Chapter IX

Formal Specification of Software Model Evolution Using Contracts

Claudia Pons, Universidad Nacional de La Plata, Argentina

Gabriel Baum, Universidad Nacional de La Plata, Argentina

Abstract

During the object-oriented software development process, a variety of models of the system is built. All these models are semantically overlapping and together represent the system as a whole. In this chapter, we present a classification of relationships between models along three different dimensions, proposing a formal description of them in terms of mathematical contracts, where the software development process is 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 to compare the capabilities of various groupings of agents in order to accomplish a particular contract. The goal of the proposed formalization is to provide formal foundations for tools that perform intelligent analysis on models assisting software engineers through the software life cycle.

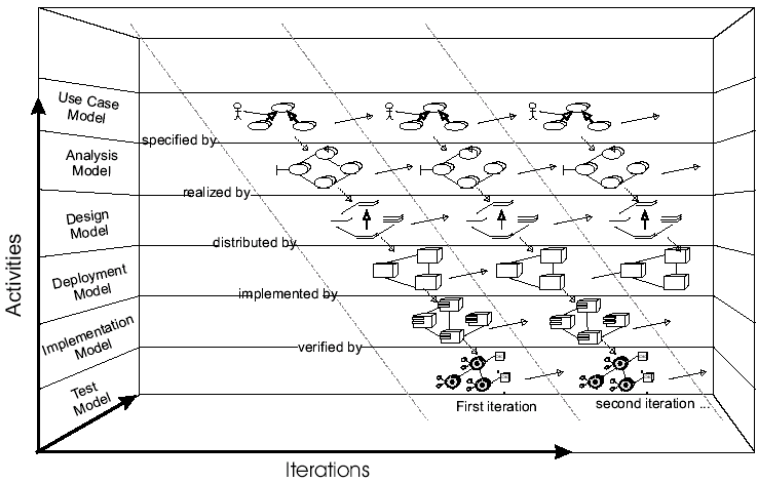
Introduction

A software development process is a set of activities that jointly convert users' needs to a software system. Modern software development processes, such as the Unified Process (Jacobson, Booch, & Rumbaugh, 1999), 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 consists of one pass through the requirements, analysis, design, implementation, and test activities, building a number of different models. Due to the incremental nature of the process, each iteration results in an increment of models built in previous iterations. This creates a natural relationship between the elements among different phases and iterations; elements in one model can be related to elements in another model. For instance, a use case (in the use case model) can be traced to a collaboration (in the analysis or design model) representing its realization. Figure 1 lists the classical phases or activities – requirements, analysis, design, implementation, and test – in the vertical axis and the iteration in the horizontal axis. Three different dimensions are distinguished in order to classify relationships between models:

- horizontal dimension (internal dimension)
- vertical dimension (activity dimension)
- evolution dimension (iteration dimension)

The horizontal dimension deals with relations between submodels that coexist consistently making up a more complex model. The UML incorporates several sublanguages, each one allowing a specific view on the system. Models of different viewpoints have a certain overlap, for instance, an analysis model consists of sequence diagrams and

Figure 1. Dimensions in the development process



collaboration diagrams both representing different aspects of the behavior of the system.

The vertical dimension considers relations between models belonging to the same iteration in different activities (e.g., a design model realizing an analysis model). Two related models represent the same information but at different levels of abstraction.

The evolution dimension considers relations between artifacts belonging to the same activity in different iterations (e.g., a use case is extended by another use case). In this dimension, new models are built or derived from previous models by adding new information that was not considered before or by modifying or detailing previous information.

An essential element to the success of the software development process is the support offered by case tools. Existing case tools facilitate the construction and manipulation of models, but in general, they do not provide checks of consistency between models along either vertical or evolution dimension. Tools neither provide automated evolution of models (i.e., propagation of changes when a model evolves, to its dependent models). The weakness of tools is mainly due to the lack of a general underlying formal foundation for the software development process (particularly focused on relations between models).

To overcome this problem, we propose to apply the well-known mathematical concept of contract to the specification of software development processes by introducing the concept of software process contract (sp-contract). Sp-contracts introduce precision of specification, avoiding ambiguities and inconsistencies, and enabling developers to reason about the correctness of their joint activities. The goal of the proposed formalism is to provide foundations for case tools assisting software engineers during the development process. Sp-contracts provide a formalization of software artifacts and their relationships. They clearly specify pre- and post-conditions for each software development task, allowing for the verification of consistency between models through evolution.

The remainder of this chapter is organized as follows. First, we describe the underlying formalism we use to develop our proposal. Then, we introduce the concept of software process contract (sp-contract), which constitutes our proposal to improve formality of software development process. The next section contains some ideas about future trends and the construction of a case tool based on sp-contracts. Finally, we present the conclusion and related works.

Background: The Notion of Software Contract

Generally, a computation can 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. 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.

The notion of contract regulating the behavior of a software system has been investigated by several authors (Andrade & Fiadeiro, 1999; Back, Petre, & Porres Paltor, 1999; Helm, Holland, & Gangopadhyay, 1990; Meyer, 1992; Meyer, 1997). In particular, in our work, we apply the formalism of contracts proposed by Ralf Back which is based on the Refinement Calculus (Back & von Wright, 1998).

The refinement calculus is a logical framework for reasoning about programs. It is concerned with two main questions: Is a program correct with respect to a given specification? And, how can we improve, or refine, a program while preserving its correctness? Both programs and specifications can be seen as special cases of a more general notion, that of a contract between independent agents. Refinement is defined as an ordering relation between contracts. Correctness is a special case of refinement where a specification is refined by a program.

The refinement calculus is formalized within higher order logic, allowing us to prove the correctness of contracts and to calculate contracts refinements in a rigorous, mathematically precise manner. The refinement calculus is equipped with automatic tools: the Mechanised Reasoning Group led by Joakim von Wright has developed a system for supporting program derivation, precondition calculation and correctness calculation within the Refinement Calculus framework that is called the Refinement Calculator (Butler, Grundy, Langbacka, Ruksenas, & Von Wright, 1997; Celiku & von Right, 2002). This system is based on the HOL theorem prover.

Contract Language

Consider a collection of agents, where each agent has the capability to change the world in various ways through its actions and can choose different courses of action. The behavior of agents and their cooperation is regulated by contracts. The contract can stipulate that the agent must carry out actions in a specific order. This is written as a sequential statement $S_1;...;S_m$, where $S_1,...,S_m$ are the individual actions that the agent has to carry out. A contract may also require the agent to choose one of the alternative actions S1,...Sm. The choice is written as the alternative statement S1È...ÈSm.

The world is described as a state s. The state space S is the set of all possible states s. 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. An agent changes the state by applying a function f to the present state s, yielding a new state f.s. These functions mapping states to states are the most primitive form of action that agents can carry out. An example of state transformer is the assignment x:=exp, that updates the value of attribute x to the value of the expression exp.

The language for contracts is simple:

$$S ::= \langle f \rangle \mid \textbf{if } p \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi} \mid S_1 \textbf{ ; } S_2 \mid \textbf{assert}_a\ p \mid \textbf{R}_a \mid S_1 \cup_a S_2 \mid \textbf{rec } X \bullet S$$

Here a stands for an agent while f stands for a state transformer, p for a state predicate (i.e., a boolean function p:Σ→Bool) and R for a state relation (i.e., relation R:Σ→ Σ→ Bool relates a state σ to a state σ´ whenever R.σ.σ´ holds).

Each statement in this language describes a contract for an agent. Intuitively, a contract is executed as follows:

The functional update <f> changes the state according to the state transformer f, that is, if the initial state is σ_0 then the final state is $f.\sigma_0$. 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.

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 carried out first, 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 nondeterminism into the language of contracts. Both are indexed by an agent which is responsible for deciding how the nondeterminism is resolved. The relational update Ra requires the agent a to choose a final state σ´ so that R.σ.σ´ is satisfied, where ?σ is the initial state. In practice, the relation is expressed as a relational assignment. For example, updatea {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 $S_1 \cup_a S_2$ 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. It also 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) where skip is the well-known “do nothing” statement.

Cooperation Contract

Consider a set of agents that work on the same state independently of each other. Each agent has a will of its own and makes decisions for itself. If these agents want to cooperate, they need a contract that stipulates their respective obligations.

A typical situation is that one of the agents acts as a server, and the other as clients. Assume that a client follows contract S,

Contract S = (f1 ∪ skip) ; T ; f2

Where f1 and f2 are primitive actions and T is the contract for the server,

Contract T = f3 ∪ f4

The occurrence of T in the contract statement S signals that the client asks the server to carry out its contract T.

We can combine the two statements S and T into a single contract statement regulating the behavior of both agents. The combined contract is described by

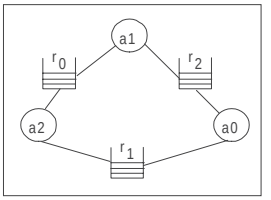
Contract V = (f1 $\cup_{client}$ skip) ; (f3 $\cup_{server}$ f4) ; f2

The combined collaborative contract is the result of substituting the contract statement T for the invocation on T in the contract S and explicitly indicating for each choice which agent is responsible for it.

Another form of interaction between agents occurs when they need to synchronize their individual actions. Assume that the agents (a_0, a_1, a_2) are placed in a ring, with a collection of resources situated between them (r_i is the collection of resources placed between agents a_{i-1} and a_{i+1}, where modulo-3 arithmetic is used). This situation is illustrated in Figure 2.

Each agent a_i has access to the resource in r_{i-1} and r_{i+1} but not to r_i . Resources are nonrenewable, and we assume that the agents take turns grabbing one of them from either

Figure 2. Resource game



side (left or right). An agent also can choose to do nothing. Initially, no agents have grabbed any resource, and the resources are evenly distributed,

$init = n_0,n_1,n_2,r_0,r_1,r_2 := 0,0,0,m,m,m$

where n_i models the number of resources that agent a_i has grabbed and m is the initial number of resources at each point.

The alternatives open to agent a_i are described by the following contract:

Contract $S_i = grabl_i \cup_{ai} skip \cup_{ai} grabr_i$

Where

$grabl_i = assert_{ai}\ r_{i-1} > 0\ ;\ n_i\ ,\ r_{i-1} := n_i + 1,\ r_{i-1} - 1$

$grabr_i = assert_{ai}\ r_{i+1} > 0\ ;\ n_i\ ,\ r_{i+1} := n_i + 1,\ r_{i+1} - 1$

Now the whole system can be described as the combination of subcontract S_0, S_1 and S_2 into a single contract statement that regulates the behavior of the three agents, as follows:

Contract System = init; while $r_o + r_1 + r_2 > 0$ do $S_0\ ;\ S_1\ ;\ S_2$ od

According to this contract, on every round the order of choices is deterministic: agent a_0 chooses first, then a_1 and finally a_2. It is possible to write a different contract permitting a different order of choices.

Semantics of Contracts: The Rules of a Game

Agents try to achieve their goals, that is, to reach a new, more desirable state. The desired states are described by giving condition that they have to satisfy (the post-condition). The possibility of an agent to achieve such a desired state depends on the functions that it can use to change the state.

Given a contract for a single agent and a desired post-condition, we can ask whether the agent following the contract can establish the post-condition. This will depend on the initial state, the state in which the agent starts to carry out the contract. For instance, consider the contract

Contract $S = x:=x+1 \cup x:=x+2$

The agent can establish the post-condition x=2 if x=1 or x=0 initially. When x=1 initially, the agent should choose the first alternative; but when x=0, the agent should choose the second alternative. But the agent cannot achieve its goal from any initial state satisfying either $x>1$ or $x<0$.

On the other hand, an agent cannot be required to follow a contract if the assumptions that it makes are violated for non-allied agents. Violating the assumptions releases the agent from the contract.

The main concern with a contract is to determine whether a set of agents, say A, can use the contract to achieve the stated goals. In this sense, agents have to make the right choices in their cooperation with other agents, which are pursuing different goals that need not be in agreement with their goals. The other agents need not to be hostile; they just have different priorities and are free to make their own choices. However, because no one agent in A can influence the other agents in any way, they have to be prepared for the worst and consider the other agent as hostile. 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.

As far as analyzing what can be achieved with a contract, it is justified to consider the agents involved as the opponents in a game. The actions that the agents can take are the moves in the game. The rules of the game are expressed by the contract; it states what moves the opponents can take and when.

A player in the game is said to have a winning strategy in a certain initial state if the player can win (by doing the right moves) no matter what the opponents do.

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 satisfying q. Satisfaction of a contract (denoted by $\sigma\{S\}q$) corresponds to the existence of a winning strategy. It means that $\sigma\{S_A\}q$ holds if and only if the set of agents has a winning strategy to reach the goal q when playing with the rules S, when the initial state of the game is σ.

If some of the agents in A are forced to breach an assertion, then the coalition loses the game. If the opponents are forced to breach an assertion, they lose the game, and the coalition wins. In this way, an agent can win the game either by reaching a final state that satisfies the post-condition or by forcing the opponents to breach an assertion.

This notion of satisfaction is precisely defined, in the following way: The predicate transformer $\mathbf{wp}_A.S$ maps post-condition 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, $\mathbf{wp}_A.S.q$ is the weakest precondition that guarantees that the agents in A can cooperate to achieve post-condition q. This means that a contract S for a coalition A is mathematically seen as an element (denoted by $\mathbf{wp}_A.S$) of the domain $\mathrm{P}\Sigma \rightarrow \mathrm{P}\Sigma$. Then, the satisfaction of contracts is captured naturally by the notion of weakest precondition, as follows: $\sigma\{S\}q \equiv \mathbf{wp}_A.S.q.\sigma$

The definition of the predicate transformer is as follows. See Back & von Wright (1998) for a more detailed explanation:

(i) $\mathbf{wp}_A.\langle f\rangle.q = (\lambda\sigma.q.(f.\sigma))$

(ii) $\mathbf{wp}_A.(\text{if p then } S_1 \text{ else } S_2 \text{ fi}).q = (p \cap \mathbf{wp}_A.S_1.q) \cup (\neg p \cap \mathbf{wp}_A.S_2.q)$

(iii) $\mathbf{wp}_A.(S_1;S_2).q = \mathbf{wp}_A.S_1.(\mathbf{wp}_A.S_2.q)$

(iv) $\mathbf{wp}_A.(\text{assert}_a\ p).q = \lambda\sigma.(p.\sigma \wedge q.\sigma)$, if $a \in A$

$\lambda\sigma.(\neg p.\sigma \vee q.\sigma)$, if $a \notin A$

(v) $\mathbf{wp}_A.R_a.q = \lambda\sigma.\exists\sigma' \bullet R.\sigma.\sigma' \wedge q.\sigma'$, if $a \in A$

$\lambda\sigma.\forall\sigma' \bullet R.\sigma.\sigma' \rightarrow q.\sigma'$, if $a \notin A$

(vi) $\mathbf{wp}_A.(S_1 \cup_a S_2).q = \mathbf{wp}_A.S_1.q \cup \mathbf{wp}_A.S_2.q$, if $a \in A$

$\mathbf{wp}_A.S_1.q \cap \mathbf{wp}_A.S_2.q$, if $a \notin A$

The Notion of a Software Process Contract

While the notion of a 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 no explicit contract establishing obligations and benefits of members of the development team. At best, the development process is specified by either a graph of tasks or object-oriented diagrams in a semi-formal style, while in most 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 to compare the capabilities of various groupings of agents (coalitions) in order to accomplish a particular goal.

We propose to apply the notion of a formal contract described in the previous section, 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) carrying out actions with the goal of building a software system that meets the user requirements. The software development process consists of a collection of interacting activities. When specifying a specific activity, we may consider the other activities to be controlled by other agents. We may need some of these activities in order to carry out the set of tasks of our activity, but we cannot influence the choices made by the other agents. This situation is analogous to a contractor using subcontractors.

A specification of an activity is a contract that gives some constraints on the results and effects of the activity but leaves freedom for the agent to decide how the actual behavior is to be realized. For example, a member of the development team, say the agent ai, agrees to take over the task of specifying a method of a given Class by either creating a State machine, or a sequence diagram or a set of pre-and post-conditions,

$$\text{Contract S} = \text{create-SM} \cup_{ai} \text{create-SeqD} \cup_{ai} \text{write-Pre\&Post}$$

The rest of the agents can assume that after ai carries out their contracts, some specification for the method does exists, but they do not know precisely which alternative was chosen; so whatever they want to achieve, it should be achieved no matter which alternative was chosen.

A remarkable difference between traditional software contracts and software process contracts (sp-contracts) is the kind of object constituting a state. While in software contracts, objects in the state represent objects in a software system (e.g., a bank account object in a banking system), in sp-contracts, objects in the state are process artifacts, such as a class diagram or a use case model. But this difference is just conceptual, from the mathematical point of view we can reason about process contracts in the standard way, as if they were software contracts. This view of software process as software is not new, we can go back to the work of Osterweil (1997).

Building sp-Contracts

There are different levels of granularity in which sp-contracts are defined. On the 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 evolutions are non-atomic tasks which are composed by a number of primitive tasks. We start specifying atomic contracts (contracts explaining primitive tasks) which will be the building blocks for non-atomic contracts (i.e., regulations for complex evolution activities).

Primitive sp-contracts

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 by Hruby (1999) and the metamodel defined by the OMG Process Working Group (OMG, 1998), among others. In the object-oriented approach, software artifacts produced during the development process are considered objects with methods and attributes.

A Class is a template used to describe objects with identical behavior. The Refinement Calculus has been applied to the specification of Classes, by giving a syntax for the Class declaration and a formal semantics for object instantiation, message passing, inheritance and substitutability (Back, Mikhajlova, & von Wright, 1997; Back, Mikhajlov, & von Wright , 2000).

A Class is given by the following declaration:

C = **subclass** of P

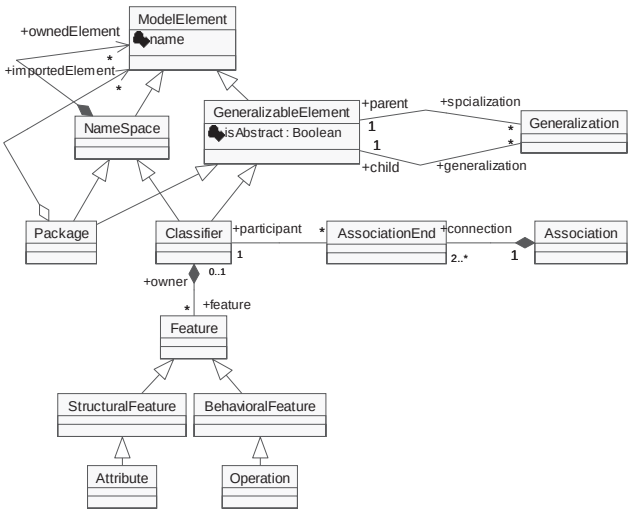
var $attr_1$:Σ1,...,$attr_m$:Σm
C(val x_0:Γ_0)= K,
$Meth_1$(val x_1:Γ_1 , **res** y_1:Δ_1) = $\mathbf{M}_1$,
...
$Meth_n$(val x_1:Γ_1 , **res** y_1:Δ_1) = $\mathbf{M}_n$,

end

This class C describes attributes, specifies the way the objects are created, and gives a (possibly nondeterministic) specification for each method. Class attributes $attr_1$,..,$attr_m$ have the corresponding types Σ_1...Σ_m. The identifier self represents the tuple ($attr_1$,...,$attr_m$). The type of self is $\Sigma = \Sigma_1$ x ... x Σ_m. A class constructor is used to instantiate objects and has the same name as the class. The statement K :$\Gamma_0 \rightarrow \Sigma$ x Γ_0, representing the body of the constructor, introduces the attributes into the state space and initializes them using the input parameter x_0:Γ_0. Methods $Meth_1$... $Meth_n$ specified by bodies M_1 ... M_n operate on the attributes and realize the object functionality. Every statement M_i is of type $(\Sigma \text{ x } \Gamma_i \text{ x } \Delta_i) \rightarrow (\Sigma \text{ x } \Gamma_i \text{ x } \Delta_i)$. The identifier self acts as an implicit result parameter of the constructor and an implicit variable parameter of the methods.

In general, every body M_i includes a precondition p_i and an effect S_i (M_i =*assert* p_i ; S_i). When a method M_i is called there is an agent ***a*** responsible for the call. The method invocation is then interpreted as the following contract: ($assert_a$ p_i ; S_i), that is, 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.

Figure 3. Part of the UML metamodel



For the sake of readability, we write $Meth_i$(**val** $x_i : \Gamma_i$) : Δ_i = M_i[resu/y_n], to denote the declaration $Meth_n$(**val** x_n:Γ_1 , **res** y_n:Δ_n) = M_n, . The result variable y_n is replaced by the special variable called *resu* in the body of the method. And regarding method invocation, we write o.$meth_i$(x_i).$meth_j$(x_j) to indicate that the second method is applied on the object that results from the first invocation, that is to say: o.$meth_i$(x_i, z); z.$meth_j$(x_j).

The library of primitive contracts intentionally reflects the class hierarchy of the UML metamodel (OMG, 2003). Contract library consists of a set of UML artifact's specifications (metaclasses), where each specification describes both the artifact's properties (i.e., attributes of the artifact) and all the possible ways of modifying the artifact (i.e., operations that can be applied on the artifact, such as adding a new feature to a class).

Figure 3 shows a part of the UML metamodel. Primitive contracts for these artifacts are (partially) specified as follows:

Generalization = **subclass of** Relationship

var parent, child : GeneralizableElement,

Constructor Generalization(**val** p,c : GeneralizableElement) = parent:=p; child:=c, parent() : GeneralizableElement = **resu**:=parent,

child() : GeneralizableElement = **resu**:=child,

end

The Class Generalization has an internal state composed by two attributes called parent and child, respectively, both storing a GeneralizableElement. The Class defines a constructor operation and two observer methods, one for each attribute.

NameSpace = **subclass of** ModelElement

var ownedElements : Set of ModelElement,
Constructor NameSpace() = ownedElements:= {} ,
ownedElements() : Set of ModelElement = resu:=ownedElements,
addElement(val e:ModelElement) =
assert (e∉ ownedElements ∧ ∀g• (g∈ ownedElements → e.name ≠ g.name)) ;
ownedElements:= ownedElements ∪ {e} ,
deleteElement(val e:ModelElement) =
assert (e∈ ownedElements) ; ownedElements:= ownedElements - {e} ,

end

GeneralizableElement = **subclass of** ModelElement

var generalizations, specializations : Set of Generalization,
isAbstract: Bool
Constructor GeneralizableElement() = generalizations := {}; specializations := {},

```
                parents() : Set of GeneralizableElement = resu:=
generalizations.collect(parent)1,
               children() : Set of GeneralizableElement = resu:=
specializations.collect(child),
      allParents() : Set of GeneralizableElement = ps := self.parents() ;
                                          resu := ps ∪ ps.collect(allParents),
         isA(val c : GeneralizableElement) : Bool = resu:= (self=c ∨ c ∈
self.allParent()) ,
end
```

The Class GeneralizableElement has an internal state composed by three attributes, the first two attributes containing a set of Generalizations and the third attribute containing a Boolean value. The Class defines a set of methods: method parents() returns a Set consisting of all direct parents of the generalizable element which are accesible through its Generalizations; the method children() returns a set of all direct children; the method allParents() results in a Set containing all ancestors. IsA() returns true if the receiver of the message is a subclass (direct or indirect) of the parameter.

```
Feature = subclass of ModelElement
      var owner : Classifier,
      Constructor Feature (val o : Classifier) = owner:=o,
      owner() : Classifier = resu:=owner,
      setOwner(val o:Classifier) = owner:=o,
end
Classifier = subclass of GeneralizableElement, NameSpace
       var features : Set of Feature,
            associationEnds : Set of AssociationEnd,
      Constructor Classifier() = features :={}; associationEnds :={},
             allFeatures() : Set of Feature = resu:= (features ∪
self.parents.collect(allFeatures) ),
                    associations(): Set of Association = resu:=
self.association.collect(association),
       oppositeAssociationEnds() : Set of AssociationEnd = ...
          .....
       addFeature(val f : Feature) =
            assert (f ∉ features ∧ g • (g ∈ features ∨
                    g ∈ self.oppositeAssociationEnds → f.name ≠ g.name) )
;
            features:= features ∪ {f} ; f.setOwner(self),
                          deleteFeature(val f:Feature) = assert
f∈self.features;self.features:=self.features - {f}
end
```

The Class Classifier has an internal state with two attributes; the first one stores a set of Features while the second one stores a set of AssociationEnds. The Class defines a

set of query methods: the method allFeatures() results in a Set containing all Features of the Classifier itself and all its inherited Features; the operation associations results in a Set containing all Associations of the Classifier itself; the operation oppositeAssociationEnds results in a set of all AssociationEnds that are opposite to the Classifier. Additionally, the Class declares a set of mutator methods which modify the object internal state: the method addFeature() has a precondition stating that the new feature does not belong to the classifier, and the new feature should have a different name from all the other attributes in the classifier, and from all the opposite associationEnds of the classifier. The effect of the method is that the feature is added to the list of features and the classifier is set as the feature's owner.

Package = **subclass of** NameSpace, GeneralizableElement
var importedElements : Set of ModelElement,
Constructor Package() = importedElement:= {} ,
allContents() : Set of ModelElement = resu:=ownedElement ∪ importedElement,
addGeneralization(val g:Generalization) =
assert (g ∉ ownedElements ∧ g.parent ∈ ownedElements ∧
g.child ∈ ownedElements ∧ ¬ g.parent.isA(g.child) ;
self.addElement(g) ,
end

The class Package inherits from NameSpace and GeneralizableElement. It specifies a method addGeneralization() to insert a new generalization in the package. The preconditions for the method are that the generalization is not in the package, all elements connected by the new relationship (i.e., the parent and the child) are included in the package and that the new generalization preserves absence of circular inheritance. The effect of the method is that the new element is added in the collection of owned elements of the package by invoking the method addElement() inherited from the class NameSpace.

Apart from software artifact specifications, the other component in the formalism of sp-contract is the specification of software developers. Software developers are specified by declaring their attributes and the contracts for their activities:

Developer = **subclass of** Object
var name : String , skills : Set of String,
Constructor Developer(val n : String) = name := n ,
.....
end

The class Developer is the root in the hierarchy, it will be subsequently specialized in order to specify concrete behavior of specific developers.

Complex sp-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 $\mathbf{wp}_A$. 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 preconditions.

The **wp** formalism allows us to analyze a single contract from the point of view of different coalitions of agents. If computing the **wp** we obtain 'false,' we can look for a different coalition (e.g., we can permit an outside agent to join the coalition) and compare the results. In other case (if the coalition should be preserved) to achieve the goal the contract have to be modified.

In the following sections, we give examples of complex contract.

Example 1: Contract on the Evolution Dimension

Consider a collaborative activity, in which two software developers have to carry out a refactoring on a class diagram. One of the agents will detect and move all the features that could be pulled up to a superclass, while the other agent will simplify the class diagram by collecting empty classes.

To coordinate this collaborative activity, both agents (e.g., the lifter and the cleaner) subscribe a complex contract that is built on top of primitive contracts establishing the primitive responsibilities for each agent.

The primitive specification for the cleaner agent describes a method called deleteEmptyClass(), as follows:

```
Cleaner = subclass of Developer
        deleteEmptyClass(val p : Package) =
                (update_self c:=c´ | c' ∈ p.ownedElement ∧ c'.features=∅ ∧
                                                c'.children=∅∧ c'.associations=∅) ;
                  p.deleteElement(c);
end
```

The contract for this method states that the agent will detect nondeterministically a class c from the package p given as parameter, such that class c is empty (i.e., it has no children and no features). Then the selected class is deleted from the package.

The primitive specification for the lifter agent contains three methods:

Lifter = **subclass of** Developer
pasteRepeatedFeature(**val** c : Class) =
(**update**$_{self}$ f:=f´ | c´∈ c.children • f´∈ c'.features) ; c.addFeature(f)
,
deleteRepeatedFeature(**val** c : Class) =
(**update**$_{self}$ f:=f´ | c´∈ c.children • f´∈ c'.features) ;
for i=1 **to** (c.children.size) **do** c´:=c.children.at(i); c´.deleteFeature(f)
od,
liftRepeatedFeature(**val** p : Package) =
(**update**$_{self}$ c:=s | s ∈ p.ownedElement ∧
∃f:Feature • (c´∈ s.children • f∈c'.features)) ;
(pasteRepeatedFeature(c); deleteRepeatedFeature(c))
$\cup_{self}$
(deleteRepeatedFeature(c); pasteRepeatedFeature(c)) ;
end

The method pasteRepeatedFeature() says that the agent will receive a class as parameter and will select nondeterministically a feature that appears in all subclasses of the given class. Then, the selected feature is pasted in the class; the method deleteRepeatedFeature() states that the agent will select nondeterministically a feature that appears in all subclasses of a given class. Then, the selected feature is deleted from all the subclasses; finally liftRepeatedFeature() is a more complex method that allows the agent to choose nondeterministically in which order to carry out its activities, after having selected (nondeterministically) a class that is candidate for refactoring).

The complex contract **R** states that both developers (plus a coordinator agent named coord) commit themselves to carry out the refactoring task in a collaborative way. The coordinator agent will nondeterministically choose either asking a_1 to lift a repeated feature or asking a_2 to delete an empty class. The terms of the contract are as follows:

R = a_1 := **new** Lifter ; a_2 := **new** Cleaner ;
while (¬**Q**) **do** a_1.liftRepeatedFeature(p) $\cup_{coord}$
a_2.deleteEmptyClass(p) **od**;

Where **Q** specifies the expected effect of the refactoring activity: (i) there is no repeated feature and (ii) the model does not contain any empty class:

Q = ∀c:Class • c∈p.ownedElement →
(¬∃f:Feature • (∀c´∈c.children • f∈c'.features) ∧ (i)
(c.features≠∅ ∨ c.children≠∅ ∨ c.associations≠∅)) (ii)

We may be interested in calculating the weakest precondition for agents a_1 and a_2 to reach the goal G by using the contract R. That is to say: $\mathbf{wp}_{\{coord,\ a1,\ a2\}}$. R. G

where G = QÙT , being T a formula specifying that the resulting model keeps all the functionality of the original model.

Applying the calculus is possible to determine that if agents a_1 and a_2 work together (i.e., both of them integrate the coalition), then they can reach the goal.

But if a_1 leaves the coalition, the **wp** is false. The achievement of the goal cannot be guaranteed because agent a_1 is free to resolve their nondeterministic choices in a hostile way. For example, in the following choice:

(pasteRepeatedFeature(c) ; deleteRepeatedFeature(c))
$\cup_{a1}$
(deleteRepeatedFeature(c) ; pasteRepeatedFeature(c))

only the first option guarantees the achievement of the goal. If agent a_1 chooses the second option a problem will occur: A feature is deleted before being pasted in the superclass; consequently, the model loses functionality and the final goal cannot be achieved.

Example 2: Contract on the Horizontal Dimension

Arbitrary modifications that do not cause problems when they are applied exclusively, may originate conflicts when they are integrated. Consider a collaborative task in which two agents a_1 and a_2 need to add a generalization relationship respectively to a model, preserving the consistency of the model.

Contract statement **C** specifies that agents a_1 and a_2 will perform their activities sequentially, one after the other:

C = a_1 := **new** Designer ; a_2 := **new** Designer ;
a_1.addGeneralization(p,r) ; a_2.addGeneralization(p,g)

The primitive contract regulating the behavior for Designer states that any designer will accomplish this task by directly invoking the method addGeneralization() of the package artifact:

Designer = **subclass of** Developer
addGeneralization(**val** p : Package , g: Generalization) = p.addGeneralization(g)
end

As we explained before, the method invocation is interpreted as the following contract: ($assert_{a1}$ p_i ; S_i), that is, the agent takes over the responsibility for the preconditions of the method. If agent a_1 (respectively a_2) invokes the method in a state that does not satisfy the precondition, then a_1 breaches the contract.

Using the calculus, 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: $\mathbf{wp}_{\{a1,a2\}}$. C . Q, where C is the contract between agents and Q is the post-condition that specifies the goal of the activity, which is the creation of new generalization relationships guaranteeing the absence of circularity in the class hierarchy:

$$Q = (r \in p.ownedElements \wedge g \in p.ownedElements) \wedge$$
$$\forall c_1,c_2 : GeneralizableElement.\ (c_1.isA(c_2) \wedge c_2.isA(c_1) \rightarrow c_2 = c_1)$$

The weakest precondition P for agents a_1 and a_2 to reach the goal Q by using the contract C, (i.e., P = $\mathbf{wp}_{\{a1,a2\}}$. C. Q) can be semi automatically calculated applying the rules in section 2.3 arriving to the following result:

P =

(i) $r \notin p.ownedElements \wedge r.parent \in p.ownedElements \wedge$
$r.child \in p.ownedElements \wedge \neg\ r.parent.isA(r.child) \wedge$

(ii) $g \notin p.ownedElements \wedge g.parent \in p.ownedElements \wedge$
$g.child \in p.ownedElements \wedge \neg\ g.parent.isA(g.child) \wedge$

(iii) $\forall c_1,c_2 : GeneralizableElement.(c_1.isA(c_2) \wedge c_2.isA(c_1) \rightarrow c_2 = c_1)$

(iv) $\neg\ (g.parent.isA(r.child) \wedge r.parent.isA(g.child))$

Where (i), (ii) and (iii) specify the precondition for applying the first and the second evolution, respectively (as if they were applied in isolation), and (iv) specifies a special requirement to avoid circular inheritance in the case that both evolution actions were applied together.

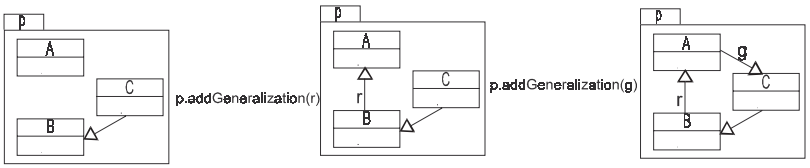
Figure 4 illustrates a conflictive case, in which the expected weakest precondition does not hold in the initial state. As a consequence agents cannot achieve their goals (because a circularity is introduced) in spite of fulfilling the contract.

Future Trends

The sp-contract formalism should be equipped with automatic tools supporting contract derivation, precondition calculation, and correctness calculation. These tools should be connected with the Refinement Calculator (Butler et al., 1997; Celiku & von Right, 2002), which supports the Refinement Calculus (Back & von Wright, 1998).

The need for automatic support is the main motivation for future work. It is necessary to count with a tool to assist developers in the task of writing and applying sp-contracts. This tool should be integrated with an environment for thesoftware development process, providing the following functionality:

Figure 4. Collaborative evolution breaching consistency rule



- Contract edition and storage: Developers can create new contracts and store them in a repository. There are different ways in which a new contract can be created: from scratch as a primitive contract; by specializing an existing contract stored in the repository; or by selecting a group of contracts stored in the repository and composing them to form a new complex contract.
- Pre condition calculus: Given a contract between a set of agents and a specific goal, the tool would be able to compute the weakest precondition for the application of that contract.
- Contract refinement: Specific contracts can be derived from abstract contracts by applying the refinement calculus.
- Contract correctness: If a precondition p and a post-condition q are given and contract S has already been defined, we can prove that S is correct with respect to precondition p and post-condition q.
- Visual assistance: functions of edition of contracts have a textual interface using mathematical notation, but also may have a graphical interface. A contract can be created using a UML editor that both records the operations applied on models (such as adding a new class) and translates them to the mathematical notation. This translation is straightforward using the primitive contracts on the UML artifacts described. On the other hand, the task of selecting a contract from the repository will be assisted by the generation of a graphical view of the contract. This is provided by animating a contract, that is to say, showing how the execution of the contract modifies a given UML model, step by step.

Conclusion and Related Work

During the software development process different UML models are employed to specify the system from different viewpoints at different levels of abstraction. Models of different viewpoints have a certain overlap (Spanoudakis, Frinkelstein, & Till, 1999) and models produced at different levels of abstractions in the development process also are related. Consequently, handling of consistency between models is of major importance (Ghezzi & Nuseibeh, 1999; Kuzniarz, Huzar, Reggio, & Sourrouille, 2002; Kuzniarz, Huzar, Reggio, Sourrouille, & Ataron, 2003).

Different types of consistency problems have been identified; Engels, Küster, Heckel, and Groenewegen (2001) distinguish two dimensions of consistency problems — horizontal and vertical:

- Horizontal consistency concerns specifications consisting of different parts representing the different points of view from which the system is specified.
- Vertical consistency arises when a model is transformed into another refined model. For example a collaboration diagram can be derived from a use case diagram.

However, we need to distinguish three dimensions of consistency (Pons, Giandini, & Baum, 2000): Horizontal, Vertical, and Evolution dimensions (the last two refine Engels' vertical dimension) because two dimensions are insufficient to comprise an iterative and incremental software process where model refinement occurs vertically, inside each iteration; but also horizontally, from one iteration to the next one.

A wide range of different approaches for checking consistency of UML models has been proposed in the literature. Here is an overview of the most relevant works, classified in two groups. The first group focuses on the consistency between a fixed set of artifacts:

Glinz (2000) defines a lightweight approach to consistency between a scenario model and a class model. He assumes semi-formal, loosely coupled models that are complementary: scenarios model the external system behavior, the class model specifies the internal functionality. He achieves consistency by minimizing overlap between the two models and by systematically cross referencing corresponding information. He gives a set of rules (some of them automatically checked) that can be used both for developing a consistent specification and for checking the consistency of a completed specification.

Petriu Sun (2000) analyze the consistency between two different UML sublanguages: Activity diagrams and Sequence Diagrams.

Whittle and Schumann (2000) developed an algorithm for automatically generating statechart designs from a collection of sequence diagrams.

Ehrig and Tsiolakis (2000) investigate the consistency between UML class and sequence diagrams by representing them by attributed graph grammars.

Works in the second group propose a general methodology that can be applied to different consistency problems:

Astesiano and Reggio (2003) look at the consistency problems in the UML in terms of the well-known machinery of classical algebraic specifications. Thus, first they review how the various kinds of consistency problems were formulated in that setting. A similar approach, but using dynamic logic, was defined by Pons and Baum (2000).

Engels et al. (2001) discuss the issue of consistency of behavioral models in the UML and present a general methodology about how consistency problems can be dealt with. According to the methodology, those aspects of the models relevant to the consistency are mapped to a semantic domain in which precise consistency tests can be formulated. The choice of the semantics domain and the definition of consistency conditions vary according to each concrete consistency problem. An instantiation of this approach is the work of Fradet, Le Métayer, and Périn (1999) where systems of linear inequalities are used to check consistency for multiple view software architectures. The general idea is further enhanced in Engels, Heckel, Küster, and Groenewegen (2002) with dynamic metamodeling rules. Model transformation rules are used to represent evolution steps, and their effect on the overall model consistency is explored.

Egyed (2001) presents an approach for automated consistency checking among UML diagrams, called ViewIntegra. The approach makes use of consistent transformation to translate diagrams into interpretations to bring models closer to one another in order to simply comparison.

Grundy, Hosking, and Mugridge (1998) claim that a key requirement for supporting inconsistency management is the facilities for developers to configure when and how inconsistencies are detected, monitored, stored, presented and possibly automatically resolved. They describe their experience with building complex multiple-view software development tools supporting inconsistency management facilities.

Toval and Alemán (2000) formalize the UML notation and transformations between different UML models within rewriting logic. They implement their formalization in the Maude system, focussing on using reflection to represent and support the evolution of models.

Van Der Straeten, Mens, Simmonds, and Jonckers (2003) propose and validate an approach to detect and resolve inconsistencies between different versions of a UML model, specified as a collection of class diagrams, sequence diagrams, and state diagrams. The formalism used is description logic, a decidable fragment of first-order predicate logic. Logic rules are used to detect and to suggest ways to resolve inconsistencies.

The proposal described in this chapter belongs to the second group; sp-contract is a mathematical tool the objective of which is to improve the formality of software development processes. The core of sp-contracts is the formalization of UML software artifacts and their relationships on three dimensions. Sp-contracts handle consistency between models through evolution by specifying state invariant and pre- and post-conditions for each software development task. This feature is closely related to the mechanism of reuse contracts (Steyaert, Lucas, Mens, & D'Hondt, 1996; Mens, ., Lucas, & D'Hondt, 2000). 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.

The originality of sp-contracts resides in the fact that software developers are incorporated into the formalism as agents (or a coalition of agents) who make decisions and have responsibilities (Pons & Baum, 2001; Pons & Baum, 2002). 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 weakest precondition 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. On the other hand, formal refinement techniques can be applied to a contract in order to obtain an improved contract preserving its correctness.

We believe the formalism of sp-contracts can play an important role in the study of software development process: sp-contracts can be useful for reasoning about and justifying good practices in software process, providing a formal rational for them; sp-contracts can provide a means to analyze and reason about refactoring tasks, refinements, and transformation of models.

Regarding scalability issues, when the software development process becomes complex, the formalism allows us to manage the complexity by means of a hierarchical definition and classification of contracts. On the one hand, the library of contracts is organized into a generalization-specialization hierarchy. Then, it is possible to define a new contract by specializing an existing one, by writing only the incremental features. On the other hand, contracts can be specified in a compositional way. It means that complex contracts are built in terms of less complex ones, and weakest preconditions for a complex contract are calculated from weakest preconditions of its constituent contracts. Furthermore, specifications are organized along three different dimensions (horizontal, vertical, and evolution dimension), thus increasing the cohesion and readability of each contract.

References

Andrade, L.F. & Fiadeiro, J.L. (1999). Interconnecting objects via contracts. *Proceedings of The Second International Conference on the Unified Modeling Language. Lecture Notes in Computer Science 1723.* Springer.

Astesiano E. &, Reggio G. (2003). An Algebraic Proposal for Handling UML Consistency, Workshop on Consistency Problems in UML-based Software Development,. Blekinge Institute of Technology Research Report 2003:06.

Back, R., Mikhajlova, A. & von Wright, J. (1997) *Class refinement as semantics of correct subclassing.* Turku Centre for Computer Science. TUCS Technical Report No 147. ISBN 952-12-0114-2. December 1997.

Back, R. & von Wright, J.(1998). *Refinement calculus: a systematic introduction, graduate texts in computer science,* Springer Verlag.

Back, R. Petre L. & Porres Paltor, I. (1999). Analysing UML use cases as contract. *Proceedings of The Second International Conference on the Unified Modeling Language. Lecture Notes in Computer Science 1723.* Springer Verlag.

Back, R., Mikhajlov, L. & von Wright, J. (2000). *Formal semantics of inheritance and object substitutability.* Turku Centre for Computer Science. TUCS Technical Report No 337. ISBN 952-12-0637-3. January 2000.

Butler, M. Grundy, J., Langbacka, T., Ruksenas, R., & Von Wright, J. (1997). The refinement calculator – proof support for program refinement. *Proc. Formal Methods Pacific,* Springer-Verlag.

Celiku, O. & von Right, J. (2002). Theorem prover support for precondition and correctness calculation. *Proc. 4th International Conference on Formal Engineering Methods ICFEM (LNCS 2495),* Springer-Verlag.

Egyed, A. (2001, November). Scalable consistency checking between diagrams – the VIEWINTEGRA approach. *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE),* San Diego.

Ehrig, H. & Tsiolakis, A. (2000). Consistency analysis of UML class and sequence diagrams using attributed graph grammars. In H. Ehrig and G. Taentzer (Eds.), *ETAPS 2000 workshop on graph transformation systems* (pp. 77-86).

Engels G., Küster J., Heckel R., & Groenewegen L. (2001). A methodology for specifying and analyzing consistency of object oriented behavioral models. *Procs. of the IEEE International Conference on Foundation of Software Engineering,* Vienna.

Engels,G., Heckel, R., Küster, J. & Groenewegen, L. (2002). Consistency-preserving model evolution through transformations. In *Proc.of the International Conference on. The Unified Modeling Language. Model Engineering, Concepts, and Tools, number 2460 in Lecture Notes in Computer Science,* (pp. 212–227). Springer-Verlag.

Fradet, P., Le Métayer, D., & Périn, M. (1999). Consistency checking for multiple view software architectures. In *Proc. Int. Conf. ESEC/FSE'99, volume 1687 of Lecture Notes in Computer Science* (pp. 410-428). Springer-Verlag.

Ghezzi, C. & Nuseibeh, B. (1999). *Special Issue on Managing Inconsistency in Software Development (2). IEEE Transaction on Software Engineering, 25*(11).

Glinz, Martin. (2000, June 10-23). A lightweight approach to consistency of scenarios and class models. *Proceedings of the Fourth International Conference on Requirements Engineering*Schaumburg, IL.

Grundy, J.C, Hosking, J.G., & Mugridge, W.B. (1998). Inconsistency management for multiple-view software development environments. *IEEE Transactions on Software Engineering, 24*(11), 960-981.

Helm, R. Holland, I., & Gangopadhyay, D. (1990). Contracts: specifying behavioral compositions in object-oriented systems. *Proceedings of OOPSLA '90.* ACM Press.

Hruby, Pl. (1999). Framework for describing UML compatible development processes. *Proceedings of the Unified Modeling Language Conference. Lecture Notes in Computer Science 1723.* Springer.

Jacobson, I., Booch, G., & Rumbaugh, J. (1999). *The unified software development process,* Addison Wesley.

Kuzniarz, L., Huzar, Z., Reggio, G., and Sourrouille, J., editors (2002) *Proceedings of the first Workshop on "Consistency Problems in UML-Software Development,"* RR-2002-6.

Kuzniarz, L., Huzar, Z., Reggio, G., Sourrouille, J., & Ataron, M., editors (2003) *Proceedings of the 2nd Workshop on "Consistency Problems in UML-Software Development."* Blekinge Institute of Technology Research Report 2003:06.

Mens, T., Lucas, C., & D'Hondt, T. (2000). Automating support for software evolution in UML. *Automated Software Engineering Journal 7,*1, Kluwer Academic Publishers.

Meyer, B. (1992). Design by contract. In *Advances in object oriented software engineering.* Prentice Hall.

Meyer, B.(1997). *Object-oriented software construction, Second Edition,* Prentice Hall.

OMG (1998, July). *Analysis and design process engineering.* Process Working Group, Analysis and Design Platform Task Force..

OMG (2003, March). The unified modeling language specification version1.5, revised by the Object Management Group. Available online at *http://www.omg.org.*

Osterweil, L. (1997). Software processes are software too. Revisited: an invited talk on the most influential paper of ICSE, *Proc. 19th International Conference on Software Engineering.* ACM Press.

Overgaard G. & Palmkvist K.(2000). Interacting subsystems in UML. In *Proceedings of The 3rd. International Conference on the Unified Modeling Language. Lecture Notes in Computer Science.* Spring Verlag.

Petriu, D. & Sun, Y. (2000) Consistent behaviour representation in activity and sequence diagrams. In *Proceedings of The 3rd. International Conference on the Unified Modeling Language. Lecture Notes in Computer Science.* Spring Verlag.

Pons, C. & Baum, G. (2000). Formal foundations of object-oriented modeling notations. *3rd International Conference on Formal Engineering Methods, ICFEM 2000,* York, UK. IEEE Computer Society Press.

Pons, C., Giandini, R., & Baum, G. (2000). Dependency relationships between models through the software development process. *10th International Workshop on Software Specification and Design (IWSSD),* California, IEEE Computer Society Press.

Pons, C. & Baum G. (2001). Software development contracts, *5th European Conf. on Software Maintenance and Reengineering, Special Session on Formal Foundation of Software Evolution.* Portugal.

Pons, C. & Baum G. (2002). Contracts soundness for object oriented software development process. *OOPSLA'2002 Workshop on Behavioral Semantics.* Seattle, WA. Northeastern University, College of Computer Science, 163-177.

Spanoudakis, G., Frinkelstein, A., & Till, D. (1999). Overlaps in requirement engineering. *Automated Software Engineering: An International Journal. 6*(2), 171-198.

Steyaert, P., Lucas, C., Mens, K., & D'Hondt, T. (1996). Reuse contracts: managing the evolution of reusable assets. *Proceedings of OOPSLA'96,* New York, ACM Press.

Toval, A. & Alemán, J. (2000). Formally modeling UML and its evolution: a holistic approach. In S. Smith and C. Talcott (Eds.), *Formal methods for open object-based distributed systems IV,* (pp. 183-206). Kluwer Academic Publishers.

Van Der Straeten, R., Mens, T., Simmonds, J., & Jonckers, V.(2003) Using description logic to maintain consistency between UML-models. *Proc. 6th International Conference on the Unified Modeling Language. Lecture Notes in Computer Science number 2863.* Springer.

Van Gorp, P., Stenten, H., Mens, T., & Demeyer, S. (2003). Towards automating source-consistent UML refactoring. *Proc. 6th International Conference on the Unified Modeling Language. Lecture Notes in Computer Science number 2863.* Springer.

Whittle, J. & Schumann, J. (2000). Generating statechart designs from scenarios. *Proceedings of International Conference on Software Engineering ICSE 2000.* Limerick, Ireland.

Footnotes

[1] The type Set provides the traditional operations: select, reject, collect (or map), size, $\cup$, $\cap$, $-$, $\in$, $\subseteq$.