of use case functionality improved 30% in the model-based suite. This was achieved with 10% fewer use case invocations. • Testing team estimates achieving comparable coverage to model-based suite would require twice the effort.
3	• Modeling and generation done in testing organization. • UML use case model built during system design, used by testing organization. • Testers analyzed the suite for usability in the project.	• >90% coverage of the functionality with a suite of 60 test cases. • The suite tested 40 use cases. In total, there were 145 input parameters, 335 partitions, and 117 flow edges in the model.

Table 1. Pilot Results for UCBT

5 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 extend the mml, useCase, and actions packages to develop a basis for a test-ready use case 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 in [S01], generalization requires a thorough treatment in order to be formalized in a way that is useful. 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 ≪iinclude≫ 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 formalization for use 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? Use cases are (or should be) in their most detailed form by test case generation time. If this is true, and state is not required at this most detailed level, 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 a broader conversation about these issues needs to take place.

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.

[W01] Williams, C.: A Meta-Model for Use Cases. IBM Research Report, publication pending.

[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.