The Three R's of Use Case Formalisms: Realization, Refinement, and Reification

Russell R. Hurlbut
Expertech, Ltd.
P.O. Box 4151 Wheaton, IL 60189
Document: XPT-TR-97-06
rhurlbut@charlie.iit.edu

ABSTRACT. *Version 1.1 of the Unified Modeling Language (UML) includes a behavioral elements subpackage devoted to use cases. This paper identifies three primary mechanisms for expressing use cases in design models: realization, refinement, and reification. A use case is typically realized in terms of collaborating objects belonging to classes. A use case may be refined through a collaboration of use cases that provide the specification for each of the elements of the entity specified by the superordinate use case. A use case is reified through a use case instance that constitutes the performance of the sequence of actions specified in a scenario of the use case. This paper describes the full set of semantics for use cases. Diagrams of UML model elements tailored for expressing these three use case mechanisms are developed. Using the graphical models and semantics presented, the relationships among these three models are explored to derive guidelines for appropriate representations of use cases when developing a domain model.*

KEY WORDS: *Use Case Formalisms, Object Oriented Analysis and Design, Unified Modeling Language, Domain Modeling*

1. Introduction

The Unified Modeling Language (UML) specification version 1.1 [UML97] suggests several different formats for representing a use case. A use case can be represented as plain text, operations, activity diagrams, state-machines. Other behavior description techniques, such as pre- and post conditions, are also permitted. A more complete coverage of use case representation can be found in [Hurl97c]. Regardless of the representation technique utilized for use cases, a use case model is first and foremost a requirements and analysis construct. Consideration of a use case from a design perspective involves one or more of the following mechanisms: realization, refinement, and reification. Each of these three mechanisms relies on the relationship of the a use case to other modeling elements in the UML meta-model. The primary specification technique for realizing a use case is through a collaboration model element in terms of collaborating objects belonging to classes. A use case may also be realized as a state machine or activity model. Since entities at all levels of realization may also have use cases attached to them, the UML specification permits the refinement of use cases through a collaboration of subordinate use cases that provide the specification for each of the elements of the entity specified by the superordinate use case.

The UML semantics also specifically define a use case instance as a model element. This is the only model element that is given such first class status in the meta-model[1]. A use case is reified

[1] It can be argued that a message instance shares this distinction as an instance of a request model element. However, there is ambiguity with respect to its name, since UML contain a message model element as well. Furthermore, since Request is an abstract model element, the actual reification should be an operation instance or a signal instance.

through a use case instance that constitutes the performance of the sequence of actions specified in a scenario of the use case. UML defines a scenario as "a specific sequence of actions that illustrates behaviors." This definition is consistent with that from its roots in the Object Modeling Technique (OMT) definition of a scenario [RBP+91]: "a scenario is a sequence of events that occurs during one particular execution of a system…a scenario can be the historical record of executing a system or a though experiment of executing a proposed system." Since use cases can be recursively refined into subordinate use cases, it is also fully consistent with this definition to maintain the assertion from OMT that the scope of a scenario can vary. Thus, a use case instance, or scenario, may capture action at varying levels of use case refinement. Ultimately, a use case instance may contribute to fleshing out requirements, or it may reify the execution sequence of the system for testing or auditing purposes.

2. Use Case Meta-Model Constructs and Semantics

The presentation of the use case behavioral elements package in the UML Semantics appears deceptively simple, at least from the diagram of the model elements described in the abstract syntax. Including the semantics of UML modeling elements from which a use case inherits reveals both complexity and ambiguity. Well-formedness rules for a use case do not exclude consideration of the additional semantics attached to a use case through the UML generalization hierarchy. Figure 2.1 reveals that a use case is progressively specialized from the following UML elements: *Classifier*, *GeneralizableElement*, *Namespace*, *ModelElement*, and *Element*. This model diagram identifies each UML modeling element that has direct semantic impact on a use case.

As an *Element*, a use case may be extended through the *Stereotype* construct along with its associated *Constraints and TaggedValues*. Constraints and tagged values may also be attached directly to a use case through inheritance from *ModelElement*. The model element construct provides for dependencies to be associated with a use case. UML defines four first class dependency metaclasses: *Refinements*, *Usage*, *Trace*, and *Binding*. A refinement permits different levels of abstraction, i.e. use case refinements. This allows a refinement relationship between superordinate and subordinate use cases to exist. Usage allows one use case to reference another use case, particularly those in different packages. Trace permits linking models that may represent use cases from different points of view. Binding permits a use case to be parameterized though a template. Graphical and textual representations of a use case are also defined through *ModelElement*. Inheritance from *NameSpace* places the constraint of unique names for the attached model elements of a use case, i.e. attributes and operations. Through *GeneralizableElement*, use cases are allowed to participate in generalization/specialization hierarchies, which are constrained to be either a «extends» or «uses» stereotyped *Generalization*.

Classifier provides most of the characteristics of a use case that are described in the UML Semantics documentation, namely *Attributes*, *Operations*, and *Interface*. A use case is permitted structural and behavioral features as a specialization of a classifier. Two other behavioral features not explicitly ruled out by well-formedness rules are *Receptions* and *Methods*. *Classifier* permits relationship to other classifiers for purposes or realization, namely *Classes* and *Subsystems*. Realized subsystems and classes are associated through *Collaborations*. A *Collaboration* is also used to associate subordinate use cases attached to these subsystems and classes in representing a use case refinement. When use cases collaborate, one plays the *ClassifierRole* of Actor. The UML *UseCase* dynamic semantics require that it use *Signals* to communicate with actors.

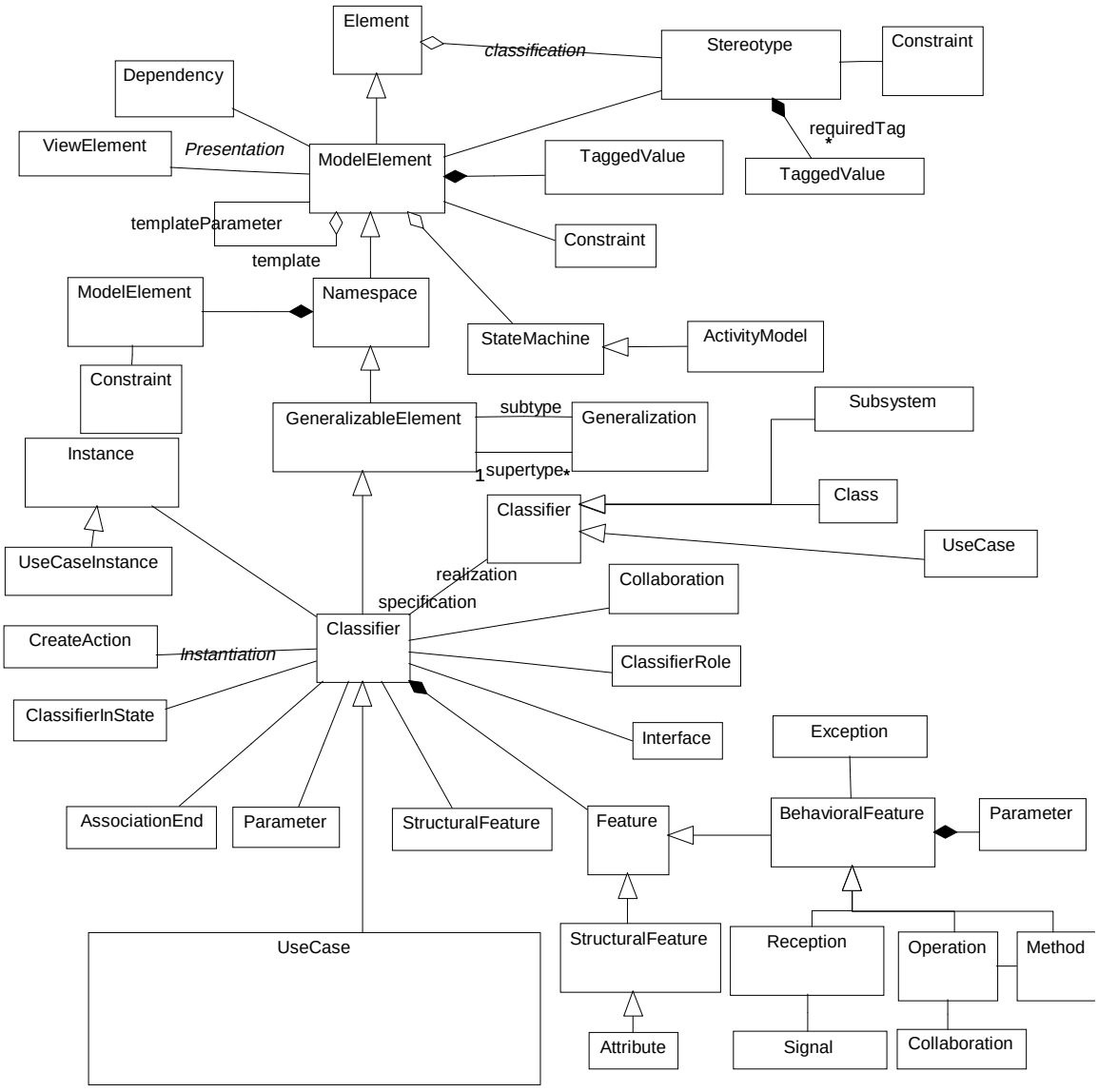

Figure 2.1 : Use Case Inheritance Hierarchy and Inherited Associations

3. Use Case Realization

Use cases may be realized through three types of models: collaboration of subsystems and classes, state machines, or activity models. These three representations may be combined in various manners to provide a desired level of detail and precision. For example, an action that results in a change in the state of a state machine model may be realized by sending a message to an object. That message may be refined through collaboration. Conversely, a collaboration may send a message to an object that is represented by a state machine.

Each of the three types of models is presented at a level of abstraction appropriate to highlight the relationship with a use case. Consequently, some of the details and variations supported by the UML semantics are suppressed. The full semantics for collaborations, state machines, and activity models is available in the UML Semantics document.

Collaboration Specification Models

The use case collaboration model in Figure 3.1 illustrates how subsystems and classes may cooperate to realize the use case specification. Each message that is sent by an actor is ultimately implemented as a method associated with an operation of one of the classes contained within the entity that the use case specifies. If that class is nested within a subsystem, then the operation of the subsystem will be realized through a collaboration at a deeper level of refinement.

A collaboration is a prototype from a structural point of view. The roles involved in the collaboration must be bound to actual objects that will play the roles. The collaboration may be the basis for a design pattern that in turn may be part of an object-oriented framework. In such a case, the binding of objects results from a selection of a framework along with parameters that determine the classes and subsystems that constitute the variant part of the pattern.

The interaction of elements within the entity is modeled through a set of messages. Complex dynamic semantics involving multiple threads of control are possible. Forks and joins to threads of control that represent implementation decision and strategies may be appropriately modeled through collaborations. However, if such behavior constitutes a requirement of the functionality of the entity being modeled, then another realization approach should be considered. This will avoid mingling two point of views, i.e. analysis and design, within the same model. In other words, we should separate our presentation of what is known about the problem from our description of how we have constructed the mechanisms for solving the problem.

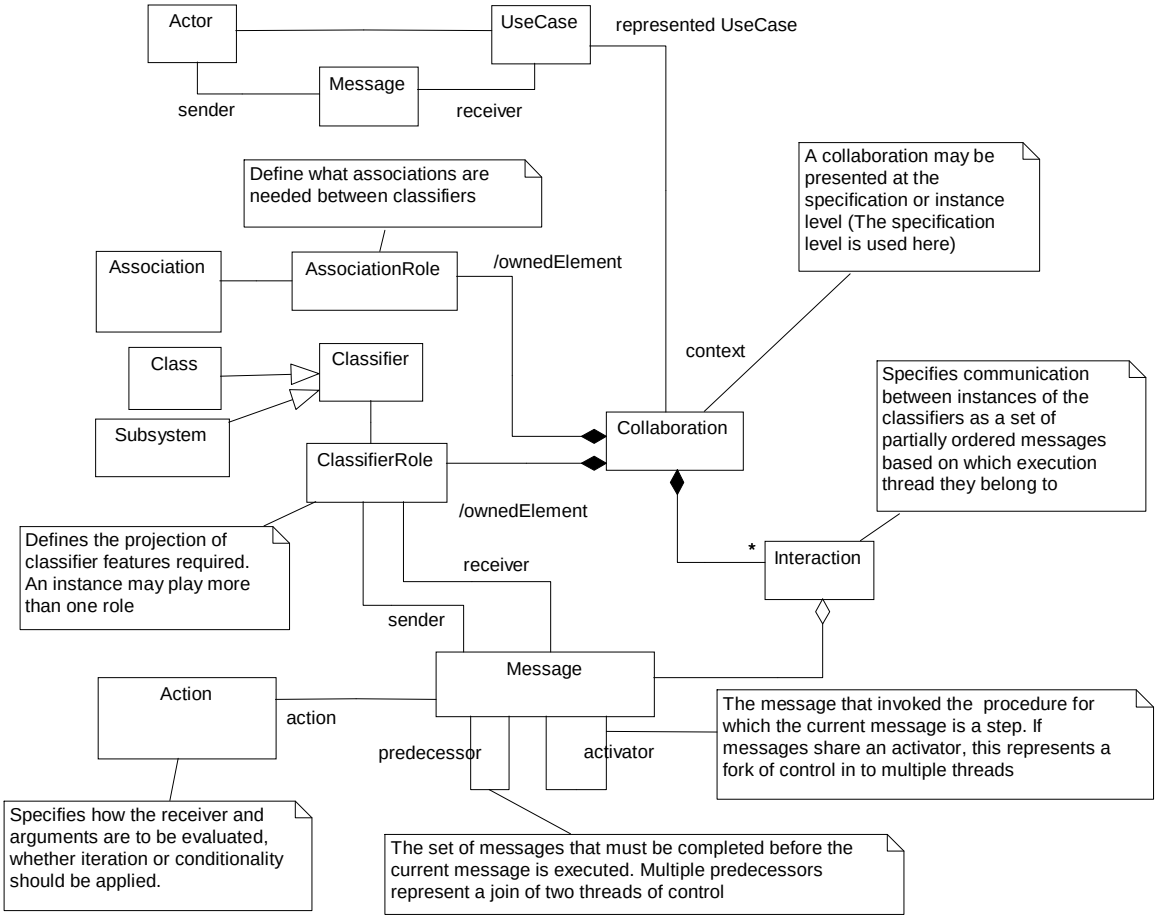

Figure 3.1 : Use Case Realization - Collaboration

State Machines

A state machine may be used as either a protocol specification or as a specification for the complete behavior. The former use provides a description for a use case, the latter use provides the realization for a use case. This section is only concerned with the latter case. The model representing a state machine for a use case is presented in Figure 3.2.

Choosing a state machine to provide the realization for a use case is appropriate for implementations have a class that performs controlling and coordinating functions for the entity being modeled. As suggested in the introduction to this section, collaborations may be used to realize messages that result from actions modeled by the state machine. It also provides a natural separation of analysis and design when also using a state machine for describing a use case, thus providing two views of the state machine. This includes the notion that state-event matrices can be considered a stereotyped state machine use for analysis purposes.

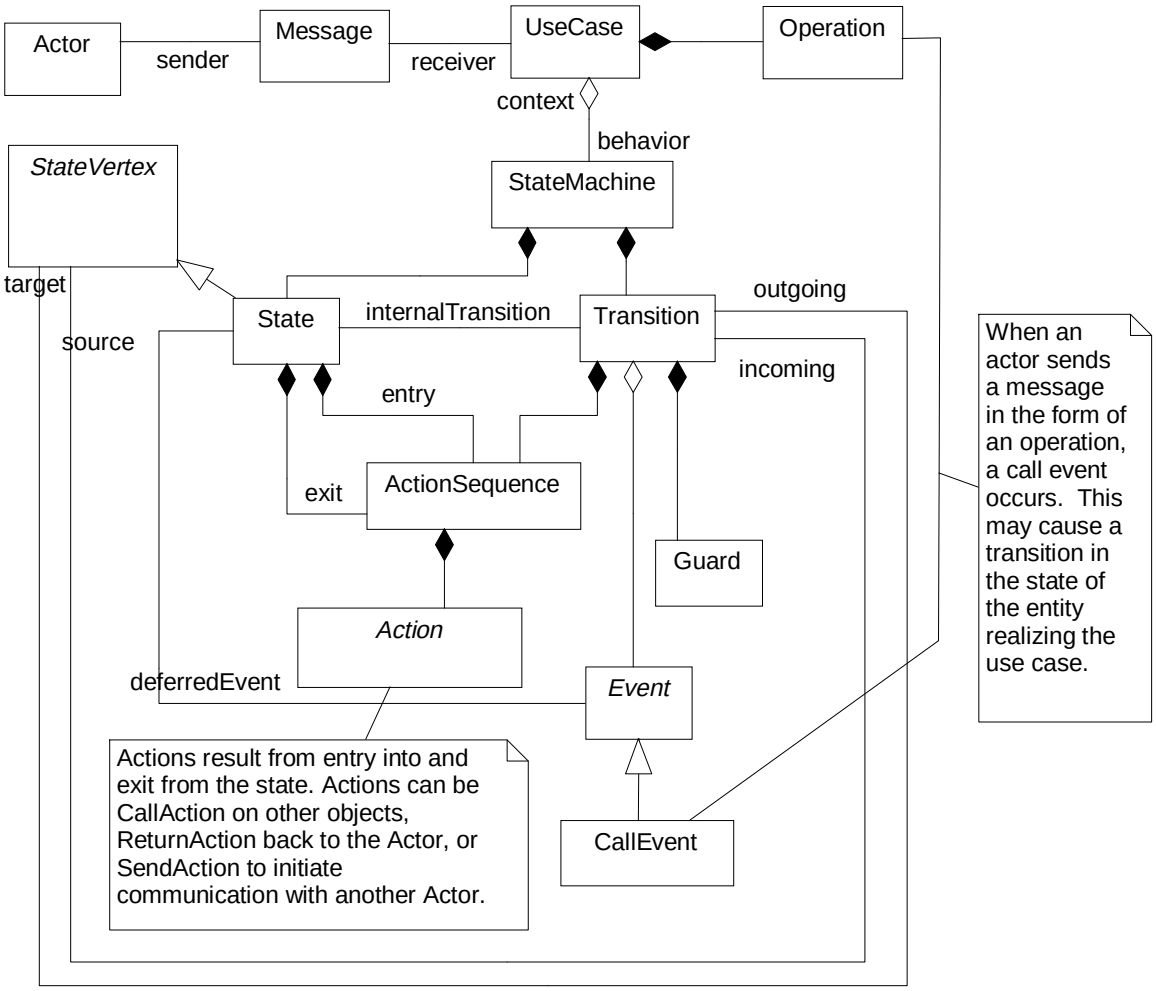

Figure 3.2 : Use Case Realization – State Machine

One of the key distinctions between the use of a collaboration and a state machine involves the treatment of requests. Collaborations use operations and state machines use signals. A signal is used to trigger a reaction in the entity being modeled by a state machine in an asynchronous way and without a reply. The ability to receive and react to a signal is declared through a *Reception*. The reception of a signal invokes a *SignalEvent* that typically causes a transition and subsequent

effects. *Operations* and *Receptions* merely declare that a classifier accepts a given *Request*. Their implementation is described by methods and state machine transitions, respectively. Table 3.1 summarized the distinctions between to two subtypes of *Request.*

Request	Action	Declared by	Described by
Operation	Call action	Operation	Method
Signal	Send action	Reception	State machine transition

Table 3.1 : Request Types

A call action invokes an operation that is realized through a method. A send action may cause a transition has subsequent effects, such as actions. These subsequent actions can be of type *call action.* Thus, we can get to the same place in a round about manner that conforms to the semantics of a state machine.

Activity Models

Activity models represent a special constrained state machine, particularly well suited for use cases and business process modeling (see Figure 3.3). An activity model is characterized by procedural flow of control. State changes represent the completion of an action. The actions contained in the activity model correspond nicely to use cases that are represented as a collection of scenarios described in a structured format as a series of steps in narrative form.

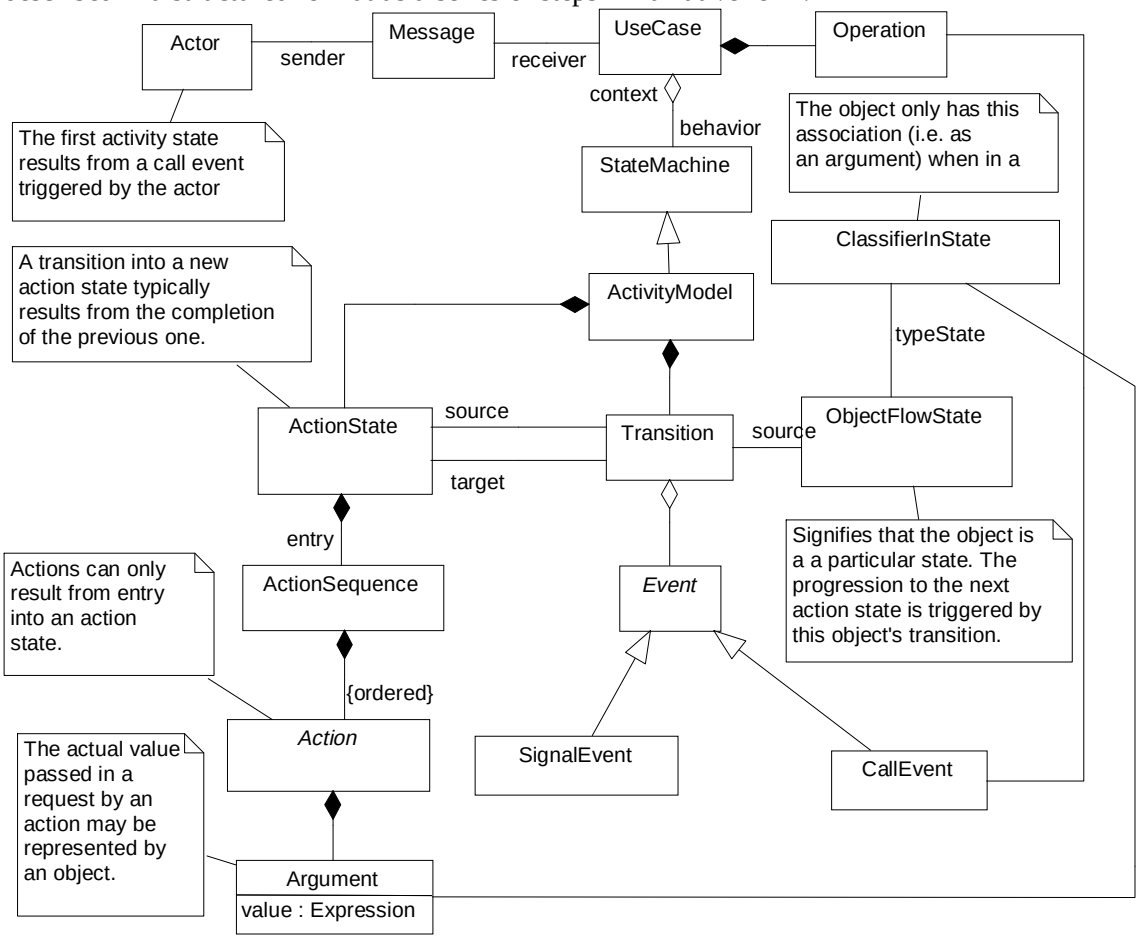

Figure 3.3 : Use Case Realization – Activity Model

The inclusion of object-flow states in an activity model extends the normal semantics of data flow in business process models. Since these object-flow states correspond directly to a classifier, the data is represented as attributes of an object. Values attached to this set of attributes represents specific states. These states may form the trigger for invoking an operation belonging to the entity being modeled that requires the object (i.e. the *ClassifierInState*) as input. Thus, object flow states are not stateless, passive data definitions as are data stores in traditional process data flow models.

The inclusion of the extended semantics afforded by the object-flow states is appropriate when the same object is manipulated by a number a successive actions. The change of state of the object being manipulated is the trigger for the next action sequence rather the completion of the action sequence doing the manipulation. When the change of state is the last action, there is no conceptual distinction between the two. The only semantic difference is the source of the transition.

One other optional extension to the semantics of the activity model involves dividing the states of the activity model into groups. This is appropriate when the system being modeled represents interaction with several organizational units that can be used to partition the business model. When combined with object-flow states, the change of state of the manipulated object can be used to track its movement among the functional boundaries defined for each partition.

The combination of object-flow states and partitions provides effective modeling constructs for use cases that involve multiple actors. If participation from a secondary actor is required to complete a use case for the initiating actor, then each such secondary actor represents a potential partition for the activity model. It is not uncommon in such scenarios for the same object to be manipulated by several such actors, especially when a system is being modeled within the context of a larger business process model.

1. Use Case Refinement

The essence of use case refinement is to gather all of the use cases for each modeling element contained within the system and model the collaborations among these elements, i.e. subsystems and classes. Thus, use case refinement parallels the refinement of each model element in the package hierarchy. This parallel shows that collaborations can be used to specify both refinement and realization of a use case.

By definition, a subsystem may be divided into a specification part and a realization part. The inclusion of a use case in the specification part will determine whether operations will be attached directly to the subsystem or indirectly through the use case. The model in Figure 4.1 graphically depicts the recursive nature of use case refinement where the use case for the whole entity being modeled at a given level of abstraction is superordinate to its refining use cases, which in turn are *subordinate* to the first one. There is full traceability of functionality between the superordinate use and each of its subordinate use cases, which cooperate to perform the superordinate one. Their cooperation is specified by *collaborations* and may be presented in collaboration diagrams. Collaborations among subordinate use cases are specified in an actor/use case relationship. Moreover, all actors of a superordinate use case must appear as actors of one or more subordinate use cases.

A class may be specified by a use cases in terms of the operations of the classes. A simple use case may consist of a single operation invocation on the class. When more than one class operation is involved, the use case may describe a well-defined sequence to the invocation of operations on the class. Class operations may appear in several use cases in order to specify the various services offered by the class.

The owning element of a use case depends on the kind of model element it specifies. A use case for a subsystem is owned in the specification part of the subsystem model element. However, it also directly owns the use cases that specify the classes that it owns in the realization part. When a use case of a subsystem involves only one contained class, then the use case for subsystem and the use case for the class are conceptually equivalent, possibly identical[2]. However, since a use case for a subsystem may only provide behavioral specification without revealing internal structure, there is no guarantee that this equivalence exists.

For example, subsystem A contains subsystem B as a model element. Subsystem B, in turn, contains classes C, D, E, and F as model elements. Subsystem B has two use cases U1and U2. Use case U1 is realized through a collaboration of classes C, D, and E. The use cases for these three classes are hidden from subsystem A. Use case U2 is completely realized by a series of operations invoked on class F. Although the superordinate-subordinate relationship between U2 and the use case for class F is a one-to-one mapping of equivalent use cases, this refinement may still be necessary to hide the implementation details with respect to class F. Moreover, class F may contain additional use cases that are not part of the desired functionality offered by subsystem B[3].

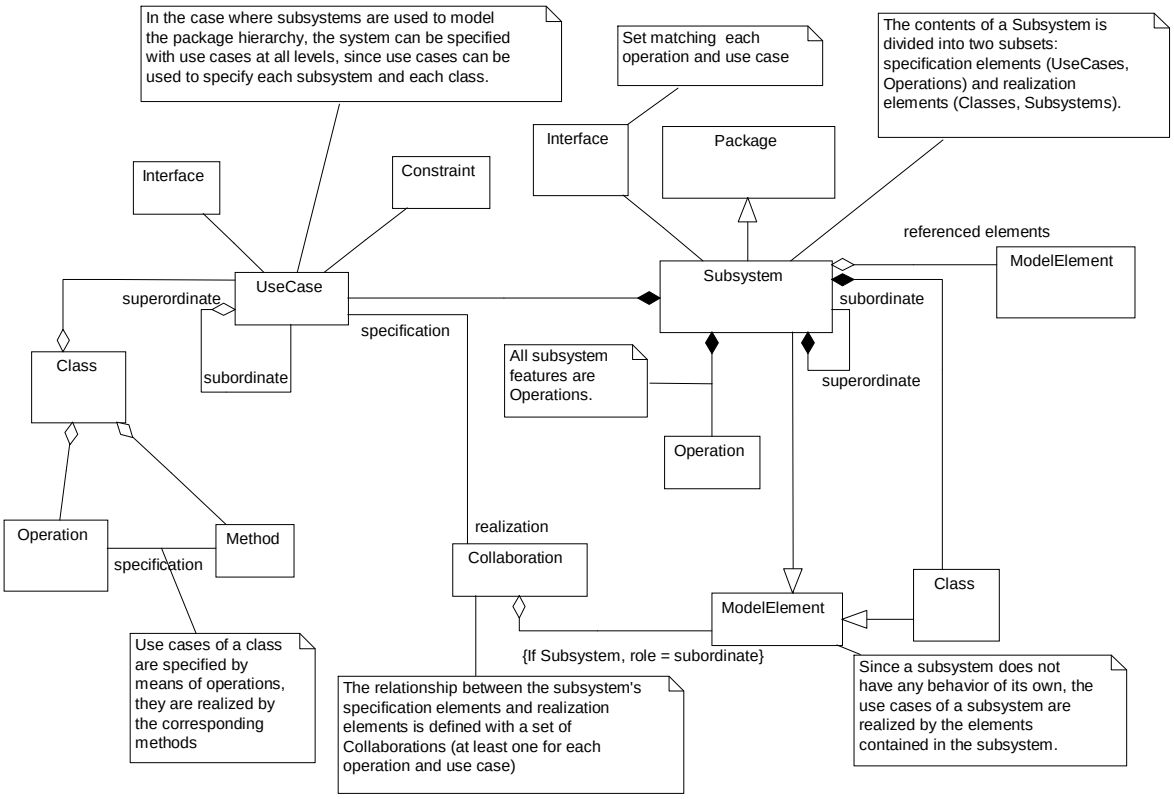

Figure 4.1 : Use Case Refinement

[2] The UML is vague as to the precise semantics between a use case and a class with respect to an owning subsystem. However, it does state: "The specification of the functionality of the system itself is usually expressed in a separate use-case model, i.e. a *Model* stereotyped «useCaseModel». The use cases and actors in the use-case model are equivalent to those of the system package.

[3] Since use cases and classes both inherit from generalizable element, services not needed in the subsystem may exist in both the class and the use case. However, descendents of these classes located in other subsystems may rely upon these services.

It is possible that a fully specified class in terms of use cases may not have a corresponding role in a subordinate use case collaboration or mapping to a subsystem level use case, although these use cases are owned by the subsystem.

2. Use Case Reification

A use case is reified through a use case instance that constitutes the performance of the sequence of actions specified in a scenario of the use case. One explanation as to how and why reification of a use case is accomplished was provided in version 3.0 of Rational Rose in 1995. As the object-oriented analysis and design tool offered by the principle shapers of the UML specification, the tool's application notes offer some insights into the concept of use case reification [Jans95]. It should be emphasized UML makes no direct reference to reification. It is merely permissible under UML semantics. Therefore the concepts presented in this section are debatable and even treatment of the subject of reification can be considered controversial.

The general approach presented by Rational Rose was to promote the concept of use cases as classes. This begins with making the use case model into a class category. Each use case is an operation of the class *System*. *AllActors* become another class. Each actor is an instance of *AllActors*. A class is then created for each use case operation on *System*, each scenario is added as an operation of this class. An actor object starts a scenario by sending a use case message to the System, which in turn instantiates a use case object. The second message sent by the actor is the scenario name. It is sent to the instantiated use case object. Additional objects can then be instantiated by the scenario that are an actual part of the system. The actor interacts with these new class objects normally, but also can send additional use cases messages (to *System*) or scenario messages (to *UseCase* objects).

The explanation as to why use cases should be made into classes is explained in the tool documentation as follows. "From use case classes, you have traceability links to the test source code …One of the benefits with use cases is that they can be used as the basis for system and function tests. You can generate code from classes. That means that it is also possible to generate code from the use case classes. That code can be used as system and function test drivers."

The approach described above is appropriate when the chosen scenario can be determined in advance. However, when the actual scenario that is performed is determined by elements outside the control of the initiating use case, this approach breaks down. Accordingly, the approach modeled depicted in Figure 5.1 ignores the additional operation on the use case object. Instead, the actual communication protocols between the actor and use case are reified in precisely the same manner as specified in the use case model. Instead, a state machine is reified as part of the use case instance to perform actions necessary to determine the appropriate messages to send to actual system objects. If the entire system being modeled consists of automated processes, then the reified use case instance degenerates to the equivalent state machine realization describe in Section 3. However, if the system combines manual business processes with an automated system, then determining which scenario actually gets invoked may provide valuable control information for workflow purposes.

A use case may be considered an aggregation of scenarios that represent a basic course of action and any variants. Rather than require the actor to know in advance which scenario should be performed, the use case instance records the actual scenario as part of its state. In this manner, a reified use case instance that is persistable provides both a record of execution and an explanation facility as to why a particular scenario was performed. Combining the use of use case refinement with use case reification would allow an execution record to the desired level of detail. It is likely

that some of the refined use cases represent completely automated systems. In such a case, a specialized version of the state machine realization may be necessary to persist the desired state if such a record was only required for development and testing purposes.

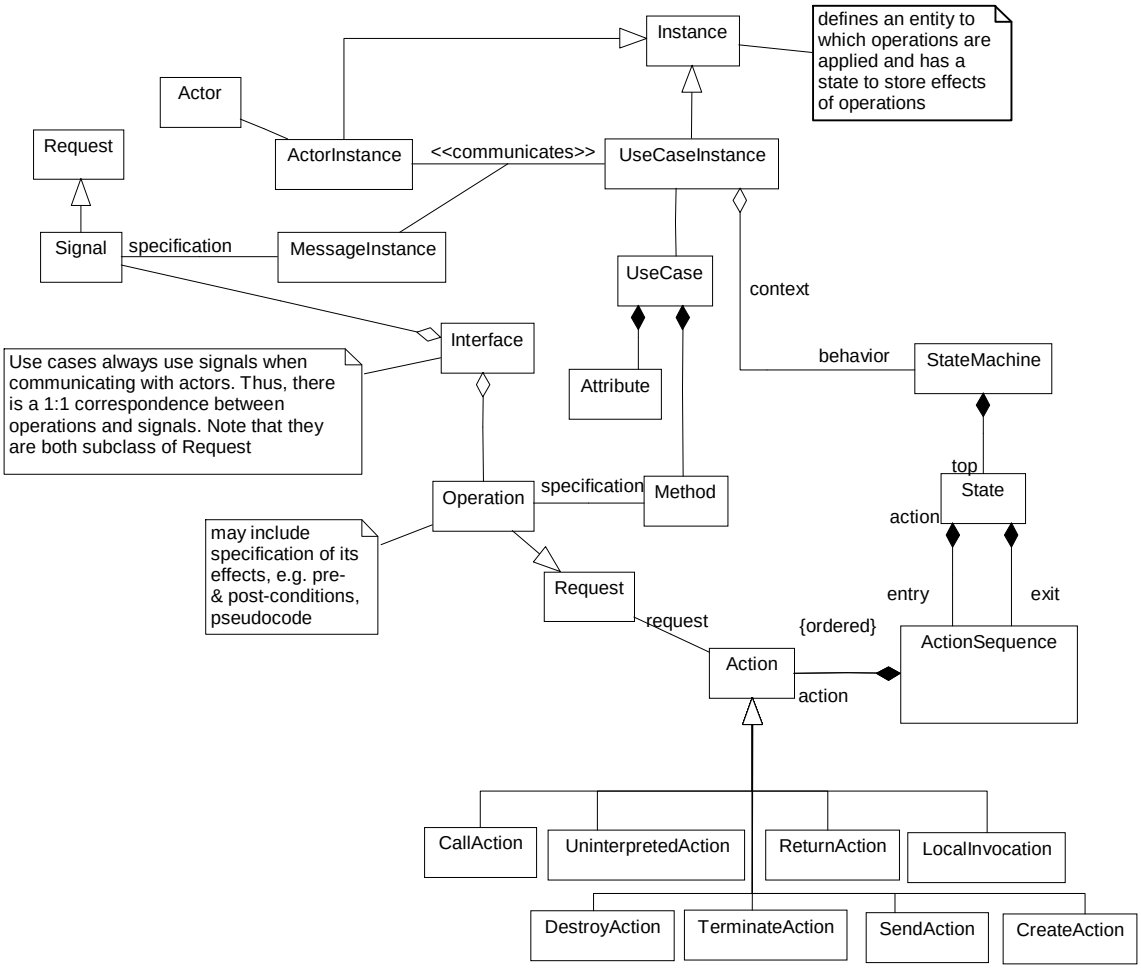

Figure 5.1 : Use Case Reification

3. Guidelines for Usage of Use Case Formalisms

This section offers guidelines on appropriate uses for the various representation schemes presented in the previous sections. At some point, a use case must be realized through one of the three mechanisms discussed. Use case refinement and use case reification each presents opportunities to leverage development effort under certain circumstances.

While this paper does not prescribe the representation scheme for use cases themselves, the choice of representation can have an impact of the selection of the approach for realization. For example, when a state machine or activity model is used to describe a use case through protocol specifications, it can usually easily map to a design model of the same type.

The following points are intended to provide some guidance in the modeling of use case formalisms with respect to realization, refinement, and reification. They summarize suggestions from the previous three sections as well as provide additional guidance and rationale:

- Maintain the natural separation between analysis and design. Use cases provide the requirement specifications for the system being modeled. Realizations are design models. If complex states or sequencing of actions are needed to adequately describe the problem domain, then these constructs should be represented in use case analysis models. On the other hand, if such states or sequencing merely constitute implementation decision and strategies, they should generally be left out of use case models.
- Collaborations are a preferred choice for use case realization for subsystems. However, a state machine may be more appropriate for subsystems that contain a single active object. An active object is an object that owns a thread and can initiate control activity and run its thread of control concurrently with other active objects. With only one active object, all of the other objects are passive, meaning they can only send messages when processing a received request.
- If the system being modeled is being implemented from an existing domain architecture, realization of the use case through a collaboration will probably best leverage the existing domain assets. This is especially true for mature object-oriented framework and well-defined design patterns that provide parameterized prototypes for the collaborations.
- A state machine than contains an attribute or link to an object that maintains a history of transitions can be used to reify a use case. When such a state machine provide the realization for a lower level subsystem or class, then providing the query capability for this history may be sufficient in lieu of declaring the owning classifier persistent.
- If functional requirements specify the need for concurrent threads of control, realization of the use case through collaborations of the active objects may provide a valuable alternative perspective of the complex behavioral patterns better suited for developers (vs. users).
- When there is only a single thread of control and the system being modeled at the current level of abstraction involves both manual and automated processing, an activity model provides an appropriate realization of a use case.
- When the use case scenario that is being performed is not known in advance, then modeling and implementation of use case reification should be considered. Stakeholders of the system may have a need to know which scenario was performed after the fact. This capability is especially important for auditing and validation purposes.
- When a single passive entity provides the basis for sharing among collaboration participants for many different use case realizations, an activity model with object-flow extensions may be a good choice. Using the terminology contained in the UML Extension for Business Modeling, these *entities* are usually persistent. They are produced, manipulated, and accessed by *workers* and automated *work units*.
- When a use case for a system represents interaction with several organizational units, use of an activity model with partitions can be effectively used. If the activity model support the reification of the use case, then the change of states can be used to track processing and movement among the functional boundaries defined for each partition.

1. Conclusion and Summary

Use cases provide a means to drive requirements and improve communications with end users. Realization of use cases present a view of use cases necessary for the implementation of the requirements. Although the UML specification has already gained widespread support within the software development community, it exhibits a considerable degree of complexity [Mele97]. In

order to reach critical mass of acceptance as a standard, capitulation to incorporate contributions from a large base of methodologists has fostered this complexity. Numerous proposed extensions promise to further increase this complexity. As far as UML has come, there remains considerable latitude in interpreting the significance of the defined static and dynamic semantics of the model elements.

This paper has attempted to analyze the current UML version 1.1 within a narrower context of use cases. It provides focused models that highlight the key relationships involved in the realization, refinement, and reification of use cases. Areas where the specification is vague and in need of more precision have been discussed. Moreover, semantics that are not explicitly stated in the specification were presented. Every attempt was made to include consideration of representations allowed under the UML specification in order to explore the full range of afforded degrees of freedom. Guidelines were then presented in order to help reduce the choices in hope of providing consistency in models created and exchanged among developers and other stakeholders of the systems being developed.

Since the models presented in this paper capture the state of the UML specification at a particular point in its development, they will likely require revision as the specification further evolves. Specifically, as additional well-formedness rules are defined to constrain the use of modeling elements, the degrees of freedom will inevitably be reduced. Furthermore, none of the modeling approaches described is without controversy. Although realization and refinement are well entrenched concepts of the current UML specification, the use of collaborations for use case refinements is a relatively new concept that adds considerable formalism where very little previously existed. Reification of use cases is a particularly controversial concept. Whereas it is quite acceptable to conceptually grasp the performance of a use case as an instance, to reify this concept as a class is much harder to accept. Each of these concepts has some conceptual overlap, but it could be argued that use case reification is nothing more that a convoluted realization mechanism that went awry.

Russ Hurlbut is a principal of Expertech, Ltd., an information systems consulting organization specializing in the development of business domain-specific architectures. He has taught graduate courses in Object Oriented Development and Technology Transfer at the Illinois Institute of Technology. This paper represents one aspect of his research towards his Ph.D. Thesis "Managing Business Domain Architectures through Use Case Formalisms."

References

[Hurl97c] Russ Hurlbut, "A Survey of Approaches for Describing and Formalizing Use Cases", Technical Report: XPT-TR-97-03, Expertech, Ltd., 1997

[Jans95] Par Hansson, "Use Case Analysis with Rational Rose", Rational Rose Applications Notes Version 3.0, Chapter 4

[Mele97] Deborah Melewski, "UML – Ready for Prime Time?", Application Development Trends, Nov. 1997, Vol. 4, No. 11. pp. 30-44

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorenson, Object-Oriented Modeling and Design, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1991

[UML97] Unified Modeling Language, Version 1.1, http://www.rational.com/uml