Software Development Contracts

Claudia Pons Gabriel Baum

LIFIA – Universidad Nacional de La Plata
50 esq.115. 1er Piso
CP 1900 Buenos Aires, Argentina
email: cpons@info.unlp.edu.ar

Abstract

While the notion of formal contract regulating the behavior of software agents is accepted, the concept of contract regulating the activities of software developers is quite vague. In general there is not documented contract establishing obligations and benefits of members of the development team. However, a disciplined software development methodology should encourage the use of formal contracts between developers.

We propose to apply the notion of formal contract to the object-oriented software development process itself. That is to say, the software development process can be seen as involving a number of agents (the development team and the software artifacts) carrying out actions with the goal of building a software system that meets the user requirements. In this way, contracts can be used to reason about correctness of the development process, and comparing the capabilities of various groupings of agents (coalitions) in order to accomplish a particular contract.

Keywords: object-oriented software development process, process modeling, formal methods, refinement calculus, contract.

1. Introduction

Object-oriented software development process (e.g. The Unified Process [Jacobson et al., 1999], Catalysis [D´Souza and Wills, 1998], Fusion [Coleman et al. 1994]) is a set of activities needed to transform user's requirements into a software system. A software development process typically consists of a set of software development artifacts together with a graph of tasks and activities. Software artifacts are the products resulting from software development, for example, a use case model, a class model or source code. Tasks are small behavioral units that usually results in a software artifact. Examples of tasks are construction of a use case model, construction of a class model and writing code. Activities (or workflows) are units that are larger than a task. Activities generally include several tasks and software artifacts. Examples of activities are requirements, analysis, design and implementation.

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. 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. For instance, a use case (in the use-case model) can be traced to a collaboration (in the design model) representing its realization.

On the other hand, due to the incremental nature of the process, each iteration results in an increment of artifacts built in the former iteration. An increment is not necessarily additive. Generally in the early phases of the life cycle, a superficial artifact 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.

Figure 1 lists the classical activities – requirements, analysis, design, implementation and test – in the vertical axis and the iteration in the horizontal axis , showing the following kinds of relations:

-horizontal relations between artifacts belonging to the same activity in different iterations (a use case is extended by another use case)

-vertical relations between artifacts belonging to the same iteration in different activities (e.g. an analysis model is realized by a design model).

Traditional specifications of development process typically consist of quite informal descriptions of a set of software development artifacts together with a graph of tasks and activities. But, the software development process should be formally defined since the lack of accuracy in its definition can cause problems, for example:

- Inconsistency among the different artifacts: 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.

- Evolution conflicts: when a artifact is modified, unexpected behavior may occur in other artifacts that depend on it.

- Confusion regarding the order in which tasks should be carried out by developers.

- It is not possible to reason about the correctness of the development process.

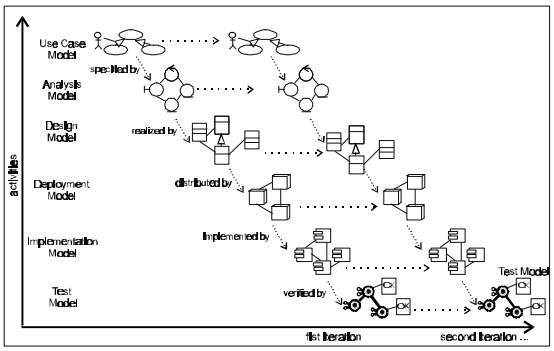

Figure 1. dimensions in the development process

We propose to apply the well-known mathematical concept of contract to the description of software development processes in order to introduce precision of specification, avoiding ambiguities and inconsistencies, and enabling developers to reason about the correctness of their activities. Furthermore, development contracts are organized in a modular and hierarchical way leading to a better understanding of the whole software development process.

2. The notion of software contract

A computation can generally be seen as involving a number of agents (objects) carrying out actions according to a document (specification, program) that has been laid out in advance. This document represents a contract between the agents involved. The notion of contract regulating the behavior of a software system has been already introduced by several authors [Helm et. al 90, Meyer 91, Meyer 97, Back and von Wright, 98; Andrade and Fiadeiro 99]. A contract imposes mutual obligations and benefits. It protects both sides (the client and the contractor):

- It protects the client by specifying how much should be done: the client is entitled to receive a certain result.

- It protects the contractor by specifying how little is acceptable: the contractor must not be liable for failing to carry out tasks outside of the specified scope.

As example consider the contract in figure 2, in which a *subject* object, containing some data, and a collection of *view* objects, which represent the data graphically, cooperate so that at all times each *view* always reflects the current value of the *subject*. This contract defines the behavioral composition of subject and views participants. The contract specifies the following aspects: firstly, it identifies type obligations, where the participant must support certain external interface, and causal obligations, where the participant must perform an ordered sequence of actions and make certain conditions true in response to these messages. Secondly, the contract defines invariants that participants cooperate to maintain.

contract SubjectView

Participants: Subject, View

Subject supports [
data: Data
setData(val:Data) Ä Ddata {data=val};notify()
notify() Ä ⟨∀v:v∈ views: v.update()⟩
attachView(v:View) Ä {v∈ views}
detachView(v:View) Ä {v∉ views}]
Views:Set(View) where each View supports [
update() Ä {view *reflects* subject.data}
setSubject(s:Subject) Ä {subject=s)}]
invariant
subject.setData(val) Ä
⟨∀v∈ views: v *reflects* subject.data⟩

end contract

Figure 2: contract SubjectView [Helm et al, 90]

3. Software contracts as mathematical entities

We take the view of contracts as proposed by [Back and von Wright, 98] and [Back et al., 99]. The world that a contract talks about is described as a state σ. The state space Σ is the set of all possible states σ. The state is observed as a collection of attributes x_1, x_2, ...,x_n, each of which can be observed and changed independently of the others. Attributes are partitioned into objects

An agent changes the state by applying a function f to the present state σ, yielding a new state f.σ. A function f:Σ→ Σ that maps states to states is called state transformer. An example of state transformer is the assignment x:=exp, that updates the value of attribute x to the value of the expression exp.

A boolean function p: Σ→Bool is called a state predicate. A state relation R:Σ→ Σ→ Bool relates a state σ to a state σ´ whenever R.σ.σ´holds.

Assume that there is a fixed collection A of agents. Let ***a***, ***b***, ***c*** denote individual agents. We describe contracts using the notation for contract statements [Back and von Wright, 98]. The syntax for these is as follows:

S ::= ⟨f⟩ | if p then S_1 else S_2 fi |S_1 ; S_2| $assert_a$ p | R_a | $choice_a$ S_1 U S_2 | while p do S_1 od

Here a stands for an agent while f stands for a state transformer, p for a state predicate, and R for a state relation, both expressed using higher-order logic. Intuitively, a contract statement is executed as follows:

The functional update <f> changes the state according to the state transformer f, i.e., if the initial state is σ_0 then the final state is f. σ_0 . An assignment statement is a

special kind of update where the state transformer is expressed as an assignment. For example, the assignment statement <x:=x+y> requires the agent to set the value of attribute x to the sum of the values of attributes x and y. The name skip is used for the identity update <id>, where $id.\sigma = \sigma$ for all states σ.

In the conditional composition if p then S_1 else S_2 fi, S_1 is carried out if p holds in the initial state, and S_2 otherwise.

In the sequential composition S_1 ; S_2 , statement S_1 is first carried out, followed by S_2.

An assertion $assert_a$ p , for example, $assert_a$ (x+y=0) expresses that the sum of (the values of) x and y in the state must be zero. If the assertion holds at the indicated place when the agent a carries out the contract, then the state is unchanged, and the rest of the contract is carried out. If, on the other hand, the assertion does not hold, then the agent has breached the contract.

The relational update and choice both introduce non-determinism into the language of contracts. Both are indexed by an agent which is responsible for deciding how the non-determinism is resolved.

The relational update R_a requires the agent ***a*** to choose a final state σ' so that $R.\sigma.\sigma'$ is satisfied, where σ is the initial state. In practice, the relation is expressed as a relational assignment. For example, $update_a$ {x := x´| x´ <x} expresses that the agent ***a*** is required to decrease the value of the program variable x. If it is impossible for the agent to satisfy this, then the agent has breached the contract.

The statement $choice_a$ S_1 U S_2 allows agent ***a*** to choose which is to be carried out, S_1 or S_2 .

Finally, recursive contract statements are allowed. A recursive contract is defined using an equation of the form X = S. where S may contain occurrences of the contract variable X. With this definition, the contract X is intuitively interpreted as the contract statement S, but with each occurrence of statement variable X in S treated as a recursive invocation of the whole contract S. Also it is permitted the syntax (rec X•S) for the contract X defined by the equation X=S. An important special case of recursion is the while-loop which is defined in the usual way: while p do S od =(rec X•if p then S ; X else skip fi)

3.1 Predicate transformer semantics (Weakest preconditions)

In order to analyze a contract it is necessary to express the precise meaning of each statement , i.e. we need the semantics of contract statements. The semantics is given within the refinement calculus using the weakest precondition predicate transformer [Back and von Wright, 98].

A predicate transformer is a function that maps predicates to predicates. Predicate transformers are ordered by pointwise extension of the ordering on predicates, so $F \subseteq F'$ for predicate transformers holds if and only if $F.q \subseteq F'.q$ for all predicates q. The predicate transformers form a complete lattice with this ordering, and $\cup$ and $\cap$ are the operators of this lattice.

Different agents are unlikely to have the same goals, and the way one agent makes its choices need not be suitable for another agent. From the point of view of a specific agent or a group of agents, it is therefore interesting to know what outcomes are possible regardless of how the other agents resolve their choices.

Consider the situation where the initial state σ is given and a group of agents A agree that their common goal is to use contract S to reach a final state in some set q of desired final states. It is also acceptable that the coalition is released from the contract, because some other agent breaches the contract. This means that the agents should strive to make their choices in such a way that the scenario starting from σ ends in a configuration σ', where either σ' is an element in q, or some other agent has breached the contract.

Assume that S is a contract statement and A a coalition, i.e., a set of agents. We want the predicate transformer $wp_A.S$ to map postcondition q to the set of all initial states σ from which the agents in A jointly have a winning strategy to reach the goal q. Thus, $wp_A.S.q$ is the weakest precondition that guarantees that the agents in A can cooperate to achieve postcondition q. This means that a contract S for a coalition A is mathematically seen as an element (denoted by $wp_A.S$) of the domain $P\Sigma \rightarrow P\Sigma$

These definitions are consistent with Dijkstra original semantics for the language of guarded commands [Dijkstra, 76] and with later extensions to it, corresponding to non-deterministic assignments, choices, and miracles.

The definition of the weakest precondition semantics is as follows (see[Back and von Wright, 98] for a more detailed explanation):

$wp_A.\langle f \rangle.q = (\lambda\sigma.q.(f.\sigma))$

$wp_A.$(if p then S_1 else S_2 fi).q =
$(p \cap wp_A.S_1.q) \cup (\neg p \cap wp_A.S_2.q)$

$wp_A.(S_1;S_2).q = wp_A.S_1.(wp_A.S_2.q)$

$wp_A.(assert_a\ p).q = \lambda\sigma.(p.\sigma \wedge q.\sigma)$, if $a \in A$
$\lambda\sigma.(\neg p.\sigma \vee q.\sigma)$, if $a \notin A$

$wp_A.R_a.q = \lambda\sigma.\exists\sigma' \bullet R.\sigma.\sigma' \wedge q.\sigma'$, if $a \in A$
$\lambda\sigma.\forall\sigma' \bullet R.\sigma.\sigma' \rightarrow q.\sigma'$, if $a \notin A$

$wp_A.(choice_a\ S_1\ U\ S_2).q = wp_A.S_1.q \cup wp_A.S_2.q$, if $a \in A$
$wp_A.S_1.q \cap wp_A.S_2.q$, if $a \notin A$

4. The notion of software development contract

The notion of formal contract described in section 3, can be applied to the software development process itself. That is to say, the software development process can be seen as involving a number of agents (the development team and the software artifacts) who carry out actions with

the goal of building a software system that meets the user requirements.

While the notion of formal contract regulating the behavior of software agents is accepted, the concept of contract regulating the activities of software developers is quite vague. In general there is not documented contract establishing obligations and benefits of members of the development team. As we remarked in section 1, in the best of the cases the development process is specified by either graph of tasks or object-oriented diagrams in a semi-formal style, while in most of the cases activities are carried out on demand, with little previous planning.

However, a disciplined software development methodology should encourage the existence of formal contracts between developers, so that contracts can be used to reason about correctness of the development process, and comparing the capabilities of various groupings of agents (coalitions) in order to accomplish a particular contract.

Assume you are planning a work to be performed by a development team in order to adapt the model of a system to new requirements (e.g. during the n+1 iteration of the development process). This work can be expressed as a combination (in sequence or in parallel) of sub-works, each of them to be performed by a member of the development team. It is necessary to make sure that sub-works will be performed as required. This is only possible if the agreement is spelled out precisely in a contract document. This idea is based on the metaphor: software development is a sequence of documented contract decisions.

A remarkable difference between software contracts and development contracts is the kind of object constituting a state. While in software contracts, objects in the state represent object in a system, such as a bank account or a book, in software development contracts, objects in the state are development artifacts, such as a class diagram or a use case. But this difference is just conceptual, from the mathematical point of view we can reason about development contracts in the standard way, as if they were software contracts.

There are different levels of granularity in which development contracts can be defined. On one hand we have contracts regulating primitive evolution, such as adding a single class in a Class diagram, while on the other hand we have contracts defining complex evolution, such as the realization of a use case in the analysis phase by a collaboration diagram in the design phase, or the reorganization of a complete class hierarchy. Complex evolution are not atomic tasks, instead they are made up with primitive evolutions. So, we start specifying atomic contracts (contracts explaining primitive evolution) which will be the building blocks for non-atomic contracts (i.e. regulations for complex evolution).

4.1 Primitive development-contracts

In order to specify primitive development-contracts we may associate a precondition and a postcondition with each primitive evolution operation on models.

In order to make contracts more understandable and extensible, we use the object-oriented approach to specify them. The object oriented approach deals with the complexity of description of software development process better than the traditional approach. Examples of this are the framework for describing UML compatible development processes defined in [Hruby 99] and the metamodel defined by the OMG Process Working Group [OMG 98], among others. In the object-oriented approach, software artifacts produced during the development process are considered objects with methods and attributes. Evolution during the software development process is represented as collaborations between software artifacts and users of the method.

We use the following object oriented syntax for specifying classes of artifacts:

Specification of ClassName	
Superclasses list of direct superclasses	
Attributes	
	list of attributes and associations.
Derived Attributes	
	list of attributes and associations whose values can be calculated from other attributes or associations.
Predicates	
	list of boolean functions
Invariants	
	list of predicates that should be true in all states.
Operations	
	list of method declarations
End specification of ClassName	

Where a method declaration has a name m, a precondition p and an effect S (the body of the method). When a method is called there is an agent ***a*** responsible for the call. The method invocation is then interpreted as ´$assert_a$ p ; S´, i.e. the agent is responsible for verifying the preconditions of the method. If agent ***a*** has invoked the method in a state that does not satisfy the precondition, then ***a*** has breached the contract.

At the present the Unified Modeling Language [UML, 2000] is considered the standard modeling language for object oriented software development process. As example, we present the evolution contracts of some UML artifacts. Lets consider a part of the UML metamodel describing Class, Feature, Package and Generalization artifacts. The contract for some primitive operations on these artifacts can be specified as follows (parts of the specification are omitted due to space limitations):

Specification of GeneralizableElement	
Superclasses ModelElement	
Attributes	
	generalizations: Set of Generalization specializations: Set of Generalization

	isAbstract: Bool
Derived Attributes	
	[1] c.parents returns the set of direct parents of c. parents: Set of GeneralizableElement c.parents=c.generalizations.collect(parent) [2] c.children returns the set of direct child of c. children: Set of GeneralizableElement c.children = c.specializations.collect(child)
Predicates	
	IsA : GeneralizableElement x GeneralizableElement →Bool IsA(c,c1) ↔ c=c1 ∨ c1∈ c.allParents
Invariants ∀ c_1,c_2 : GeneralizableElement	
	[1] Circular inheritance is not allowed. $IsA(c_1,c_2) \wedge IsA(c_2,c_1) \rightarrow c_2 = c_1$
End specification of GeneralizableElement	

Specification of Classifier	
Suplerclasses GeneralizableElement, NameSpace	
Attributes	
	features: Seq of Feature associationEnds: Set of AssociationEnd
Derived Attributes	
	[1] The operation allFeatures results in a Set containing all Features of the Classifier itself and all its inherited Features. allFeatures : Set of Feature c.allFeatures = c.features ∪ ($\cup_{ci\in c.parents}$ ci.allFeatures) [2] The operation allAssociationEnds results in a Set containing all AssociationEnds of the Classifier itself and all its inherited associationEnds. allAssociationEnds: Set of AssociationEnd c.allAssociationEnds= c.associationEnds ∪ ($\cup_{ci\in c.parents}$ ci.allAssociationEnds) [3] The operation oppositeAssociationEnds results in a set of all AssociationEnds that are opposite to the classifier. .....
Invariants ∀c:Classifier	
	[1] No Attributes may have the same name within a Classifier ∀f,g∈ c.attributes (f.name=g.name →f=g) [2] No Operations may have the same signature in a Classifier. ∀f,g∈ c.operations ((hasSameSignature(f,g) →f=g)
Operations	
	proc c.addFeature (f:Feature) ***Precondition*** [1] The class exists (it is stored in some package) c.package≠null [1] the new Feature does not belong to c f∉ c.attributes [2]No Features may have the same name within a Classifier ∀g∈ attributes(c) f.name≠ g.name [3] The name of an Attribute cannot be the same as the name of an opposite AssociationEnd. ∀e∈ c.oppositeAssociationEnds f.name≠e.name [4] The connected type should be included in the Package of the Classifier. f.type∈ (c.package).allContents ***Effect*** [1] the feature is added to the list of features c.features:=c.features∪{f} ; f.owner:=c
End specification of Class	

Specification of Package	
Superclasses NameSpace, GeneralizableElement	
Attributes	
	importedElements: Set of ModelElement ownedElements: Set of ModelElement
Derived Attributes	
	[1] The operation contents results in a set containing all ModelElements owned or imported by the Package. contents : Set of ModelElement p.contents = p.ownedElements ∪ p.importedElements
Invariants ∀p: Package	
	[1] in a Package the Classifier names are unique ∀c_1,c_2: Classifier ((c_1∈p.contents ∧ c_2∈ p.contents ∧ c_1.name = c_2.name) → c_1 =c_2)
Operations	
	proc p.addGeneralization (g:Generalization) ***Precondition*** [1] the generalization is not in the package g∉ p.allContents [2] all elements connected by the new relationship must be included in the Package. g.parent∈ p.allContents ∧ g.child∈ p.allContents [5] Circular inheritance. *IsA*(g.parent, g.child) → g.parent = g.child [6] multiple inheritance. ∀c:Classifier (*IsA*(g.child,c) → ∀f,g:Feature((f∈ ((g.parent).allFeatures) ∧ g∈ c.allFeatures ∧ f.name=g.name) → f=g)) ***Effect*** [1] the generalization is inserted into the package p.ownedElement::= p.ownedElement ∪ {g}; g.package:=p [2] The new generalization is linked to the generalizable elements g.parent.specializations := g.parent.specializations ∪ {g}; g.child.generalizations := g.child.generalizations ∪ {g} .....
End specification of Package	

4.2 Complex development-contracts

On top of primitive contracts it is possible to define complex contracts, specifying non-atomic forms of evolution through the software development process. Then, by using the wp predicate transformer we can verify whether a set of agents (i.e. software developers) can achieve their goal or not. We can analyze whether a developer (or team of developers) can apply a group of modifications on a model or not by means of a contract designed in terms of a set of primitive operations conforming the group.

Developers will successfully carry out the modifications if some preconditions hold. We can determine the weakest preconditions to achieve a goal by computing:

wp_A . C . Q

where C is the contract, A is the set of software developers (agents) and Q is the goal.

If computing the wp we obtain a predicate different from false, then we proved that with the contract the developers can achieve their goal under certain pre-conditions.

Example 1: a collaborative work

Lets consider a collaborative work, in which three software developers have to modify a class diagram. One of the agents will detect and delete all the features that could be lifted to a superclass (i.e. features that appear repeated in all of the subclasses of a given class). The second agent has the responsibility of lifting the feature (i.e. to add the deleted feature in the superclass). As a consequence of the lifting process, some classes may become empty (i.e. without proper features). Finally the third agent will detect and delete empty classes. Figure 3 illustrates the collaborative process described above.

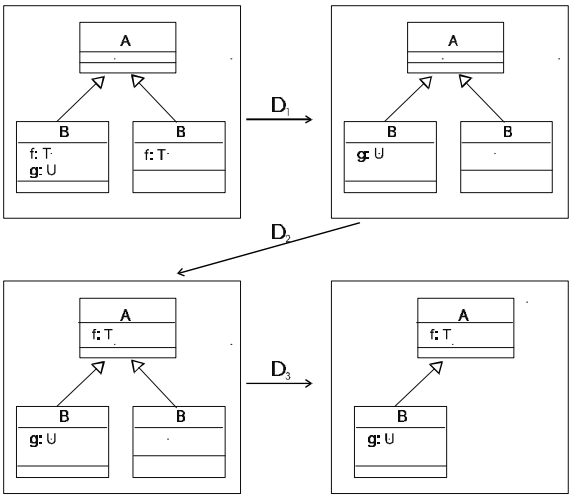

Figure 3: the collaborative refactoring task

We are interested in calculating the weakest precondition for agents D_1 , D_2 and D_3 to reach the goal Q by using the contract R. That is to say:

$wp_{\{D1,D2,D3\}}$. R. Q

Where:

Def 1: the contract

R ≡ CONTRACT refactoring
agents D, D_1, D_2, D_3
var p:Package, c:Class, f:Feature
proc liftingRepeatedFeature:
$update_{D1}$ c:=s | ∃f:Feature • (∀c´∈s.subclasses • f∈c'.features) ;
$update_{D1}$ f:=f´ | ∀c´∈ c.subclasses • f´∈ c'.features ;
while (∃c´∈c.subclasses • f∈c'.features)
do $update_{D1}$ c´:=c´´ | c´´∈ c.subclasses ∧ f∈c''.features ;
c´.deleteFeature(f)$_{D1}$;
od;
c.addFeature(f)$_{D2}$;
end proc.

proc deletingEmptyClass:
$update_{D3}$ c:=c´ | c'.features≠∅;
p.deleteClass(c)$_{D3}$;
end proc.

begin
while (¬Q)
do $choice_D$ liftingRepeatedFeature U deletingEmptyClass
od;
end.

Def 2: the postcondition

Q ≡ $q_1 \wedge q_2$
where:
q_1 ≡ ∀c:Class • ¬∃f:Feature • (∀c´∈c.subclasses • f∈c'.features)
q_2 ≡ ∀c:Class • c.features≠∅

Q specifies the expected effect of the refactoring process as the combination of two facts: q_1 says that there are no repeated features while q_2 specifies that the model does not contain any empty class.

Example 2: Using contracts to reasoning about evolution conflicts

Arbitrary modifications that do not cause problems when they are applied exclusively, may rise conflicts when they are integrated (i.e. they are applied together). For example if both evolutions - deleting a class and adding a feature to the class- are applied sequentially a conflict may occur because it is not possible to add a feature to a missing class.

C ≡ CONTRACT conflict
agent D1, D2

var p:Package, c:Class, f:Feature;
begin
$p.delClass(c)_{D1}$; $c.addFeature(f)_{D2}$
end.

We can prove that $wp_{\{D1,D2\}}$. C. Q is false. Where Q is any predicate. It is impossible for agents D1 and D2 to carry out the contract.

Example 3: checking consistency between artifacts

Lets consider a collaborative work in which two agents D1 and D2 need to add a generalization relationship respectively, preserving the well formedness property of the model.

In particular, it is possible to find out which is the weakest precondition to achieve the goal of introducing two generalization relationships without breaking the non-circularity principle of inheritance hierarchies by computing:

$wp_{\{D1,D2\}}$. C . Q

Where C is the contract between agents and Q is a predicate that specifies absence of circularity in the hierarchies and that the new relationships were established.

We will calculate the weakest precondition for agents D_1 and D_2 to reach the goal Q by using the contract C, That is to say:

$wp_{\{D1,D2\}}$. C. Q = P

where:

- C ≡ CONTRACT circular
agents D1, D2
var p:Package, r,g:Generalization;
begin
$p.addGeneralization(r)_{D1}$; $p.addGeneralization(g)_{D2}$
end.

- $Q \equiv q \wedge q'$

Where q specifies the effect of the evolution (generalizations were added in the package) and q´ specifies a well-formedness rule (there is no circular inheritance).

$q \equiv (r \in p.ownedElements \wedge g \in p.ownedElements)$

$q' \equiv \forall c_1,c_2$:GeneralizableElement. ($IsA(c_1,c_2) \wedge IsA(c_2,c_1) \rightarrow c_2 = c_1$)

Finally, the expected weakest precondition is as follows:

- $P \equiv P_1 \wedge P_2 \wedge H$

Where P_1 an P_2 specify preconditions for applying the first and the second evolutions respectively (as if they were applied in isolation). And H specifies a special requirement to avoid circular inheritance in case both evolution actions are applied together. $P_1 \equiv$ $r \notin p.allContents \wedge r.parent \in p.allContents \wedge r.child \in p.allContents \wedge$

$IsA(r.parent, r.child) \rightarrow r.parent = r.child$

$P_2 \equiv g \notin p.allContents \wedge g.parent \in p.allContents \wedge g.child \in p.allContents \wedge$

$IsA(g.parent, g.child) \rightarrow g.parent = g.child$

$H \equiv \neg (IsA(g.parent, r.child) \wedge IsA(r.parent, g.child))$

The complete derivation can be read in [Pons and Baum, 2001] Figure 4 illustrate a conflictive case, in which the expected weakest pre-condition does not hold.

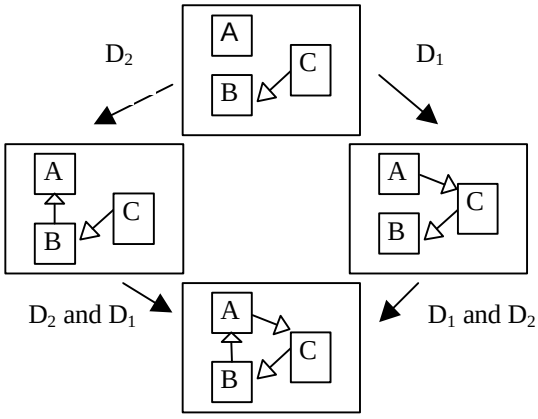

Figure 4: evolution conflict

5. Conclusion and related work

Software development process is a collaborative process. As a consequence it is necessary to formally specify benefits and obligations of partners involved in the process in order to avoid misunderstandings and conflicts.

We apply the well-known mathematical concept of contract to the specification of software development processes in order to introduce precision of specification, avoiding ambiguities and inconsistencies, and enabling developers to reason about the correctness of their joint activities.

Contracts provide a formalization of software artifacts and their relationships. Also contracts clearly establish pre and post conditions for each software development activity. The goal of the proposed formalism is to provide foundations for tools that assist software engineers during the development process.

In general there is not documented contract establishing obligations and benefits of members of the development team, i.e. software development processes are specified in a semi-formal style. For example the specification of the standard graphical modeling notation UML [UML, 2000] and the Unified Process [Jacobson et al., 99] 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 UML and providing rules for analyzing their properties; see, for instance [Back et al. 99; Breu et al., 1997; Evans et al., 1999; Kim and Carrington, 1999; Övergaard 1999; Pons et al. 1999, Pons et al 2000], while less effort has been dedicated to the formalization of UML compatible software development processes.

The mechanism of development contracts introduced in this paper, is related to the mechanism of reuse contracts [Steyaert et al. 96, Lucas 97]. A reuse contract describes a set of interacting participants. Reuse contracts can only be adapted by means of reuse operators that record both the protocol between developers and users of a reusable component and the relationship between different versions of one component that has evolved. Similarly, in [Mens et al. 2000] the authors translate the idea of reuse contracts in order to cope with reuse and evolution of UML models.

The originality of development contracts resides in the fact that software developers are incorporated into the formalism as agents (or coalition of agents) who make decisions and have responsibilities. Given a specific goal that a coalition of agents is requested to achieve , we can use traditional correctness reasoning to show that the goal can in fact be achieved by the coalition, regardless of how the remaining agents act. The wp formalism allows us to analyze a single contract from the point of view of different coalitions and compare the results. For example, it is possible to study whether a given coalition *A* would gain anything by permitting an outside agent *b* to join *A*.

Finally, sine the construction of formal development contracts is a hard task, it is important to consider evolution and reuse of contracts themselves. As contracts are written in an object-oriented style, it is possible to define a new contract by specializing an existing one. This feature does not solve the complexity problem completely, but it facilitates the task of creation and evolution of contracts.

References

Andrade,L and Fiadeiro,J.L, Interconecting objects via Contracts. Proceedings of the UML´99 conference, Lecture Notes in Computer Science 1723, Springer Verlag. (1999).

Back, R and von Wright, J., Refinement Calculus: A Systematic Introduction, Graduate texts in Computer Science, Springer Verlag, 1998.

Back, R. Petre L. and Porres Paltor I., Analysing UML Use Cases as Contract. .Procs of the UML´99 conference, Lecture Notes in Computer Science 1723, Springer. (1999).

Coleman, D..Arnols, P Bodoff,S, Dollin, C, Gilchrist,H, Hayes,F, Jeremaes,P. Object Oriented Development: The Fusion Method. Prentice-Hall 1994.

Dijkstra, E, A Discipline of Programming. Prentice Hall International, 1976.

D´Souza D. and Wills, A. Objects, Components and Frameworks with UML: the Catalysis approach, Addison Wesley, 1998.

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).

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).

Helm,R. Holland,I and Gangopadhyay,D. Contracts: specifying behavioral compositions in object-oriented systems, Proc. OOPSLA'90. ACM Press. Oct 1990.

Hruby, Pavel, Framework for describing UML compatible development processes. in <<UML>>´99 - The Unified Modeling Language. Beyond the Standard. R.France and B.Rumpe editors, Proceedings of the UML´99 conference, Lecture Notes in Computer Science 1723, Springer Verlag. (1999).

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

Kim, S. and Carrington,D., Formalizing the UML Class Diagrams using Object-Z, In Proc. <<UML>>´99 - The Second International Conference on the Unified Modeling Language, Lecture Notes in Computer Sciencie 1723, (1999).

Lucas, Carine "Documenting Reuse and evolution with reuse contracts", PhD Dissertation, Programming Technology Lab, Vrije Universiteit Brussel, September 1997.

Mens,T., Lucas,C. and D'Hondt, T.. Automating support for software evolution in UML. Automated Software Engineering Journal 7:1, Kluwer Academic Publishers, February 2000.

Meyer, B. Advances in object oriented software engineering. Chapter 1 "Design by contract". Prentice Hall, 1992.

Meyer,B. Object-Oriented Software Construction, Second Edition, Prentice Hall, 1997.

OMG White Paper on Analysis and Design Process Engineering, Process Working Group, Analysis and Design Platform Task Force, OMG Document. July 1998.

Övergaard, G., A formal approach to collaborations in the UML, In Proc. <<UML>>´99 - The Second International Conference on the Unified Modeling Language. R.France and B.Rumpe editors, Lecture Notes in Computer Science 1723, Springer. (1999).

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.

Pons, C., Giandini, R. and Baum, G. Specifying Relationships between models through the software development process Tenth International Workshop on Software Specification and Design (IWSSD), California, IEEE Computer Society Press. November 2000.

Pons,C and Baum G. Software Development Contracts, extended version.In www-lifia.info.unlp.edu.ar/~cpons

Steyaert, P..Lucas, C.Mens K and D'Hondt, T. Reuse Contracts: Managing the evolution of reusable assets. In proceedings of OOPSLA'96, New York, Oct 1996.

UML, The Unified Modeling Language Specification – Version 1.3,. UML Specification, revised by the OMG, http://www.omg.org, March, 2000.