Dependency Relations Between Models in the Unified Process

Claudia Pons Roxana Giandini Gabriel Baum

Lifia, Universidad Nacional de La Plata
Calle 50 esq.115, 1er.Piso. (1900) La Plata, Argentina
E-mail: cpons@sol.info.unlp.edu.ar

Abstract

The goal of the Unified Process is to guide developers in efficiently implementing and deploying systems that meet customer needs. During the Unified Process, a variety of models of the system is developed. All these models are not independent, but they are related to each other. Elements in one model have trace dependencies to other models; they are semantically overlapping and together represent the system as a whole.

It is necessary to have a precise definition of the syntax and semantics of the different models and their relationships, since the lack of accuracy in their definition can lead to wrong model interpretations and inconsistency between models.

In this paper we distinguish three different kinds of dependency relations between models and propose a formal description of them. The goal of the proposed formalization is to provide formal foundations for tools that perform intelligent analysis on models expressed in UML assisting software engineers through the development process.

1. Introduction

The Unified Process [8] is a software development process, that is to say it is a set of activities needed to transform user's requirements into a software system. The Unified Process uses the Unified Modeling Language [16] when preparing all blueprints of the software system. The main characteristics of the Unified Process are:

• *The Unified Process is Use-Case driven.* Customer needs are not easy to discern. This demands the existence of a mechanism for capturing the user's needs so that they can be clearly communicated to all the members of the project team. Use cases [7] 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. Based on the use-case model developers build a series of analysis, design and implementation models that realize the use cases.

• *The Unified Process is iterative and incremental,* it repeats 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, building a number of different models.

All these models are not independent. They are related to each other, they are semantically overlapping and together represent the system as a whole. Elements in one model have trace dependencies to other models. For instance, a use case (in the use-case model) can be traced to a collaboration (in the design model) representing its realization. Figure 1 illustrates these relations between models.

But this is not the only relation existing between models in the Unified Process; due to the incremental nature of the process, each iteration results in an increment of previous models. An increment is not necessarily additive. Generally in the early phases of the life cycle, a superficial model is replaced with a more detailed or sophisticated one, but in later phases increments are typically additive, i.e. a model is enriched with new features, while previous features are preserved. As a consequence the models defined in each iteration are a refinement (or an increment or an extension) of models in the former iteration.

Figure 2a lists the workflows – requirements, analysis, design, implementation and test – in the left-hand column. The curves approximate the extent to which the workflows are carried out in each iteration, through the development process. Then figure 2b shows the two different kinds of relations:

- horizontal relations between models belonging to the same workflow in different iterations

0-7695-0884-7/00 $10.00 © 2000 IEEE

- vertical relations between models belonging to the same iteration in different workflows

Finally, there is a third dimension: *the artifact dimension.* Each model is made up from several artifacts (i.e. diagrams). For instance, an analysis model consists of the following artifacts: analysis class diagram, interaction

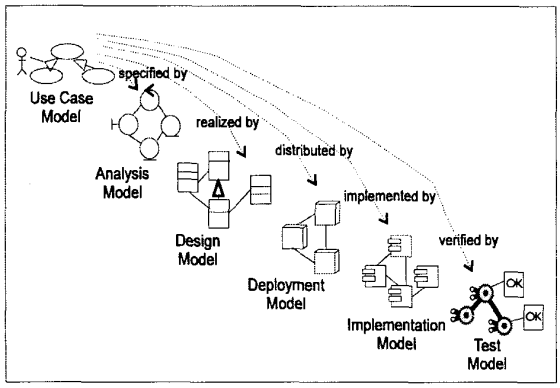

Figure 1: dependencies between models through the Process

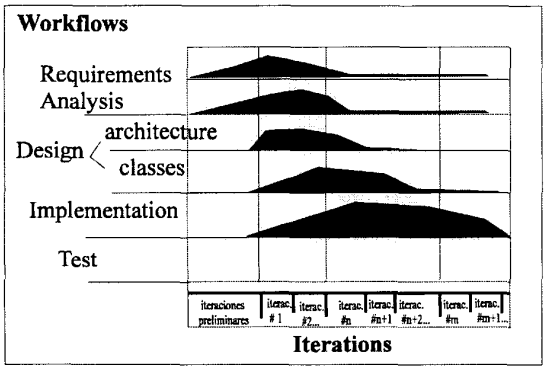

Figure 2a: two dimensions in the Unified Process: extent of workflows through iterations

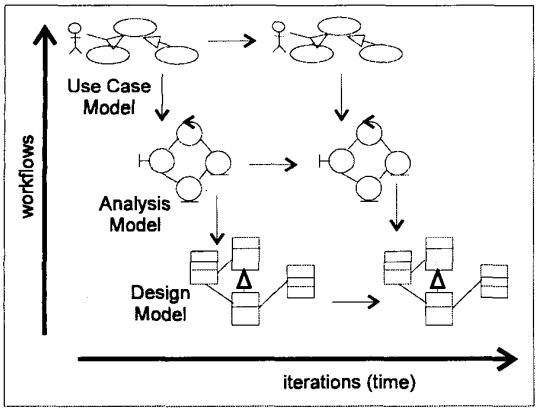

Figure 2b: two dimensions in the Process: different kinds of relation between models.

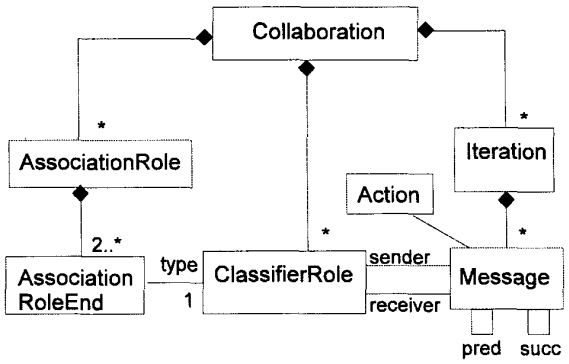

Figure 3: artifacts composing a Collaboration model

diagrams, collaboration diagrams (see Figure 3). All the artifacts within a single model are related and have to be compatible with each other.

Relations between models should be formally defined since the lack of accuracy in their definition can cause problems, for example:

- Wrong model interpretations: the interpretation done by the user that reads the model may not coincide with the interpretation of the model creator.
- Inconsistency among the different models: if the relation existing among the different sub-models is not accurately specified, it is not possible to analyze whether its integration is consistent or not.
- Discussion regarding the model meaning: the people involved in the project often waste time discussing the different possible interpretations that can be allocated to the models.
- Evolution conflicts: when a model is modified, unexpected behavior may occur in other models that depend on it..

The specification of UML constructs and their relationships [16] 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 are 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 the works of Back et al.[2], Araújo [1], Breu et al.[3], Evans et al.[5] [6], Kim and Carrington [9], Knapp [10], Övergaard [11] [12] [13], Pons et al.[14], Cibrán et al.[4], Reggio et al.[17], Smith et al.[18]. Some of these works deal with the relationships among models. In particular, the present work is closely related to the works of Övergaard [11], [12]. Its originality resides in that it distinguishes different classes of relationships and it also analyzes the relationships among these relationships. On the other hand the whole analysis is

carried out in the syntactic level (contrarily to the works of Övergaard that combine syntactic and semantic aspects in order to describe relationships between models). Working in a purely syntactic level simplifies the definitions of properties as well as their validation.

2. First-dimension relation (Workflows)

In this section we analyze the vertical relations between models, that is to say relations belonging to the same iteration in different workflows. Due to space limitations we only describe the relationships between the requirement phase and the analysis phase.

2.1. Creating analysis models from use cases

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

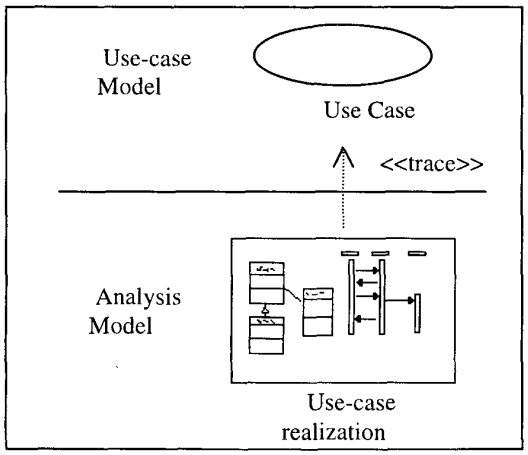

Figure 4: relation between a use case realization in the analysis model and a use case in the use-case model.

4 shows the relation between a use case and its realization.

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 renovate a loan. When returning or when renovating 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 renovate a loan. The figure 5 shows the use case RenewLoan. This use case specifies the functionality of the system, for the renew of a loan.

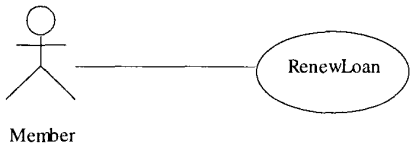

Figure 5: renewLoan use case

Use cases can be specified in a number of ways. Generally natural language structured as a conversation between user and system is used, see [7]. The conversation shows the request of a user and the corresponding answers of the system, at a high level of abstraction. Figure 6 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 UseCase 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.

User Actions	**System Answers**
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 -> ask for payment, then renew loan

Figura 6: Use Case Conversation

Let uc be the use case defined above. The definition of uc (using the standard notation and metamodel of UML (in [16] page 2-114) is as follows:

uc.operations = <op1>

op1.name=ask for renew loan

op1.method.body=

{< 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, ask for payment, renew loan >}

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

The realization of the use case:

Figure 7 shows a set of Classifier Roles and their connections, while figure 8 shows one of the iteration diagram specifying the message flows between objects playing the roles in the collaboration. Figure 9 contains the textual representation of the diagrams. These diagrams realize the use case above.

2.2. Formalizing the realization relation between Use Cases and Collaborations

Lets define a set of concepts that are necessary in order to formalize the relations between use cases and collaborations.

Def. 1: a sequence is a totally ordered set of elements. Let p and q be sequences, $p \leq q$ implies that p is a prefix of q, i.e. $q = pr$ for some sequence r.

Def. 2: let (MS, ≤) be the poset of messages in an interaction (messages are partially ordered by the predecessor/successor relation). The set of linearizations on MS is defined as the set of sequences of messages in

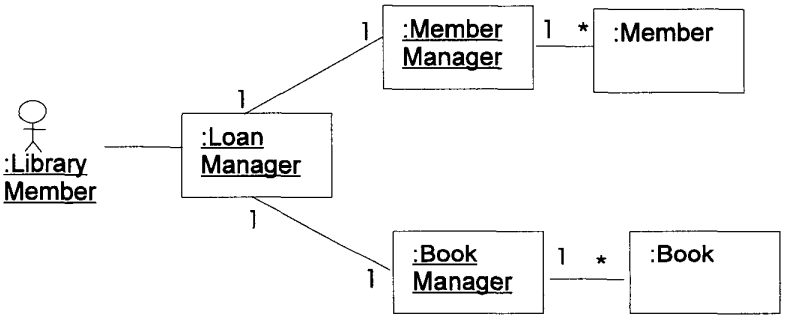

Figure 7. Realization of the use case: collaborating ClassifierRoles

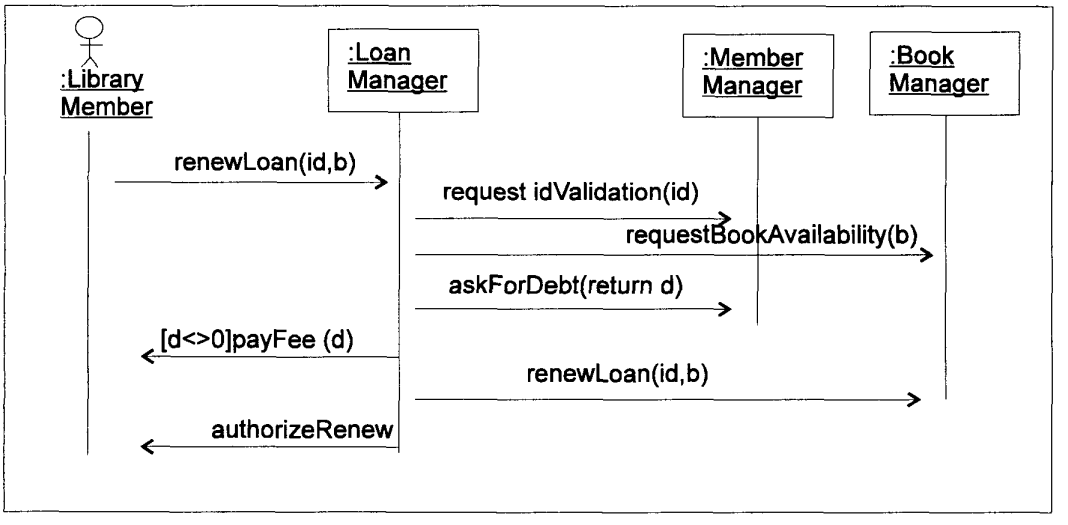

Figure 8: part of the Realization of the use case: one Interaction diagram

Let RenewLoan be the Use Case defined above, and let CRenewLoan be the collaboration. The definition of CRenewLoan (using the standard notation and metamodel of UML (UML 99, page 2-100) is as follows:

CRenewLoan.representedClassifier= RenewLoan
CRenewLoan.interaction={int1,int2, int3,int4}
CRenewLoan.classifierRole={R1,R2,R3,R4,R5}
R1.name=LoanManager, R2.name=Book, R3.name=Member,
R4.name=MemberManager, R5.name=BookManager
CRenewLoan.associationRole={A1,A2,A3,...}
.....
int1.message={ (*Actor*, LoanManager, renewLoan(id,b)), (LoanManager, MemberManager, requestIdValidation), (LoanManager, BookManager, requestBookAvailability), (LoanManager, MemberManager, askForDebt) , (LoanManager, BookManager, renewLoan) }
int2.message={ (*Actor*, LoanManager, renewLoan(id,b)), (LoanManager, MemberManager, requestIdValidation), (LoanManager, BookManager, requestBookAvailability), (LoanManager, MemberManager, askForDebt), (LoanManager , Actor,payFee), (LoanManager, BookManager, renewLoan) }
.....
Where each message m is represented by a triple (m.sender,m.receiver,m.action), where m.sender denotes the role of the instance that invokes the communication, m.receiver is the role of the instance that receives the communication, m.action is the action which causes a stimulus to be sent according to the message.

Figure 9: textual representation of the CRenewLoan Collaboration

MS (i.e. the chains in the poset), and it is denoted as *lin*(MS, ≤).

Def. 3: maxLin(MS, ≤) is the set of maximal linearizations on MS. It is obtained from lin(MS, ≤) by dropping every sequence that is contained in another sequence in the set, for example:

lin(MS, ≤) ={<a,b,c,d>, <b,c>,<c,d>}
maxLin(MS, ≤) ={<a,b,c,d>}

Def. 4 : let S be a set of sequences of actions. external(S) denotes the sequences of S obtained omitting all the actions that are not visible externally.

Def. 5: a *conformance declaration* is a correspondence between action names in a use case and action names in a collaboration. Each name in the use case is mapped to (a name of) an action in the collaboration. This mapping provides more flexibility in the development process allowing analysts to modify the name of the actions as the process evolves.

For example, the following is a *conformance declaration* between the Use Case and the Collaboration above:

Actions in the use case **Actions in the collaboration**

ask for renew loan ----------> renewLoan(id,b)
validate member identification --->requestIdValidation(id)
validate book availability ----->requestBookAvailability(b)
ask for debt---------->askForDebt(id)
ask for payment---------- >payFee
renew loan ---------- > renewLoan
reject loan ---------- > reject

At this point we can define the *realization relation* between a Collaboration C and a Use Case UC. A Use Case is realized by a Collaboration if the Classifiers Roles in the Collaboration jointly cooperate to perform the behavior specified by the Use Case, but not more. In the case that the Collaboration includes more behavior than the one specified by the Use Case, the Use Case would be only a partial specification of the behavior described by the Collaboration. On the other hand, a use cases specifies actions that are visible from outside the system, but do not specify internal actions, such as creation and destruction of instances, communication between internal instances, etc.

Definition 6: A collaboration C is a realization of a Use Case UC according to the conformance declaration δ, denoted $C \geq_{\delta} UC$, if both of the following hold

a- $\forall uo \in UC.operation.\ \forall ut \in uo.actionSequence.$
$\exists int \in C.interaction.\ \exists ms \in lin(int.message).$
$\delta(uo.name) = act.operation.name$
$\wedge\ \delta^{+}(ut) = external((\ ms \rightarrow tail\ ).action)$

b- $\forall int \in C.interaction.\ \forall ms \in maxLin(int.message).$
$\exists uo \in UC.operation.\ \exists ut \in uo.actionSequence.$
$\delta(uo.name) = act.operation.name$
$\wedge\ \delta^{+}(ut) = external((\ ms \rightarrow tail\ ).action)$

Where: act = (ms->head).action

ms->head is the first element in the sequence ms

ms->tail is the subsequence obtained from ms by dropping the first element

ms.action is an abbreviation for ms->collect (e | e.action)

δ^+(ut) = ut->collect (a | δ(a))

Definition above states that every action sequence specified by the Use Case must have a corresponding action sequence in the Collaboration, that is equal to it (except for internal actions), and vice versa.

3. Second-dimension relations (Time)

In this section we analyze the horizontal relations between models, that is to say relations belonging to the same workflow in different iterations.

3.1. Evolving the use-case model

A use case model may be evolved in different ways. The UML considers at least two forms of evolution: the *extends* and the *generalization* relationships between use cases. In this paper we only take into consideration the former.

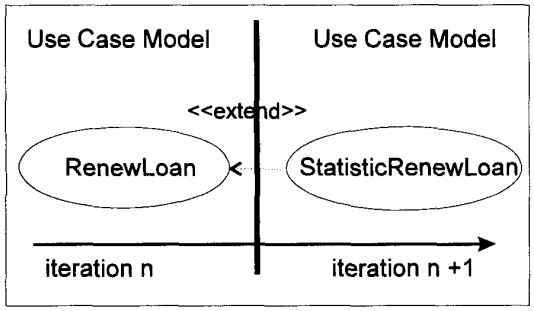

Figure 10: relation between a use case in the use case model and its extension

The extend relation represents the enrichment of a use case by the definition of additional actions (see figure 10). An extend relationship from use case A to use case B indicates that an instance of use case B may include (constrained by specific conditions specified in the extension) the behavior specified by A.

The definition of extend includes both a condition for the extension and a reference to an extension point in the target use case, that is, a position in the use case where additions may be made. Once an instance of a (target) use case reaches an extension point to which an extend relationship is referring, the condition of the relationship is evaluated. If the condition is fulfilled, the sequence obeyed by the use-case instance is extended to include the sequence of the extending use case.

Example:

The use case in figures 5 and 6 can be extended in order to count how many people have renovated the loan of a technical book. 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 10 shows this relationship between use cases. In this case the extension point specified by the extend relationships is the action of renewing a loan. The condition of the extension is that the book is a technical one.

Let Statistic be the use case above mentioned specifying the increment of behavior. Statistic has a single operation with a single action sequence, as follows:
Statistic.operation.actionSequence=
{<updateRenewsCounter>}

The extend relationship *ext* is as follows:
ext.base= RenewLoan
ext.extension= Statistic
ext.condition= *the book is technical*
ext.extensionPoint= { renew loan }

Let StatisticRenewLoan be the use case obtained from RenewLoan by the application of the extension above, i.e. StatisticRenewLoan = RenewLoan $\oplus_{ext}$ Statistic.

The textual representation of StatisticRenewLoan is as follows:
StatisticRenewLoan.operation.actionSequence=
{< validate member identification, validate book availability, ask for debt, renew loan, updateRenewsCounter> , < validate member identification, reject loan>,
< validate member identification, validate book availability, reject loan>,
< validate member identification, validate book availability, ask for debt, ask for payment, renew loan , updateRenewsCounter >}

Formalizing use case extensions:

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 Conditions: UC1 is extensible by *ext* if for each extension point of *ext*, there 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 7 (function definitions):

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

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 8: 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. (UC=UC1 $\oplus_{ext}$ UC2)

3.2. Evolving the collaboration model

The UML does not consider special dependency relationships between Collaboration. However since Collaborations realize Use Cases, it is important to reflect the relationships between Use Cases (e.g. extend relationships) on its realizing Collaborations. As well as Use Cases are extended by adding actions (defined in other Use Case), Collaborations can be extended with additional message sequences specified in another Collaboration.

For further deatails about the extension relationship between Collaborations based on the corresponding extension relationship between Use Cases, readers are referred to [15].

4. Third-dimension relations (Artifact)

Every model is made up from a number of related sub-models (or artifacts) that have to be semantically compatible obeying to several constraints between them.

The UML specification document [16] defines the abstract syntax of UML by class diagrams and well-formedness rules in OCL [16]. Most of the well-formedness rules in that document are examples of constraints on third-dimension relations. For example

- rule for ClassifierRole in page 2-104 in [16] saying that the features of the ClassifierRole must be a subset of those of the base Classifier:

Self.base.allFeatures->includesAll
(self.allAvailableFeatures)

- rule for Association in page 2-42 in [16] stating that the connected Classifiers of the AssociationEnds should be included in the NameSpace of the Association:

self.allConnections->forAll
(r | self.nameSpace.allContents->includes (r.type))

Furthermore, the building of a formal model allowed us to find out and correct ambiguities and inconsistencies in the UML Language. For example, a classifier role is a description of the features required in a particular collaboration , i.e. a classifier role is a projection of a classifier. The classifier so represented is referred to as the base classifier. Collaboration, classifier and classifier roles are generalizable elements One possible way to specialize a collaboration is to specialize some classifier role in the collaboration. The UML specification document gives a set of OCL rules to restrict generalization relation between collaborations. The rule number 5 in page 2-106 in [16] states that "a role whit the same name as one if the roles in a parent of the Collaboration must be a child (a specialization) of that role". This rule is expressed by the formula:

$\forall$s:Collaboration •$\forall$c$\in$ s.contents
•$\forall$p$\in$ s.parent.allContents•
(c.name=p.name $\rightarrow$ p$\in$ c.allParents)

This rule is too restrictive, since the specialization of a classifier role could be accomplished in other ways. For example the rule above should be extended in the following way:

$\forall$s:Collaboration •
$\forall$c$\in$ s.contents • $\forall$p$\in$ s.parent.allContents•
(c.name=p.name $\rightarrow$
(p$\in$ c.allParents
$\vee$ (p.allAvailableFeatures$\subseteq$ c.allAvailableFeatures $\wedge$
p.base$\in$ c.base.allParents)))

On the other hand it in necesary to define compatibility rules among the different views of a system

(e.g. Class diagrams, Statecharts, etc.). We give examples of compatibility rules:

Example 1: Pre/post conditions vs. State Machines

Any model element may be associated with a constraint that expresses some property of it. There are problems when the constrained element has also a behavior that is precisely defined else where in the model. For example, a constraint on an operation (as a pre-post condition) may be inconsistent with the effects of the transitions triggered by its calls in the associated state machine. As a consequence, it is necessary to integrate both views of the system guaranteeing that they are consistent with each other. The following rule formalizes this requirement:

Example 2: Generalizations vs. other elements

Generalization diagrams have a strong influence on other diagrams in the model of the system.

For example, if two classes c_1 and c_2 are connected by a generalization relation (e.g. c_1 is a subclass of c_2), the behavior of instances of c_1 should be a refinement of the behavior of instances of c_1. This requirement is defined by the following formula:

$\forall c_1,c_2$:Classifier• ($IsA(c_1,c_2) \rightarrow$
refinement(behavior(c_1) , behavior(c_2)))

A similar problem occurs when constraints are linked to classes in a generalization hierarchy: if c_1 is a subclass of c_2 then all the constraints over c_1 should be consistent with all the constraints on c_2. This requirement is expressed by the following formula:

$\forall c_1,c_2$:Classifier• ($IsA(c_1,c_2) \rightarrow$
consistent(constraints(c_1) $\cup$constraints(c_2)))

5. Relations between relations

As well as the different models of a system are not independent, the different relationships among models neither are independent. In this section we point out some properties of relationships.

Theorem: Let UC1 and UC2 be use cases. If UC2 is an extension of UC1 through *ext* , and each use case is realized by a corresponding collaboration, then there exists a collaboration realizing UC2 such that it is an extension of C1:

$\forall$UC1, UC2: UseCases• $\forall$C1, C2: Collaborations •
((UC2 $extends_{ext}$ UC1 $\wedge$ C1$\geq_{\delta}$ UC1 $\wedge$ C2 $\geq_{\delta}$ UC2) $\rightarrow$
$\exists$C3• (C3 $extends_{ext}$ C1 $\wedge$ C3 $\geq_{\delta}$ UC2))

Figure 10 illustrates the relation between the extend relationship and the realization relationship.

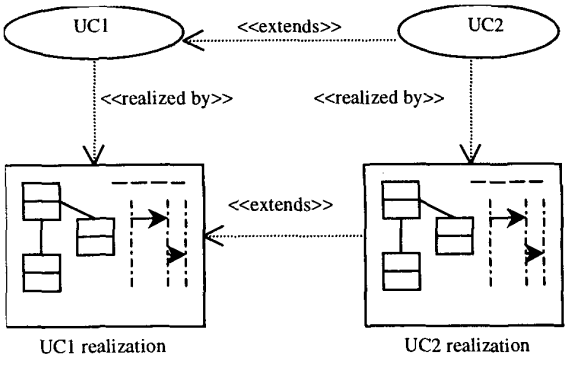

Figure 10: relations between relations in the Unified Process

6. Concluding Remarks

During the Unified Process, a variety of models of the system is developed. All these models are not independent, but they are related to each other. Elements in one model have trace dependencies to other models; they are semantically overlapping and together represent the system as a whole.

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

In this paper we distinguish three different kinds of dependency relations between models – workflows, iterations and artifacts – and propose a formal description of them. The goal of the proposed formalization is to provide formal foundations for tools that perform intelligent operation on models, such as:

- checking the consistency between models belonging to different workflows, such as a requirements model and an analysis model (i.e. consistency along the workflows dimension).
- checking the consistency of models through its evolution along the process (i.e. consistency along the time dimension)
- checking the internal consistency of models (i.e. consistency along the artifact dimension).
- checking the consistency of the process as a whole (i.e. consistency among the different dimension).

A step beyond this work will be to use the formalization to define automatic rules of evolution that assist the software engineer during the development process. For example, given an analysis model the rules could suggest possible forms of realizing such a model in terms of design models, and given a model the rules could suggest possible way to refine or to extend it.

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 Porres Paltor I., 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] Cibrán, M.A., 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. Cannes/Sophia-Antipolis, France, June 2000.

[5] 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).

[6] 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).

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

[8] Jacobson, I..Booch, G Rumbaugh, J., The Unified Software Development Process, Addison Wesley. ISBN 0-201-57169-2 (1999)

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

[10] 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).

[11] Övergaard, G., and Palmkvist,K., 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 Science 1618. Springer-Verlag, (1998).

[12] Övergaard, G., A formal approach to collaborations in the UML, <<UML>>´99 - The Unified Modeling Language. Beyond the Standard. In UML´99 conference, Colorado, USA,. Lecture Notes in Computer Science 1723, Springer. (1999).

[13] 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.

[14] Pons,C., Baum,G., Felder,M., Foundations of Object-oriented modeling notations in a dynamic logic framework, Fundamentals of Information Systems, Chapter 1, Kluwer Academic Publisher, (1999).

[15] Pons C., Giandini,R. and Baum,G, Relations between different models in the Unified Process. Pons, C., Giandini,R. and Baum,G. International Workshop on Model Engineering (IWME´00), ECOOP´2000. Cannes/Sophia-Antipolis,, France, June 2000.

[16] The Unified Modeling Language (UML) Specification – Version 1.3, July 1999. UML Specification, revised by the OMG, http://www.rational.com/

[17] Reggio,G., Astesiano,E., Choppy, C. and Hussmann,H., Analysing UML active classes and associated state machines. In Proc. FASE 2000 – Foundamental Approaches to Software Engineering, LNCS 1783, Spring Verlag, 2000.

[18] Smith, J., DeLoach,S., Kokak,M.and Baclawski,K, Category theoretic approaches of representing precise UML semantics. in Proc. ECOOP Workshop on Defining Precise Semantics for UML, France, June 2000.