Formalizing Relations between Use Cases in the Unified Modeling Language

Roxana Giandini Claudia Pons Gabriel Baum

LIFIA, Universidad Nacional de La Plata

Calle 50 esq.115, 1er.Piso, (1900) La Plata, Argentina

Tel/Fax: (+54 221) 422 8252

e-mail: [giandini, cpons, gbaum]@sol.info.unlp.edu.ar

Abstract

The Unified Modeling Language (UML) is a semi-formal graphical language that has been accepted as standard to model object-oriented software systems. This language defines various kinds of diagrams that are used to describe different aspects or views of a system. In particular, Use Cases diagrams are used to capture the requirements of the systems and to guide their development process. The different Use Cases defined throughout a development process are not independent but it is possible to set relations between them. The main relations considered by UML are the following: *Generalization, Include and Extend*. These relations as well as the remaining UML constructs are semi-formally defined, giving place to ambiguous interpretations and inconsistencies.

This paper presents a formalization that gives precision to the definition of the main relations between Use Cases. This formalization will allow us to check consistency when incrementing the Use Cases model during the software development process and when relating Use Cases model with others models.

Keywords: Software Engineering, Object-Oriented Analysis and Design, Graphical Modeling Languages, Use Cases, Formal Semantics

1 Introduction

The Unified Modeling Language (UML) [17] is a semi-formal graphical language that has been accepted as standard to model object-oriented software systems. This language defines various kinds of diagrams which are used to describe different aspects or views of a system. In the UML, one of the key tools for behavior modeling is the Use Cases construct, originated from OOSE [7] and *developed* in works as [8] between others. A Use Case specifies one way of using a system without revealing the internal structure of the system. Use cases have been adopted almost universally to capturing requirements, but they are much more than a tool for specifying the requirements of a system. They also drive its design, implementation and test; that is they drive the whole development process.

Generally, software development process -for instance the Unified Process [9]- 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 workflows. The result of each iteration represents an increment over each model building in the previous workflows. The different Use Cases defined throughout a development process are not independent but it is possible to set relations between them. The main relations considered by UML are the following:

- *Generalization*: is a relation that reflects the amplification of the use case functionality or refines its original functionality through the addition of new operations and/or attributes and/or behavior sequences.
- *Include* : is a relation that allows us to reuse an encapsulated use case in different context through its invocation from others use cases.
- *Extend* : is a relation that reflects the amplification of the use case functionality through the extension of its behavior sequences.

These relations as well as the remaining UML constructs are defined in the UML semantics document [17]. This description is semi-formal, i.e. parts of it are specified with well-defined languages, while other parts have been described informally in English. The abstract syntax of the different language constructs in UML is specified with the graphical notation of diagrams in UML itself, while the well-formedness rules of UML are given in OCL [12], an object-oriented constraint language. Ordinary English is choseen for describing the semantics of UML. This make the structure of the language rigorous whereas the semantics of the language is still quite informal.

A language needs a formal specification to be unambiguous, understandable and properly used. Furthermore, the semantics of the language must be precise if tools are to perform intelligent operations on models expressed in the language, like consistency checks and transformations from one model to another.

There is an important number of theoretical works giving a precise description of core concepts of the graphical modeling notation UML and providing rules for analyzing their properties; see, for instance [1, 3, 4, 5, 10, 11, 16]. Specifically, in the area of UML behavioral diagrams, Övergaard in [13] and [14] formalizes Use Cases and Collaborations, respectively; however, in all these cases, the definitions include semantics elements of the system, such as objects of the system. On the other hand, the work of Back [2] analyzes and formalizes the Use Cases diagrams through contracts defining previously the classes that model the system and its behavior.

In this paper, we present rigorous definitions and well-formedness rules in order to determine without ambiguity in which cases the operations between use cases are well-defined and how is the

result of applying them. These definitions conform an algebra for Use Cases. We want to pointout that these definitions are only based on syntactics elements of the system, i.e. they do not include different development levels (e.g. specification and execution) setting an advantage respect to the refered works. Moreover, we consider the current versions of the UML to describe relationship between use cases. Finally, we develop a semantics comparison between the include and extend relationships.

2 Use Case

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. Since Use Cases do not deal with technicalities inside the system but focus on how the system is perceived from the outside, they are most useful in discussions with end-users to make sure that there is agreement on the requirements on the system, on its delimitation etc. More specifically, a Use Case specifies a set of complete sequences of actions which the system can perform. Each sequence is initiated by a user of the system, and it includes the interaction between the system and its environment as well as the system's response to these interactions.

2.1 Example

We will present the model of a system to maintain a Library. The members of the library share a collection of books. The system should allow them to borrow books, to return them or to renew a loan. When returning or when renewing the loan of a book, the member should pay a fee. In the event this fee is not paid, the member won't be able to borrow a new book or to renew a loan. The figure 1 shows the use case RenewLoan. This use case specifies the functionality of the system, for the renewing of a loan.

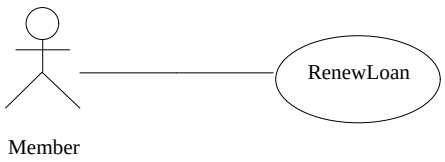

Figure 1. RenewLoan Use Case

Use cases can be specified in a number of ways. Generally natural language structured as a conversation between user and system is used, see [8]. The conversation shows the request of a user and the corresponding answers of the system, at a high level of abstraction. Figure 2 shows a conversation between an actor (a member of the library) and the system. The conversation considers the normal action sequence and also alternative sequences (e.g. the case in that the book is not available).

In the UML a Use Case is a kind of Classifier having a collection of operations (with its corresponding methods). Operations describe messages that instances of the use case can receive. Methods describe the implementation of operations in terms of action sequences that are executed by the instances of the use case. In general instead of having a set of operations, a use case has only a single operation, for example the RenewLoan use case has a single operation named "ask for renew loan". The method that implements that operation contains the set of action sequences, some of the sequences in this set correspond to normal execution paths, while others correspond to alternative cases.

Let uc be the use case defined above. The definition of uc, using the standard notation and metamodel of UML ([17], pag. 2-14, 2-114) is as follows:

uc.name= RenewLoan

uc.operation[1] = {op1}

op1.name=ask for renew loan

op1.method.body[2]= {< validate member identification, validate book availability, ask for debt, renew loan>,

< validate member identification, reject loan>, < validate member identification, validate book availability, reject loan>,< validate member identification, validate book availability, ask for debt, pay fee, renew loan >}

User Actions	**System Answers**
Actor: Member	
1. ask for renew loan	2. validate member identification
	3. validate book availability
	4. ask for debt
	5 renew loan

Alternatives:

1. member identification is not valid -> reject loan
2. book is not available -> reject loan
3. member has debt -> payFee, then renew loan

Figure 2. *Use Case RenewLoan Conversation*

3 The *Include* relation between Use Cases

An include relationship between two use cases means that the behavior defined in the target use case is included at one location in the sequence of behavior performed by an instance of the base use case. When a use case instance "reaches the location" where the behavior of another use case is to be included, it performs all the behavior described by the included use case and then continues according to its original use case. The included use case may not be dependent on the base use case. In that sense the included use case represents encapsulated behavior which may easily be reused in several use cases.

3.1 Example

In the example above, in order to enhance the use case model, we can add a new use case named PayFee that performs the pay of the debt. The figure 3 shows the use case RenewLoan related to the use case PayFee by the *include* relation.

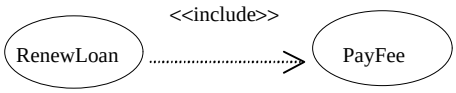

Figure 3. The Include relation between Use Cases

The use case PayFee has a single operation named *payFee*, with a single action sequence:

payFee.actionSequence= {< input payment, validate payment, modify debt>}

The *include* relation between the use cases RenewLoan and PayFee can be described as follow:

[1] For a *Classifier C, C.operation* denote the set of operations of *C*. Each operation *op* consists of a name and a set of Action sequences, denoted *op.name* y *op.method.body*, respectively. In general we abbreviate op.method.body by op.actionSequence.

[2] 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, we interpret a procedure expression as a set of action sequences.

Let PayRenewLoan be the use case obtained from RenewLoan by the inclusion of the use case PayFee. The textual representation of PayRenewLoan is:

PayRenewLoan.operation = {op1´}, op1´.name= ask for renew loan,

op1´.actionSequence= {<validate member identification, validate book availability, ask for debt, renew loan>,< validate member identification, reject loan>,< validate member identification, validate book availability, reject loan>,< validate member identification, validate book availability, ask for debt, input payment, validate payment, modify debt, renew loan >}

3.2 Formalizing the *Include* relation between Use Cases

We present a formalization for the inclusion between use cases, defined in UML by the *include* relationship.

Definition 1: Use Case Inclusion

A use Case UC is the result of including UC2 in UC1; i.e. UC = UC1 $\oplus_{inc}$ UC2 if the following holds:

a- Applicability: in UC1 there must be an invocation to UC2

$\exists o \in$ UC1.operation **.** $\exists ut \in$ o.actionSequence **.** UC2.name$\in$ ut

b- Completeness: UC contains all the possible ways to include UC2 in UC1

$\forall o1 \in$ UC1.operation **.** $\exists o \in$ UC.operation **. (**o.name=o1.name $\wedge$ ($\forall s1 \in$ o1.actionSequence**.** inclusions(s1,UC2) $\subseteq$ o.actionSequence))

c- Correctness: every action sequence in UC comes from UC1, possibly extended by UC2. $\forall o \in$ UC.operation **.** $\exists o1 \in$ UC1.operation **. (**o.name=o1.name $\wedge$ ($\forall s \in$ o.actionSequence**.** $\exists s1 \in$ o1.actionSequence**.** s$\in$ inclusions(s1,UC2)))

Definition 2:

isIncluible: ActionSequence x UseCase

The predicate is true if the action sequence contains some invocation to the use case.

$\forall$s:ActionSequence $\forall$uc:UseCase . isIncluible(s,uc) $\leftrightarrow$ uc.name $\in$ s

Definition 3:

inclusions: ActionSequence x UseCase -> Set(ActionSequence)

The function inclusions(s, uc) returns the set of all possible inclusions of the use case uc inside the sequence s. The function is defined by cases.

Case 1: $\neg$ isIncluible(s, uc)

inclusions(s, uc)={s}

Case 2: isIncluible(s, uc)

inclusions(s, uc)={ before(s,a);s2;after(s,a) / a=uc.name $\wedge$ s2$\in$ uc.actionSequence }

3.3 Applying the formalization to the example

We present part of the instantiation of the formulas that define the inclusion between use cases on the example in 3.1, using the textual representation of the use case in 2.1. The goal is to show that the model in the example fulfils the formulas, that is to say, it is a complete and correct Use Case inclusion.

According to Definition 1, we must prove that: PayRenewLoan = RenewLoan $\oplus_{inc}$ PayFee, i.e. the following points must be satisfied:

a) Applicability: the use case RenewLoan has an invocation to the use case PayFee, i.e.:

$\exists o \in$ RenewLoan.operation **.** $\exists ut \in$ o.actionSequence **.** PayFee.name$\in$ut, Since the only operation op1 in RenewLoan has an action sequence: < validate member identification, validate book availability, ask for debt, pay fee, renew loan > that includes the action *payFee* and considering that PayFee is a use case with a single operation named *payFee*, the applicability condition is proved.

b- Completeness: PayRenewLoan contains all possible ways to include PayFee in RenewLoan

$\forall o1 \in$ RenewLoan.operation**.** $\exists o \in$ PayRenewLoan.operation **.** (o.name=o1.name $\wedge$ ($\forall s1 \in$ o1.actionSequence**.** inclusions(s1, PayFee) $\subseteq$ o.actionSequence)) As RenewLoan and PayRenewLoan have a single operation each one -op1 and op1´ respectively- with the same name, only remains to verify the inclusions, i.e.:

($\forall s1 \in$ op1.actionSequence**.** inclusions(s1, PayFee) $\subseteq$ op1´.actionSequence).

In the example the action sequences are the following ones:

op1.actionSequence={ut1, ut2, ut3, ut4}

op1'.actionSequence={ut1, ut2, ut3, ut4'}

By Definition 2, only the sequence ut4 of op1 -see point a)- satisfies the predicate isIncluible. By Definition 3:

inclusions(ut4, PayFee) = {<validate member identification, validate book availability, ask for debt, input payment, validate payment, modify debt, renew loan >}. Then inclusions(ut4, PayFee) $\subseteq$ op1´.actionSequence. The remaining of the sequences exist in op1´in their original form, then the inclusion is satisfied.

In similar way, the Correctness formula can be instantiated, showing that is fulfilled. The instantiations of the next definitions are not included due to space limitations.

4 The *Extend* relation between Use Cases

An extend relationship defines that a use case may be extended with some additional behavior defined in another use case. This relation contains a condition and references a sequence of extension points in the target use case. Once an instance of a use case is to perform some behavior referenced by the first extension point in an extend relationship, the condition es evaluated. If it is fulfilled, the sequence of the use case instance is extending to include the sequence of the extending use case. The different parts of the extending use case are inserted at the locations defined by the sequence of extension points in the relationship – one part at each referenced extension point.

4.1 Example

The use case in figure 1 can be extended in order to count how many book renews are rejected because the member identification is not valid. This extension can be achieved without modifying the original use case, by means of an extend relationship and a new use case specifying the increment of behavior. Figure 4 shows this relationship between use cases. In this case the extension point specified by the extend relationships is the action of to reject loan. The condition of the extension is that the member identification is not valid.

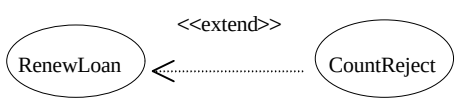

Figure 4. The extend relation between Use Cases

Let CountReject be the use case specifying the increment of behavior. CountReject has a single operation with a single action sequence, as follow:

countReject.actionSequence= {<updateRejectCounter>}

The *extend* relation ext is defined as follows:

ext.base= RenewLoan, ext.extension= CountReject

ext.condition= *the member identification is not valid*

ext.extensionPoint= { reject loan }

Let CountRejectRenewLoan be the use case obtained from RenewLoan by the application of the extension specified by ext, i.e. CountRejectRenewLoan = RenewLoan $\oplus_{ext}$ CountReject. The textual representation of CountRejectRenewLoan is as follows:

countRejectRenewLoan.actionSequence= {<validate member identification, validate book availability, ask for debt, renew loan >, < validate member identification, reject loan, updateRejectCounter >,< validate member identification, validate book availability, reject loan>, < validate member identification, validate book availability, ask for debt, pay fee, renew loan >}

4.2 Formalizing the *Extend* relation between Use Cases

As in the case of the inclusion, we present a formalization for Use Case Extension, defining the features of the resulting use case. In the UML the extension relation is a special modeling element, so the relation appears explicitily in the following definition:

Definition 4: Use Case Extension

A use Case UC is the extension of UC1 by UC2 through an "extend" relationship *ext*;

i.e. UC = UC1 $\oplus_{ext}$ UC2 if the following holds:

a- Applicability: UC1 is extensible by *ext* if the following condition holds:

For each extension point of *ext,* there should exist a corresponding action inside the sequences of actions of the use case:

$\forall i \in$ ext. extensionPoint**.** $\exists uo \in$ UC1.operation **.** $\exists uot \in$ uo.actionSequence **.** i.location $\in$ uot

b- UC1-Completeness: every action sequences in UC1 is extended in every possible way:

$\forall o1 \in$ UC1.operation . $\exists o \in$ UC.operation .

(o.name=o1.name $\wedge$ ($\forall s1 \in$ o1.actionSequence. extensions(s1,ext,UC2) $\subseteq$ o.actionSequence))

c- UC1-Correctness: every action sequence in UC is an extension of some action sequence in UC1.

$\forall o \in$ UC.operation .$\exists o1 \in$ UC1.operation .

(o.name=o1.name $\wedge$ ($\forall s \in$ o.actionSequence. $\exists s1 \in$ o1.actionSequence. $s \in$ extensions(s1,ext,UC2)))

Definition 5:

isExtensible: ActionSequence x Extend

The predicate is true if the action sequence contains some extension point defined by the extend relation.

$\forall$s:ActionSequence $\forall$ext:Extend .

isExtensible(s,ext) $\leftrightarrow \exists$ i$\in$ *ext*.extensionPoint . i.location$\in$ s

Definition 6:

extensions: ActionSequence x Extend x UseCase -> Set(ActionSequence)

The function extensions(s,ext,uc) returns the set of all possible extensions of the sequence s given by the Extend relation ext and the Use Case uc. The function is defined by cases.

Case 1: $\neg$ isExtensible(s,ext)

extensions(s,ext,uc)={s}

Case 2: isExtensible(s,*ext*)

extensions(s,ext,uc)={before(s,i.location); s2; after(s, i.location) /

i$\in$ *ext*.extensionPoint $\wedge$ i.location$\in$ s $\wedge$ s2$\in$ uc.actionSequence }

Definition 7: UC extends UC1 if there exists a use case UC2 such that UC is the extension of UC1 by UC2 through an *ext* relation: UC $extends_{ext}$ UC1 $\leftrightarrow \exists$ UC2:UseCase **.** (UC = UC1 $\oplus_{ext}$ UC2)

5 The *Generalization* relation between Use Cases

A generalization relationship between use cases implies that the child use case inherits all the attributes, sequences of behavior, extension points and relationship of the parent use case. The child use case may also define new operations, as well as to redefine or to enrich existing operations in the parent use case, with new behavior sequences. To distinguish the kind of specialization of an operation, we suggest the inclusion of an stereotype -UML element-: <<redefine>> for the first case and <<enrichment>>for the second.

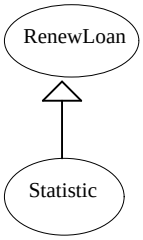

Figure5. Use Cases Generalization

5.1 Example

The use case in figure 1 and fugure 2 can be specialized in order to count how many people have renewed the loan of a technical book. This specialization can be achieved without modifying the original use case, by means of a generalization relationship and a new use case specifying the increment of behavior. Figure 5 shows this relationship between the use cases.

Let Statistic be the use case specifying the increment of behavior. Statistic has a single operation with a single action sequence that enriches the only one operation of RenewLoan, as follows:

Statistic.operation= {op1} op1.name=renewLoan op1.stereotype=<<*enrichment*>>

op1.actionSequence= {<validate member identification, validate book availability, ask for debt, renew loan, updateRenewsTechnicalCounter >, < validate member identification, validate book availability, ask for debt, pay fee, renew loan, updateRenewsTechnicalCounter >}

Let StatisticRenewLoan be the use case obtained from RenewLoan by the application of the *generalization* relation. The textual representation of StatisticRenewLoan is:

StatisticRenewLoan.operation={op},op.name=renewLoan

op.actionSequence= {<validate member identification, validate book availability, ask for debt, renew loan >,< validate member identification, reject loan >, < validate member identification, validate book availability, reject loan>, < validate member identification, validate book availability, ask for debt, pay fee, renew loan >, <validate member identification, validate book availability, ask for debt, renew loan, updateRenewsTechnicalCounter >, < validate member identification, validate book availability, ask for debt, pay fee, renew loan, updateRenewsTechnicalCounter >}

5.2 Formalizing the *Generalization* relation between Use Cases

In this section, we formalize the generalization between use cases, defining the features of the resulting use case. Since use cases are generalizable elements of UML, by the implicit mechanism of inheritance, the resulting use case will contain all attributes and asociations -e.g. with Actors outside of the system- of both the parent use case and the child, considering the redefinitions specified in the later. In the UML, regarding the Generalization relationship, the behavior of redefined or enriched operations is not clearly defined. Our formalization focus on improving the precision on this specification.

Definition 8: Use Case Generalization

A use Case UC is the generalization of UC1 by UC2; i.e. UC = UC1 $\oplus_{gen}$ UC2 if the following holds:

a- Applicability: every operation in UC2 that is not included in UC1 must have different name from any action in UC1. If this does not happen, actions can be confused with operations in the resulting use case.

$\forall o2 \in$ UC2.operation **.** (o2 $\notin$ UC1.operation $\rightarrow \forall o1 \in$ UC1.operation **.**

$\forall s1 \in$ o1.actionSequence **.** $\forall a \in$ s1**.** o2.name<>a)

b- Completeness:

b.1- every operation in UC1 that is not included in UC2 and every operation in UC2 that is not included in UC1, are in UC in its original form:

$\forall o1 \in$ UC1.operation **.** (($\forall o2 \in$ UC2.operation **.** o2.name<>o1.name)$\rightarrow$ o1$\in$ UC.operation)

$\forall o2 \in$ UC2.operation **.** (($\forall o1 \in$ UC1.operation **.** o1.name<>o2.name)$\rightarrow$ o2$\in$ UC.operation)

b.2- every operation in both UC1 and UC2, is included in UC redefined or enriched, according to its stereotype.

$\forall o1 \in$ UC1.operation **.**

($\exists o2 \in$ UC2.operation **.** o2.name=o1.name)$\rightarrow$

((o2.stereotype = <<*redefine*>> $\rightarrow$ o2 $\in$ UC.operation) $\wedge$

(o2.stereotype = <<*enrichment*>> $\rightarrow \exists o \in$ UC.operation **.** (o.name =o2.name $\wedge$

o.actionSequence= o1.actionSequence $\cup$ o2.actionSequence)))

b.3- *Inheritance of extension relations*: for each extension relation with base in either UC1 or UC2, there will be one extension relation from UC with both the same name and the same extension points.

$\forall$ext: Extend **.** (ext.base = UC1 $\vee$ ext.base = UC2 $\rightarrow \exists$ext'**.** (ext'.base=UC $\wedge$ ext'.name=ext.name $\wedge$ ext'.extensionPoint=ext.extensionPoint))

c -Correctness: every operation in UC, is included either in UC1 or in UC2. If the operation is defined in both, it will be redefined or enriched, according to its stereotype:

$\forall o \in UC.operation$ **.**

$((o \in UC1.operation \wedge \forall o1 \in UC2.operation$ **.** $o1.name <> o.name) \vee$

$(o \in UC2.operation \wedge \forall o1 \in UC1.operation$ **.** $o1.name <> o.name) \vee$

$(o \in UC2.operation \wedge o2.stereotype = <<redefine>>) \vee$

$(\exists o1 \in UC1.operation$ **.** $\exists o2 \in UC2.operation$ **.** $(o1.name = o2.name \wedge o2.stereotype = <<enrichment>>$

$\wedge o.name = o2.name \wedge o.actionSequence = o1.actionSequence \cup o2.actionSequence)))$

Definition 9: UC is a generalization of UC1 through the increment specified by UC2 if UC is obtained composing UC1 with UC2 by a *generalización* relation, i. e.:

UC $generalice_{UC2}$ UC1 $\leftrightarrow$ (UC = UC1 $\oplus_{gen}$ UC2)

6 Comparison between the *Include* and *Extend* relations

The UML defines the *extend* and *include* relationships for use cases, indicating differences between them. The former includes a construct to define the relation, that contains extension points -coinciding with actions of the extending use case- and a condition to be evaluated to reach the first extension point. On the other hand, the *include* relationship is defined as the inclusion of a use case in another, in a "modularization of procedures" style, since the "location" where the addition is performed must coincide with the name of the included use case. This mechanism is similsr to "invocation" between procedures.

After developing examples and defining a formalization of both relations, we can observe that there is not semantics differences between them. More precisely, we can conclude that the include relation can be considered a particular case of the extend relation, where:

- The extension point sequence is a unary sequence where the single point coincides with the invocation of the included use case.
- The condition that must be satisfies at this point, is always true.

6.1 Example

The inclusion relation presented in section 3, between use cases RenewLoan and PayFee can be expressed through an *extend* relation, in the following way:

Let ext´ be the *extend* relation that is defined as follows:

ext´.base= RenewLoan ext´.extension= PayFee ext´.condition= *true* ext´.extensionPoint= {pay fee}

Let PayRenewLoan´ be the use case obtained from RenewLoan by the application of the extension specified by ext´, i.e. PayRenewLoan´ = RenewLoan $\oplus_{ext'}$ PayFee. The textual representation of PayRenewLoan´ is as follows:

PayRenewLoan´.actionSequence= {< validate member identification, validate book availability, ask for debt, renew loan>, < validate member identification, reject loan>, < validate member identification, validate book availability, reject loan>, < validate member identification, validate book availability, ask for debt, input payment, validate payment, modify debt, renew loan >}

We can observe that the resulting use cases PayRenewLoan -defined in section 3 through an *include* relation- and PayRenewLoan´ -defined in this section through an *extend* relation-, are equal.

7 Conclusions and Future Works

The Unified Modeling Language (UML) is a semi-formal graphical language that has been accepted as standard to model object-oriented software systems. This language defines various kinds of diagrams that are used to describe different aspects or views of a system. In particular, Use Cases diagrams are used to capture the requirements of the systems and to guide their development process. The different Use Cases defined throughout a development process are not independent but it is possible to set relations between them. The main relations considered by UML are the following: *Generalization, Include and Extend*. These relations as well as the remaining UML constructs are semi-formally defined, giving place to ambiguous interpretations and inconsistancies.

In this paper, we present rigorous definitions and well-formedness rules in order to determine without ambiguity in which cases the operations between use cases are well-defined and how is the result of applying them. These definitions conform an algebra for Use Cases. This approach will allow us to carry out consistency checks when incrementing the Use Cases model through iterations during the software devepolment process and when relating this model with others. Regarding this topics, in [15] we distinguish three different kinds of dependency relations between UML models and propose a formal description of them.

Also, we develop a comparison between the *extend* and *include* relations, arriving to the conclusion that they not present important differences.

On top of this formalization we can analyze more complex forms of evolution obtained from the combination of primitive operations. These evolutions may arise conflictive situations that can be formally detected using this formalization. In this direction, in [6] we analyze conflicts in reusable components evolution, that we think can be applied in Use Cases evolution.

The relations considered in this work only add behavior to Use Cases, extending their action sequences. As continuation, it is interesting to analyze other ways of Use Cases evolution - add new actors, new roles for objects, etc. On the other hand, the UML Specification Document says that a complex use case can be refined into a set of smaller use cases. Then, the funtionality specified by the complex use case must be completely traceable to its subordinate use cases. This is other way of evolution to consider in future works.

References

1. Araújo, J, Formalizing Sequence Diagrams, In Luís Andrade, Ana Moreira, Akash Deshpande and Stuart Kent, editors, Proc. OOPSLA´98 Wsh. Formalizing UML. Why? How?, Vancouver, (1998).
2. Back, R. Petre L. and I. Porres Paltor. Analysing UML Use Cases as Contract. Proceedings of the UML'99 Second International Conference. Fort Collins, CO, USA, October 28-30/99. Lecture Notes in Computer Science, Springer-Verlag, 1999.
3. 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, Springer, (1997).
4. Evans,A., France,R., Lano,K. and Rumpe,B., Towards a core metamodelling semantics of UML, Behavioral specifications of businesses and systems, H,Kilov editor, , Kluwer Academic Publishers, (1999).

5. Evans,A., France,R., Lano,K. and Rumpe, B., Developing the UML as a formal modeling notation, UML'98 Beyond the notation, Muller and Bezivin editors, Lecture Notes in Computer Science 1618, Springer-Verlag, (1998).
6. Giandini R. Documentación y evolución de componentes reusables. Tesis del Magister en IS, Universidad Nacional de La Plata, Argentina, http://www-lifia.info.unlp.edu.ar/~giandini. Setiembre 1999.
7. Jacobson I.. Object-Oriented Development in an Industrial Environment. In Proceedings OOPSLA' 87, special issue of SIGPLAN Notices. Vol 22, N°12, pp.183-191, 1987.
8. Jacobson, I. Christerson, M. Jonsson P. and G. Övergaard. Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, 1993.
9. Jacobson, I Booch, G. Rumbaugh J.. The Unified Software Development Process, Addison Wesley. ISBN 0-201-57169-2, 1999
10. Kim, S. and Carrington,D., Formalizing the UML Class Diagrams using Object-Z, proceedings UML´99 Conference, Lecture Notes in Computer Sciencie 1723, (1999).
11. Knapp, Alexander, A formal semantics for UML interactions, <<UML>>´99 - The Unified Modeling Language. Beyond the Standard. R.France and B.Rumpe editors, Proceedings of the UML´99 conference, Colorado, USA,. Lecture Notes in Computer Science 1723, Springer. (1999).
12. Object Constraint Language (OCL). version 1.3, July 1999. Part of [17]
13. Övergaard G. and K. Palmkvist. A Formal Approach to Use Cases and Their Relationships. In P. Muller and J. Bézivin editors, Proceedings of the UML'98: Beyond the Notation, Lecture Notes in Computer Cience 1618. Springer-Verlag, 1999.
14. Övergaard. G. A Formal Approach to Collaborations in the Unified Modeling Language. Proceedings of the UML'99 Second International Conference. Fort Collins, CO, USA, October 28-30/99.Lecture Notes in Computer Science, Springer-Verlag, 1999.
15. Pons, C., Giandini, R. and Baum, G, Dependency relations between models in the Unified Process. In: *Proceedings of the IWSSD*. San Diego, California, IEEE Press, 5-7 Nov. 2000.
16. Pons,C., Baum,G., Felder,M., Foundations of Object-oriented modeling notations in a dynamic logic framework, Fundamentals of Information Systems, Chapter 1, T.Polle,T.Ripke,K.Schewe Editors, Kluwer Academic Publisher, (1999).
17. Unified Modeling Language (UML) Specification - Version 1.3, July 1999. UML specification revised for OMG, http://www.rational.com