Use Case Refinements in the Object Oriented Software Development Process

Roxana S. Giandini
giandini@sol.info.unlp.edu.ar

Claudia F. Pons
cpons@sol.info.unlp.edu.ar

and

Gabriela A. Pérez
gperez@sol.info.unlp.edu.ar

Universidad Nacional de La Plata, Facultad de Informática
LIFIA - Laboratorio de Investigación y Formación en Informática Avanzada
La Plata, Argentina, 1900

Abstract

Model Refinement is a dependency relationship that relates two elements that represent the same concept at different levels of abstraction. In the UML specification document this relationship, like others concepts, is still described in an ambiguous, informal way.
In order to avoid inconsistencies and wrong model interpretations, in this article we propose, in first instance, a formalization of the Use Case specification, represented by a conversation between an actor and the system. The Use Case conversation does not have representation in the UML metamodel. In second instance we propose to formalize the refinement relation between model elements of the same kind, such as refinement relation between Use Cases and between Collaborations. Then on top of these formalizations, we discuss refinement relation between models of different kind (use case models and collaboration models realizing them)
This work provides an enhancement to the UML metamodel specification. The formalization proposed should be used as a formal foundation for the construction of case tools performing consistency checking of models. Support offered by tools will improve the quality of software development process.

Keywords: Object Oriented Analysis and Design, Unified Process, Unified Modeling Language, Use Cases.

1. Introduction

A software development process, e.g. The Unified Process [10], is a set of activities needed to transform user's requirements into a software system. Modern software development processes are iterative and incremental, they repeat over a series of iterations making up the life cycle of a system. Each iteration takes place over time and it consists of one pass through the requirements, analysis, design, implementation and test activities, building a number of different artifacts (i.e models). All these artifacts are not independent; they are related to each other, they are semantically overlapping and together represent the system as a whole. Elements in one artifact have trace dependencies to other artifacts. On the other hand, due to the incremental nature of the process, each iteration results in an increment of artifacts built in previous iterations.

Different relationships existing between models can be organized along the following three dimensions (as we proposed in Pons et al. [19, 20]:

- internal dimension (artifact-dimension).
- vertical dimension (activity -dimension)
- horizontal dimension (iteration-dimension)

Figure 1 illustrates the three dimensions describes above.

Relations between models should be formally defined since the lack of accuracy in their definition can lead to wrong model interpretations and inconsistency among models.

At the present the Unified Modeling Language UML is considered the standard modeling language for object oriented software development process. The specification of UML constructs and their relationships [21] is semi-formal, i.e. certain parts of it are specified with well-defined languages while other parts are described informally in natural language. There is an important number of theoretical works giving a precise description of core concepts of UML and providing rules for analyzing their properties; see, for instance the works of Evans et al. [5, 6], Kim and Carrington [11], Breu et al.[1], Knapp [12], Övergaard [14, 15], Pons and Baum.[18], Cibrán et al. [2]. These works improve precision of syntax and semantics of isolated UML models, without dealing with relationships between models.

In addition, Övergaard and Palmkvist [13, 16], Petriu and Sun [17], Sendall and Strohmeier, [22], Whittle et al. [23], Egyed [4] and Giandini et al.[8] focus on relationships between different UML models.

In our approach, we propose a formalization of a frequently occurring kind of relationships between models: "*Model Refinements*". Model Refinement is a dependency relationship that relates two elements that represent the same concept at different levels of abstraction.

Model Refinement is carried out in different ways: for example, on the vertical dimension, analysis models are refinements of use case models, but on the other hand, on the horizontal dimension, models built in an iteration are usually refinements of models (of the same kind) built in previous iterations.

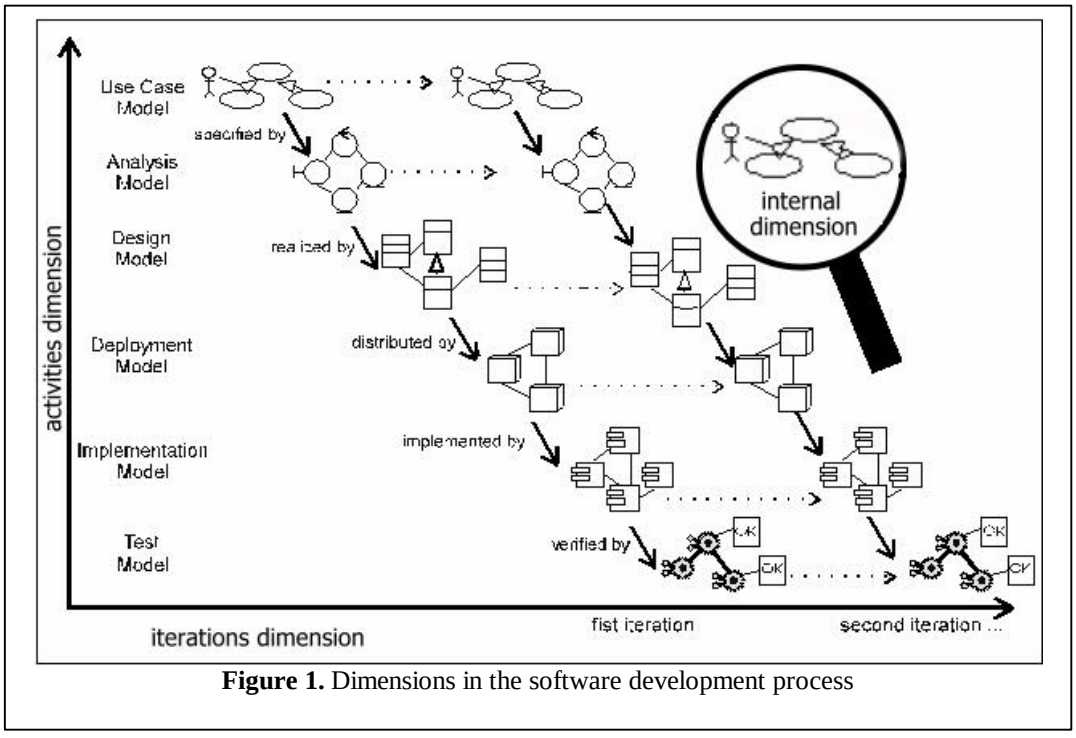

Figure 1. Dimensions in the software development process

Nature of internal, horizontal and vertical refinement is quite different. In section 2 of this article we study and propose changes in the UML metamodel restricting the Use Case specification (on the internal dimension). Then, on the horizontal dimension we analyse refinement relation between model elements of the same kind, in particular refinement relation between Use Cases and in section 4 refinement relation between Collaborations. In section 3, we discuss refinement relation on the vertical dimension (i.e. Collaboration refining Use Cases). After that, on top of this formalization we study consistency checking between models in different dimensions. Finally, we present some conclusions.

2. Use Cases

A Use Case describes one service provided by a system, i.e. a specific way of using the system. The complete set of Use Cases specifies all possible ways in which the system can be used, without revealing how this is to be implemented by the system. This makes Use Cases suitable for defining functional requirements in the early stages of system development, when the inner structure of the system has not yet been defined.

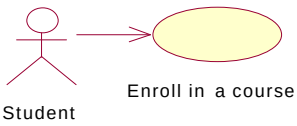

Figure 2. Use Case Enroll-in a course

A Use Case specifies a set of complete sequences of actions, which the system can perform. A user of the system initiates each sequence, and it includes the interaction between the system and its environment as well as the system's response to these interactions.
In UML a use case is a Classifier, then it is composed by operations (generally, a use case is composed by only one operation[1]).

2.1 Internal Dimension: Use case specification

Use cases can be specified in a number of ways. Generally natural language structured as a conversation between user and system is used, see [9]. The conversation shows the request of a user and the corresponding answers of the system, at a high level of abstraction.

Example: A University Department wishes to automate the registration of students to courses. A student may take a maximum of five courses. Each course has a pre-requisite set of courses that must have already been passed before the course may be taken.
A student may register with a course unit and the registration will be accepted provided that they have already passed the necessary pre-requisite courses and that they are not already taking five course. There is a maximum size of thirty students to each course.

The figure 2 illustrates the use case for this example. The figure 3 shows a conversation between an actor (an student) and the system. The conversation considers the normal action sequence and also alternative sequences (e.g. the case in that the student doesn't fulfill the conditions).

Actor Actions	System answers
1- Enroll in a course	
	2- Get pre-requisite of the course
	3- Validate Student's conditions
	4- Validate Course state
	5- Save The Inscription

Alternative threads
2.1- The student does not fulfill the conditions -> Reject
3.1- The course is closed -> Reject

Figure 3. The conversation of the Use Case Enroll in a Course

Each sequence represents a possible scenario of execution of the use case. Then, the complete description of a use case is composed by an scenario sequence, i.e. a sequence of action sequences, where some of them can be simple action.
In the UML metamodel, an action sequence is an instance of the ActionSequence metaclass.

Figure 4 shows part of this metamodel, where we can see that the ActionSequence metaclass is subclass of the Action metaclass. This fact generates some conflicting situations:

- an ActionSequence could have arguments

[1] In general we abbreviate op.method.body by op.actionSequence. The body of a method is a procedure expression specifying a possible implementation of an operation. The definition of procedure expressions is out of the scope of UML, here we interpret a procedure expression as a set of action sequences.

- an ActionSequence could have an associated message

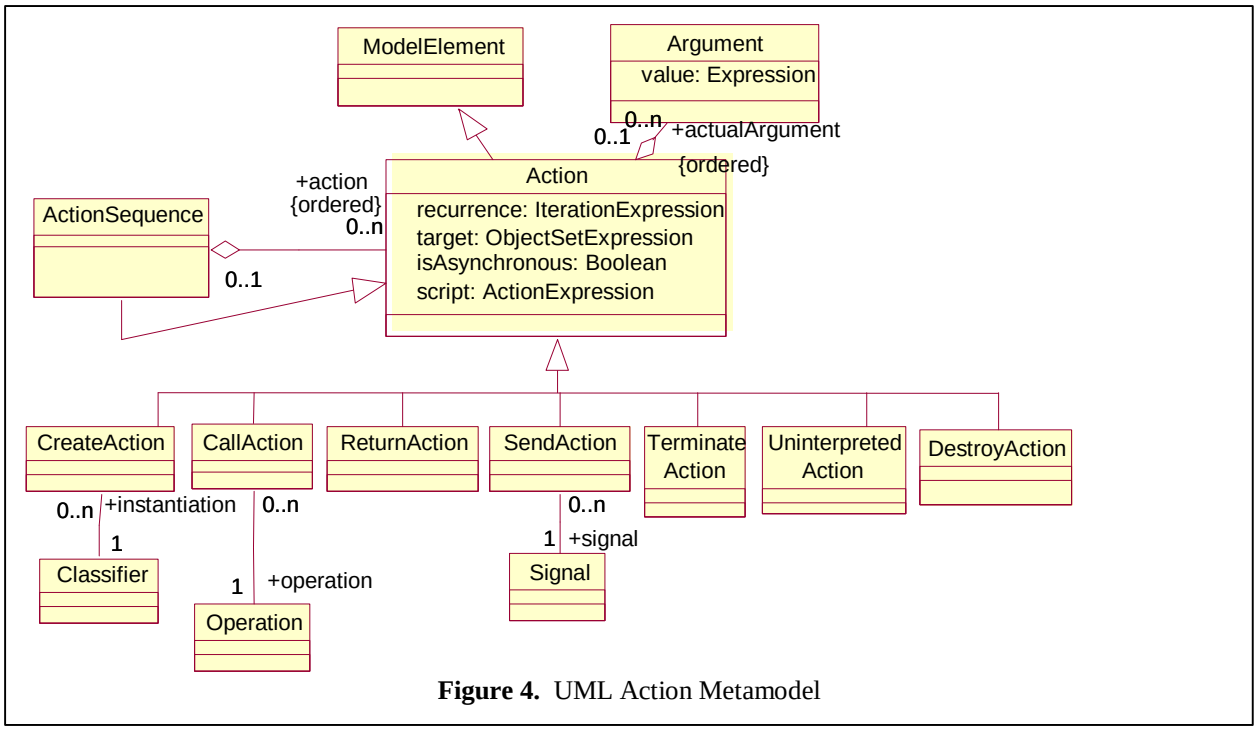

Figure 4. UML Action Metamodel

As a solution for these problems, we propose a new metamodel where the metaclass Action is sub classified with both CompositeAction and SimpleAction subclasses. A CompositeAction will be composed by actions, which could be acceded by the actionSequence association, as we can see in figure 5 (this schema follows the pattern Composite [7]).

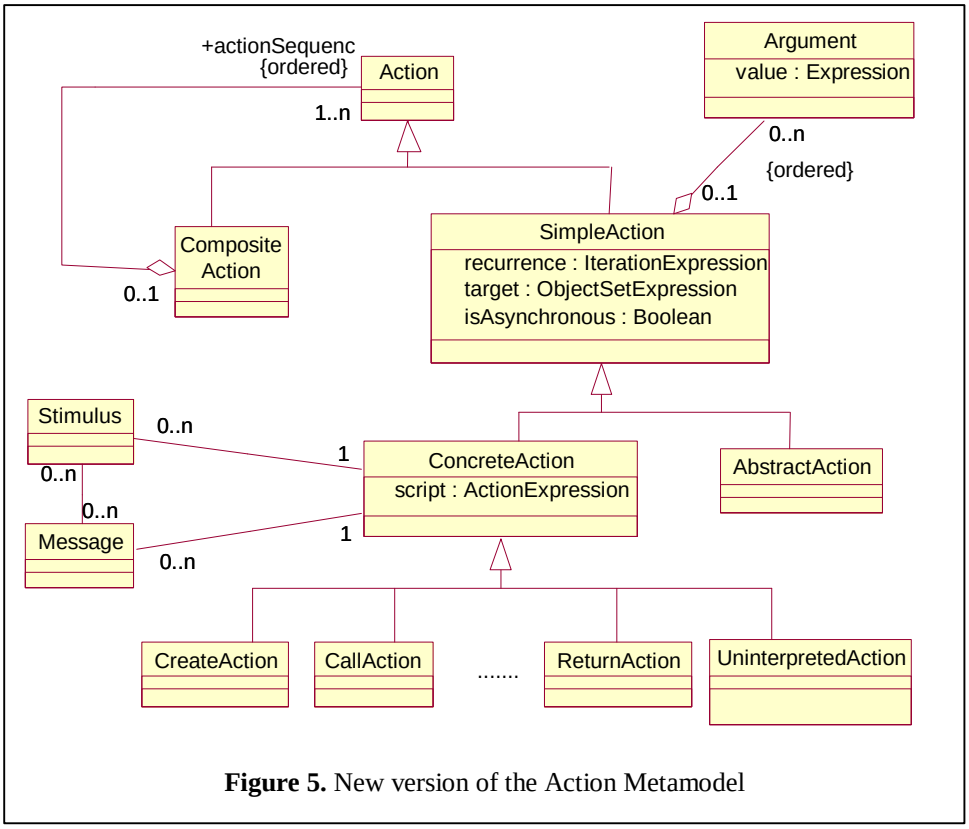

Figure 5. New version of the Action Metamodel

On the other hand, underlined actions[2] can appear in a use case conversation. The meaning of an underlined action is that it will be refined. It is interesting to be able to distinguish both concrete actions (i.e. atomic actions, that will not be refined) and abstract actions (i.e. actions that require a refinement). This situation neither is represented in the UML metamodel. We propose to sub classify a SimpleAction in both ConcreteAction and AbstractAction. ConcreteAction will be the superclass of the concrete actions that were defined in the metamodel until now (i.e. CallAction, CreateAction, etc.), while an AbstractAction will specify those actions that will be refined. These new metaclasses improve formality of the Use Case metamodel allowing for the definition of the Use Case refinement hierarchy, as we will see in next section.

[2] The notation used for conversation is based in [3]

Analyzing the specification of a Use Case

Each action sequence representing a use case scenario can be specified by an instance of either SimpleAction metaclass or CompositeAction metaclass. In case of the sequence is a CompositeAction instance, we suggest that only could be formed by SimpleAction (ConcreteAction or AbstractAction). Otherwise, it would be specifying a nesting of CompositeAction in the same use case, that reflects a mixture of abstraction levels into the specification. To express that an action will be refined we use an AbstractAction.
Also, in the use case context, we can restrict the type of concrete actions. We propose that they must belong to the UninterpretedAction metaclass because use cases show only external behavior of the system, without revealing any internal detail. Due to this, a use case scenario should not be specified through ConcreteAction such as CallAction, DestroyAction, etc.
These ideas can be expressed more formally defining some well formedness rules in the Use Case metaclass.

[1] Each CompositeAction must be formed only by SimpleAction.

```
self.operation.actionSequence ->
select (acc | acc.oclIsKindOf(CompositeAction)) ->
collect (ca | ca.actionSequence) ->
forAll (action | action.oclIsKindOf(SimpleAction))
```

[2] All ConcreteAction are UninterpretedAction

```
self.concreteActions -> (forAll ca |
                         ca.oclIsKindOf(UninterpretedAction))
```

2.2 Horizontal Dimension: Use Case Refinement

In the UML specification document [21], the use case refinement concept is defined as follows:
"In the case where subsystems are used to model the system's containment hierarchy, the system can be specified with use cases at all levels. A use case specifying one model element is then refined into a set of smaller use cases, each specifying a service of a model element contained in the first one. The use case of the whole may be referred to as superordinate to its refining use cases, which, correspondingly, may be called subordinate in relation to the first one. The functionality specified by each superordinate use case is completely traceable to its subordinate use case"
According with this, it is expected that all actions in the superordinate use case are refined (i.e., each action will be specified in more detail through a new subordinate use case[3]). Thus, from a bottom-up perspective, the composition of all subordinate use cases results in the complete functionality of the superordinate one.

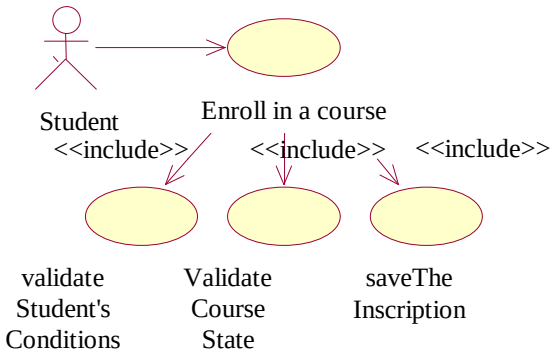

Figure 6. Use case refinement tree.

But, in most practice cases, this does not happen. It is very common that actions in a use case do not belong to the same abstraction level. We can see in the Enroll-InACourse use case in the example (see Figure 3), that not all the actions are refined. In the conversation, the actions 2 and 4 correspond to atomic actions which does not need to be refined. However, the rest of the actions will be refined by subordinate use cases. Figure 6 shows the refinement of Enroll-InACourse use case, forming a refinement tree.

Therefore, the requirement of complete traceability from superordinate use case to its subordinate use cases is too restrictive and it does not suit most real life cases.
We believe this requirement should be loosened, in the following way: "a superordinate use case is composed by a sequence of scenarios. Each scenario is formed by a sequence of actions. Some of these actions can be atomic (i.e. not amenable of refinement), while others can be refined by a subordinate use case".
A group of use cases related by refinement relationship form a hierarchical structure where each subordinate use case is a refinement of a non-atomic (AbstractAction) action belonging to its superordinate use cases.
The UML specification document does not include any constraint to regulate Use Case Refinement.
To specify well formedness property of use case refinement hierarchy, we define new well formedness rules expressed in OCL [21], on the metaclass Use Case:

[3] In the use case model, a way to express refinement is by the *include* relationship (page 2-143 in [21])

[1] For each AbstractAction must exists a subordinate Use Case with the same name.

```
self. abstractActions -> forAll (ab |
                        self.include.addition -> exists ( subUC | subUC.name =
                        ab.name)
                                  )
```

[2] For each subordinate Use Case must exists an Abstract Action with the same name.

```
self.include.addition -> forAll (subUseCase |
self.abstractActions -> exists (acc | acc.name = subUseCase.name))
```

Additional Operations in the metaclass Use Case

[1] The operation abstractActions results in a Set containing all AbstractAction of the Use Case

```
abstractActions: Set (Abstract);
abstractActions = (self.operation.actionSequence ->
                     select (sa | sa.oclIsKindOf(SimpleAction)))
                     union
                     (self.operation.actionSequence ->
                     select (ca | ca.oclIsKindOf(CompositeAction ))->
                     collect (ca | ca.actionSequence) )

                    ->
                    select(action |action.OclIsKindOf(AbstractAction))
```

3. Vertical Dimension: Collaborations realizing Use Cases

A use case in the use-case model is realized by a collaboration within the analysis model that describes how a use case is realized and performed in terms of analysis classes and their interacting analysis objects.

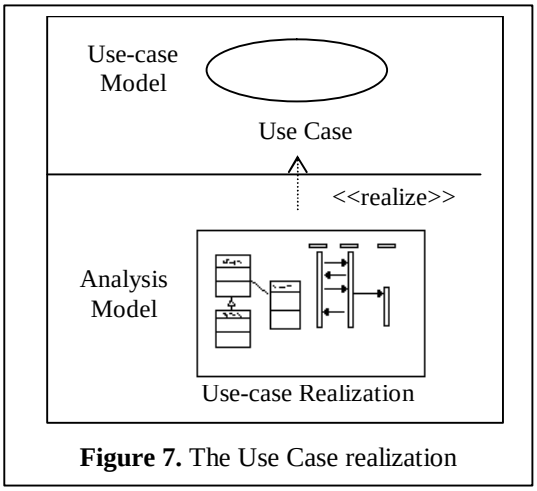

Figure 7. The Use Case realization

This realization relation was formalized in a previous work [19]. A use case realization has class diagrams that depict its participating analysis classes, and interaction diagrams that depict the realization of a particular flow or scenario of the use case in terms of analysis object interactions. Figure 7 shows the relation between a use case and its realization.

The figure 8 shows part of the Class Diagram at analysis level for this example.

In Figure 9 we can see a Collaboration model including a set of Classifier Roles and their connections and one of the iteration diagrams specifying the message flows between objects playing the roles in the Collaboration. These diagrams are expected to realize the use case above.

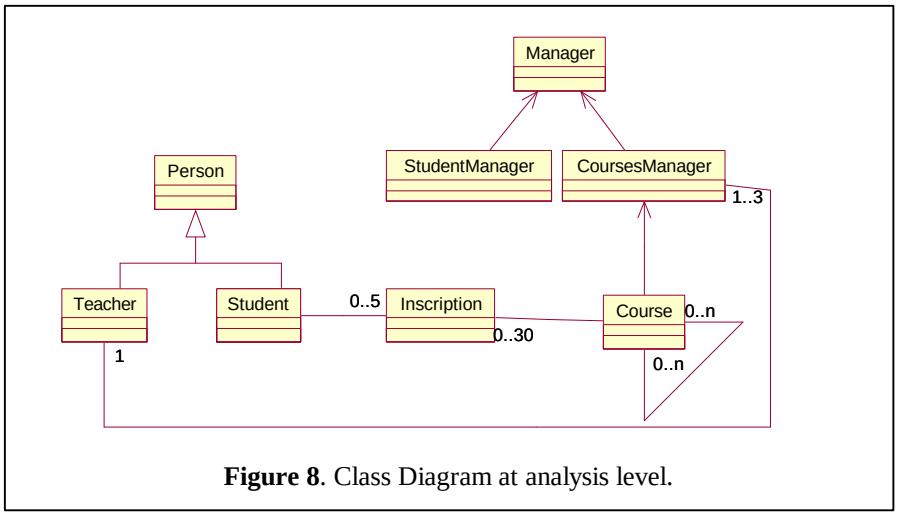

Figure 8. Class Diagram at analysis level.

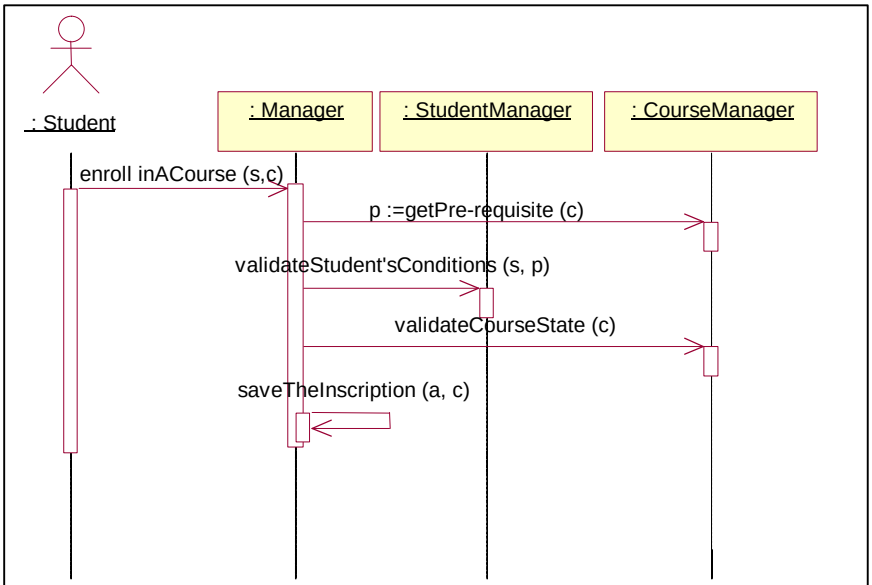

Figure 9. Collaboration for the use case Enroll-inACourse

4. Horizontal Dimension: Collaboration Refinement

Collaborations as well as use cases can be refined through subordinate collaborations, forming a refinement hierarchy. Each subordinate collaboration implements in more detail one part of the global functionality and can have its own sets of roles and interactions. Subordinate collaborations can be referenced in the UML metamodel by the attribute usedCollaboration in the superordinate collaboration.
Figure 10 illustrates the refinement tree[4] of the collaboration Enroll-inACourse presented before.

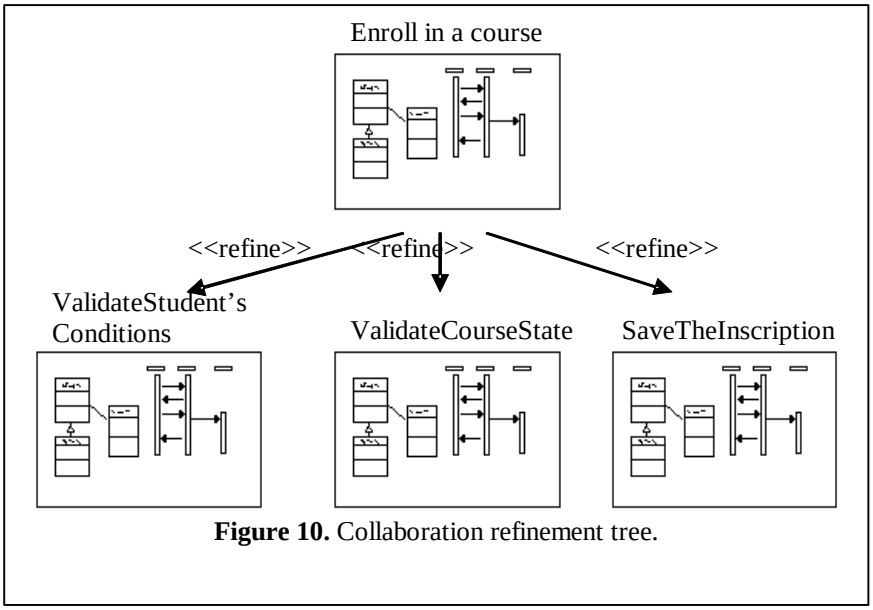

Figure 10. Collaboration refinement tree.

In order to formally specify the refinement relation between collaborations we define additional well formedness rules on the metaclass Collaboration.
In the case that a collaboration refines a message, we propose a standard format for such collaboration that allows us for a formal verification of well formedness of the refinement relation.

[4] Collaboration Refinement has not an explicit notation in UML. We use a dependency relation with <<*refine*>> stereotype to show it. To reference the subordinate collaborations UML metamodel defines the attribute usedCollaboration in the Collaboration metaclass.

In order to understand this format it is necessary to consider that the sender of a message has less importance than the receiver of this message. (See figure 11a)
The sender can be an instance of any ClassifierRole, however the receiver of the message is the responsible of the interpretation of that message.
For this reason, we propose that the subordinate collaboration that describes the refinement of the original message begins this refinement with an instance of the ClassifierRole that received this message in the superordinate collaboration. (see figure 11b)
In addition, considering that the only messages that can be refined are the associated to CallActions, (since the other actions are atomic), we propose that the name of each subordinate collaboration will be formed by both the name of

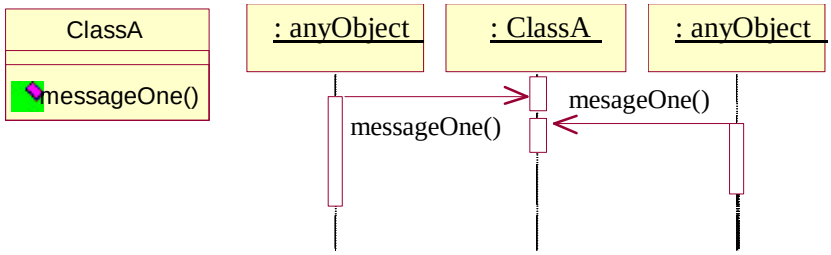

Figure 11a. Superordinate collaboration

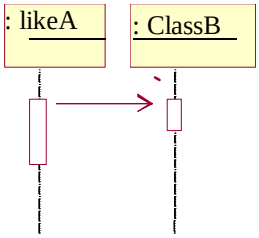

Figure 11b. Subordinate collaboration - messageOne() ClassA

the Operation associated to that instance of CallAction (associated to the message), and the name of the base Classifier that contains such operation.
In this way, examining the name of a subordinate collaboration we can associate it to at least one message in the superordinate collaboration, without ambiguity.
The figure 12 shows the subordinate collaboration, which refines the message saveTheInscription (s, c) of the superordinate collaboration Enroll-inACourse in the figure 9. We can see in this example that the subordinate collaboration name is formed by the concatenation of the Operation signature and the name of the owner Classifier of that Operation.
On the other hand, the ClassifierRole receiver of a refined message must maintain a relation with the ClassifierRole that sends the first message in each thread in the subordinate collaboration. It would be natural that in the refinement, we want to specify in more detail to the ClassifierRole in charge to that behavior.
The ClassifierRole (first sender) in the subordinate collaboration should belong to the same generalization hierarchy as the original Classifier. In case of that this Classifier being an interface the first sender must be a class that implements it.

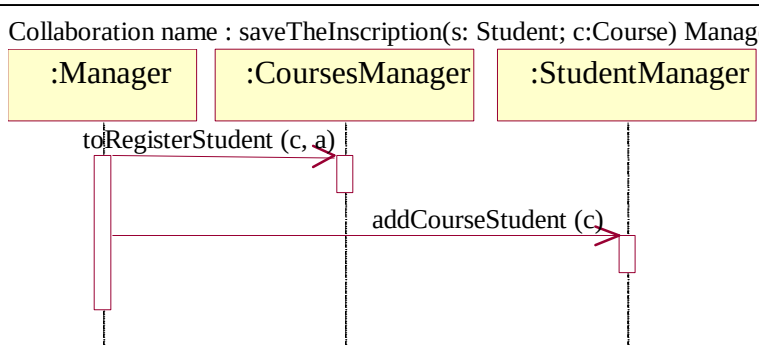

Figure 12. Subordinate Collaboration SaveTheInscription.

Well formedness requirements expressed above are formalizad by the following rules.

[1] Each interaction in each subordinated collaboration, must have as first sender's base a Classifier of the same hierarchy that some of the bases of the ClassifierRole receiver of the message in the superordinate collaboration. If the ClassifierRole's base in the superordinate collaboration is an interface, the first sender's base must implement this interface.

```
self.usedCollaboration -> forAll ( subcol |
  self.Interaction -> collect (int | int.message) ->
  exists (msg |
msg.action.operation.name.concat(msg.action.operation.owner.name) =
subcol.name and
(subcol.firstSender() -> forAll (cr |
```

```
                    cr.base.allParents.includesAll(msg.receiver.base)
                    or
                    cr.base.allSuppliers.includes(msg.receiver.base)
                    and msg.receiver.isOclKindOf(Interface))
        )
      )
```

[2] No Collaboration with the same name of Operation and the same name of firstSender can belong to one hierarchy.

```
Self.allSubCollaboration -> forAll (subCol1, SubCol2 |
     (subCol1.nameOperation = subCol2.nameOperation
     and subCol1.firstSender.base = subCol2.firstSender.base)
     implies subCol1 = subCol2)
```

Additional Operations in the metaclass Collaboration

[1] FirstSender returns a Set containing all ClassifierRoles that are the senders of the first message in each interaction.

```
firstSender :: Set (ClassifierRole)
firstSender = self.interaction ->
            collect (inter | inter.message) ->
            select (m | m.activator ->size()=0) ->
            collect (m | m.sender);
```

[2] AllSubCollaboration results in the Set of all subordinate collaborations of the collaboration, including all subordinate collaboration defined by transitive clousure.

```
allSubCollaboration : Set (Collaboration)
allSubCollaboration = self.usedCollaboration union
   (self usedCollaboration -> forAll ( sub | sub allSubCollaboration))
```

5. Consistency Checking between Models in different Dimensions

Since different models that are built during the software development process are related to each other along different dimensions, it is natural to perceive that interdependencies between dimensions could arise. For example, when a use case is realized by a collaboration, it is expected that some specific relations hold between their respective subordinated elements, as we can see in Figure 13.

UML does not specify any constraint regulating these relationships. We define new well formedness rules -on the Collaboration metaclass- that allow us to verify consistency between two different refinement hierarchies, as follows:

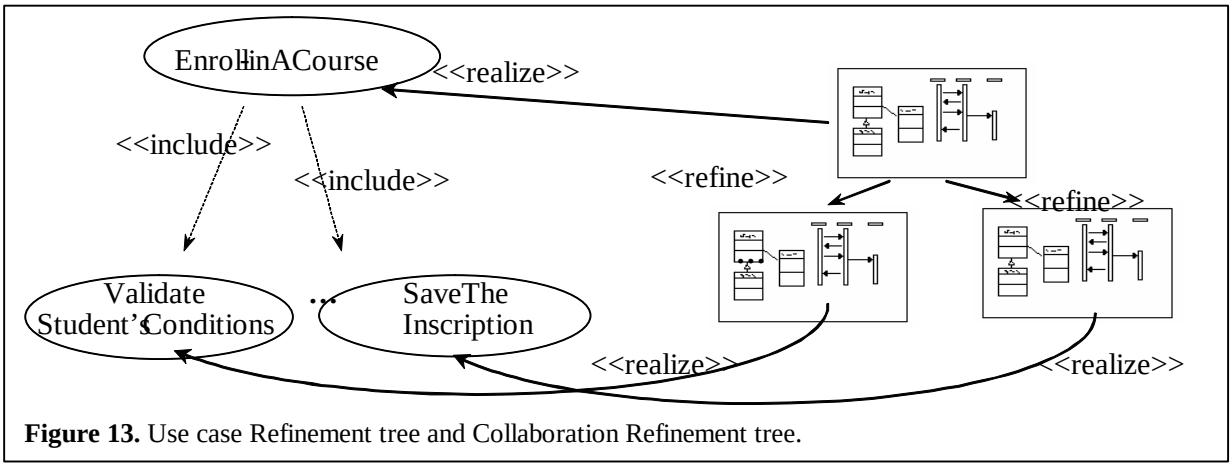

Figure 13. Use case Refinement tree and Collaboration Refinement tree.

[1] If a superordinate collaboration implements[5] a superordinate Use Case, then there must exist a subordinate collaboration implementing each subordinate Use Case.

```
self.representUseCase implies
 (self.representedClassifier.hasIncluded implies
      (self.representedClassifier.include.addition ->
      forAll (subUseCase | self.usedCollaboration ->
      exists (subcol | subcol.representedClassifier = subUseCase)
              )
      )
  )
```

Additional Operation of the metaclass Collaboration

[1] RepresentUseCase returns true if the collaboration implements a Use Case, and false in the other case.

```
representUseCase :: Boolean;
representUseCase = self.representedClassifier ->
                       select (cr|cr.isKindOf(UseCase))
                    -> size() >0
```

Additional Operation of the metaclass Use Case

[1] HasIncluded returns true if the Use Case has subordinated Use Cases, and false in the other case

```
hasIncluded :: Boolean;
hasIncluded = self.include -> size() > 0
```

6. Conclusions

In the UML specification document, several concepts are still described in an ambiguous, informal way. Such is the case of a dependency relationship between models known as: "Model Refinement". Model Refinement is a dependency relationship that relates two elements that represent the same concept at different levels of abstraction.

In order to avoid inconsistencies and wrong model interpretations, in this article we proposed, in first instance, a formalization of the Use Case specification, represented by a conversation between an actor and the system. The Use Case conversation did not have representation in the UML metamodel. In second instance we proposed to formalize the refinement relation between model elements of the same kind, such as refinement relation between Use Cases and between Collaborations. Finally, on top of these formalizations, we discussed refinement relation between models of different kind (use case models and collaboration models realizing them)

In particular, we defined well formedness rules in the OCL language, restricting Use Case specification as well as refinement hierarchy of both Use Cases and Collaborations. This approach can be extended to other models in the object oriented software development process (for example, refinement of class diagram and its relationships with other models).

The rules defined in this work form an enhancement of the UML metamodel specification. These rules should be used as a formal foundation for the construction of case tools performing consistency checking of models. Support offered by tools will improve the quality of software development process.

References

[1] Breu,R., Hinkel,U., Hofmann,C., Klein,C., Paech,B., Rumpe,B. and Thurner,V., Towards a formalization of the unified modeling language. ECOOP'97 procs., Lecture Notes in Computer Science vol.1241, (1997).

[2] Cibrán, M., Mola, V., Pons,C., Russo,W. Building a bridge between the syntax and semantics of UML Collaborations. In ECOOP´2000 Workshop on Defining Precise Semantics for UML France, June 2000.

[3] Cockburn, Alistair. Writing Effective Use Cases. Addison-Wesley (2001).

[5] The attribute representedClassifier in the Collaboration metaclass represents the Classifier (Class, Use Case, etc.) that the collaboration is realizing (page. 2-121 [21])

[4] Egyed, A. Scalable Consistency Checking between Diagrams- The ViewIntegra Approach. In Proceedings of the 16`IEEE International Conference on Automated Software Engineering(ASE), San Diego, USA, November 2001.

[5] Evans,A., France,R., Lano,K. and Rumpe,B., Towards a core metamodelling semantics of UML, Behavioral specifications of businesses and systems, Kluwer Academic Publishers, (1999).

[6] Evans,A., France,R., Lano,K. and Rumpe, B., Developing the UML as a formal modeling notation, UML'98 Conference, Lecture Notes in Computer Science 1618, Springer-Verlag, (1998).

[7] Gamma, Helm, Johnson, Vlissides, Design Patterns. Elements of Reusable Objects Oriented Software. Addison-Wesley, Professional Computing Series, 1995.

[8] Giandini,R, Pons, C and Baum,G.. An algebra for Use Cases in the Unified Modeling Language. OOPSLA´00 Workshop on Behavioral Semantics, Minneapolis, USA, October 2000.

[9] Jacobson, I., Christerson, M., Jonsson P. and Övergaard, G., Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, (1993).

[10] Jacobson, I..Booch, G Rumbaugh, J., The Unified Software Development Process, Addison Wesley. (1999)

[11] Kim, S. and Carrington,D., Formalizing the UML Class Diagrams using Object-Z, proceedings UML´99 Conference, Lecture Notes in Computer Sciencie 1723, (1999).

[12] Knapp, Alexander, A formal semantics for UML interactions, Proceedings of the UML´99 conference, Colorado, USA,. Lecture Notes in Computer Science 1723, Springer. (1999).

[13] Övergaard, G., and Palmkvist,K., A Formal Approach to Use Cases and Their Relationships. In UML'98 Conference, Lecture Notes in Computer Science 1618. Springer-Verlag, (1998).

[14] Övergaard, G., A formal approach to collaborations in the UML. In UML´99 Conference, Colorado, USA,. Lecture Notes in Computer Science 1723, Springer. (1999).

[15] Övergaard,G.. Using the Boom Framework for formal specification of the UML. in Proc. ECOOP Workshop on Defining Precise Semantics for UML, France, June 2000.

[16] Overgaard G.and Palmkvist K.. Interacting subsystems in UML, Proc. of The Third International Conference on the UML. LNCS. October 2000

[17] Petriu,D and Sun,Y Consistent behaviour representation in activity and sequence diagrams. Proc. of The Third International Conference on the UML. LNCS. October 2000

[18] Pons Claudia and Baum, Gabriel. Formal foundations of object-oriented modeling notations 3rd International Conference on Formal Engineering Methods, ICFEM 2000, IEEE Computer Society Press. Sept. 2000.

[19] Pons , Claudia, Giandini, Roxana and Baum, Gabriel. Specifying Relationships between models through the software development process, 10th International Workshop on Software Specification and Design, USA, IEEE Computer Society Press. Nov. 2000.

[20] Pons, C, Giandini R., Garbi J., Mercado P., Baum G.. Dimensions in the Object Oriented software Development Process. In Proceeding of the 2002 IRMA International Conference, UML and UP Track , Renaissance Madison Hotel, Seattle, Washington,USA, May 19-22, 2002

[21] Unified Modeling Language (UML) Specification - Version 1.4, September 2001. UML Specification, revised by the OMG, http://www.omg.org.

[22] Sendall, S. and Strohmeier, A. From Use cases to system operation specifications.. Proc. of The Third International Conf. on the UML, UK. LNCS. Oct 2000

[23] Whittle, J.. Araújo, J.Toval, A Fernandez Alemán J.. Rigorously automating transformations of UML behavioral models, UML´00 Workshop on Semantics of Behavioral Models. UK, October 2000.