Generating Operation Contracts from UML Class Diagrams: A Template-based Approach

Jordi Cabot
Estudis d'Informàtica, Multimedia i Telecomunicacions, Universitat Oberta de Catalunya
jcabot@uoc.edu

Cristina Gómez
Dept. Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya
cristina@lsi.upc.edu

Abstract

Class diagrams must be complemented with a set of system operations that describe how the user can modify and evolve the system state. To be useful, such a set must be complete and correct. A manual specification of system operations is an error-prone and time-consuming activity. Therefore, the goal of this paper is to automatically provide a minimal set of operations that verify both properties. Afterwards, these operations may be combined to build more complex ones. Operations are drawn from the elements (classes, attributes, associations …) of the class diagram. We believe that in the context of the MDD and MDA approaches, our proposal represents a new step in the fulfillment of the long-standing goal of automating information systems building.

1. Introduction

The specification of an information system must include all relevant static and dynamic aspects of the domain [4]. The static aspects are collected in structural diagrams. Structural diagrams include the definition of all relevant integrity constraints. In this paper, we assume that structural diagrams are represented by means of UML class diagrams [10] extended with OCL constraints [9].

Dynamic aspects are usually specified by means of a behavioural schema consisting in a set of system operations [6] (also known as domain events [8]) that the user may execute to query and/or modify the information modeled in the class diagram. A *system operation* consists of a non-empty set of elementary modifications over the system state that is perceived by the user of the information system as a single change in the domain.

Behavioural schemas must be complete and correct [7]. A behavioural schema *bs* is complete when, through the system operations in *bs*, the user can perform all kinds of changes (as insertions, updates or deletions) over any modifiable element of the class diagram (i.e. given an element *e* of the class diagram and a possible kind of change *c* over *e*, there is at least an operation in *bs* that includes *c*). It is correct, when, for each operation *op*, there exists at least an initial system state and a set of argument values that ensure a successful execution of *op* (an execution is successful when the new system state is consistent with the integrity constraints of the class diagram). Incomplete behaviour schemas result in information systems with parts that the user cannot modify since no available operations address them. Incorrect behaviour schemas result in information systems with operations that can never be successfully executed.

For instance, given the simple example of Figure 1, we must specify an operation to create new employees, an operation to delete employees and two operations to update the *name* and the *salary* attributes. This behaviour schema is complete since all modifiable elements of the class diagram (except for *dateOfBirth* which is marked as read only) can be created, updated and deleted by means of executing the system operations. Moreover, it is also correct. The deletion operation can be executed in all states with at least an employee instance. The insert and update operations can be applied providing that the argument corresponding to the new salary

value be greater than 600 (this is the only restriction imposed by the *ValidSalary* constraint).

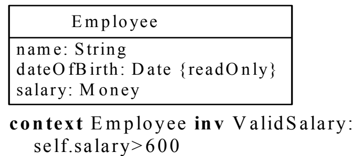


Figure 1. Example of a simple structural schema

For all non-trivial class diagrams the number of required system operations rapidly increases. Therefore, the specification of a complete and correct set of operations becomes an error-prone and time-consuming activity. Given a class diagram, the goal of our paper is to automatically generate a possible behaviour schema that satisfies the completeness and correctness properties. Moreover, among all possible behaviour schemas verifying those properties, we are interested in generating a minimal schema, that is, a behaviour schema where all system operations are necessary to satisfy the completeness property.

Our approach automatically extends a class diagram with a set of system operations (specified with contracts expressed in OCL) by means of applying a predefined set of templates over the given diagram. This generation facilitates the automatic model-driven development of the information system from its initial static specification since frees designers from a manual definition of system operations during the software development process.

As far as we know, ours is the first approach aimed at the automatic generation of a complete and correct behaviour schema. [5] generates the set of possible elementary changes over the classes and associations of a UML class diagram but does not take into account the combination of these elementary changes into system operations. [3] derives a set of operations from an EER diagram but they do not ensure completeness and correctness properties. Alternatively, other approaches try to generate system operations from the information provided in different diagrams as the use case diagram. For instance, [13] presents a method to generate system operations from use cases specifications. Nevertheless, this method is not automatic nor properties of the generated behaviour schema are analyzed. Regarding OCL contracts, [1] provides some primitives to help designers in their manual definition.

The rest of the paper is structured as follows. Section 2 introduces some preliminary concepts. Section 3 presents our template-based generation of the required system operations for a given class diagram. Finally, section 4 shows the results of our case study and section 5 presents some conclusions and further work.

2. Preliminary Concepts

Class Diagrams. We represent a class diagram CD with the tuple:

CD=<CL, ATT, ASS, GEN, IC, modifiable>

where CL, ATT, ASS, GEN and IC represent the set of classes, attributes, associations, generalizations and constraints (graphic as well as OCL textual constraints) of the class diagram CD, respectively. All elements in CD are assumed to be correct instances of the corresponding metaclasses of the UML metamodel (metaclass *Class* for elements in CL, metaclass *Property* for elements in ATT, *Association* for elements in ASS, *Generalization* for elements in GEN and *Constraint* for elements in IC). We assume that all associations are binary associations. N-ary associations and associative classes may be expressed by means of binary associations.

Modifiable is a boolean function that when applied over a model element it indicates if that element is modifiable, that is, if its values or instances can be changed at run-time. The modifiability of a model element depends on the type of the element. A class c ($c \in$ CL) is modifiable as long as c is not an abstract class (i.e. when its *isAbstract* property, defined in the *Class* metaclass evaluates to false) and c is not the supertype of a covering generalization set (in a covering generalization set no instances of the supertype can be directly created). An attribute a ($a \in$ ATT) is modifiable when neither is read only nor it is derived (i.e. *a.isReadOnly=false and a.isDerived=false*, where *isReadOnly* and *isDerived* are properties of the UML *Property* metaclass). An association *as* (*as* $\in$ *ASS*) is modifiable when none of its roles (member ends or association ends in the UML terminology) is read only or derived.

Specification of system operations. There exist two different alternatives to specify system operations. Operations can be specified *imperatively* or *declaratively* [15]. In an imperative specification the designer explicitly defines the set of elementary changes (creations of objects, attribute updates,…) to be applied over the system state during the operation execution. In a declarative specification the designer defines a contract for each operation. The contract consists of a set of *pre* and *postconditions*. A precondition defines a set of conditions on the operation input and the system state that must hold when the operation is invoked. The postcondition states the set of conditions that must be satisfied by the system state at the end of the execution.

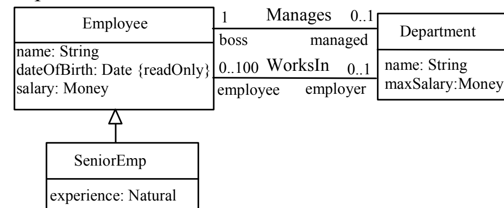
Our proposal may generate both types of specifications but for the sake of simplicity we only provide in this paper the declarative version. Operations are assigned to an appropriate class of the class diagram (according to the *best practices* defined in the *GRASP* patterns [6]).

3. Automatic Generation of System Operations

Given a class diagram CD=<CL, ATT, ASS, GEN, IC, modifiable>, we describe in this section how to automatically generate a set of system operations to evolve the system state. The number and structure of the system operations depend on the structure of CD and on the correct, complete and minimal properties.

Operations are specified declaratively, by means of a contract with a set of pre and postconditions defined in OCL. The contracts do not include the verification of the integrity constraints defined in CD (strict interpretation of operation contracts [11]) in order to avoid redundancies between the contracts and the constraints (this improves the quality of the resulting specifications, see [2]). Therefore, the postcondition of the operation refers solely to the behaviour of the own operation. It is assumed that the operation execution fails (integrity checking approach) when leaves the system in a state that does not satisfy all constraints even if the system state does satisfy the operation postcondition. Only some specific types of constraints (as minimal cardinality constraints) that affect the generation of the system operations are considered.

As an example, consider the class diagram of Figure 2 that represents the relationship between departments and the employees working and managing them and that includes some integrity constraints stating that the name of employees and departments must be unique, that the boss of a department must be a senior employee and that the salary of an employee cannot be greater than the maximum salary defined in his/her department.



context Employee **inv** nameEmployeeUnique:
Employee.allInstances()->isUnique(name)

context Department **inv** nameDepartmentUnique:
Department.allInstances()->isUnique(name)

context Department **inv** bossIsSenior:
self.boss.oclIsTypeOf(SeniorEmp)

context Department **inv** maxSalary:
self.employee->forAll(e| e.salary<=d.maxSalary)

Figure 2. Class diagram used as a running example

Given this example, our method would generate, among others, the system operation *Department::createDepartment(pname:String, pmaxSalary: Money, pboss:Employee)*, aimed at creating a department with name *pname*, maxSalary *pmaxSalary* and with *pboss* as the boss of the department. The operation does not check if *pboss* is a senior employee (i.e. an instance of *SeniorEmp*), as required by the *bossIsSenior* constraint. As we have said before, it is assumed that, when *pboss* is not a senior employee the operation execution will fail. Note that, however, *createDepartment* must be considered as a correct operation since there are plenty of possible execution scenarios for *createDepartment* that leave the system in a consistent state, in particular, all executions where the argument value of *pboss* is a senior employee.

On the other hand, due to the minimal multiplicity of the *boss* role in the *Manages* association, we are forced to include as a parameter of *createDepartment* the employee in

charge of managing the department. Otherwise, *createDeparment* would not be correct (all possible executions would end up in a state that violates the minimal multiplicity of the *boss* role).

In the following (sections 3.1- 3.4), we present a set of templates that allow generating the appropriate system operations contracts for a given class diagram. Templates are grouped depending on the kind of element of the class diagram (class, attribute, association and generalization) they tackle. The variable parts of templates are defined between angle brackets. These variable parts are instantiated with the concrete information about the given class diagram to obtain the final contracts.

3.1. System Operations for Classes

Given a modifiable class $c \in$ CL, a complete behaviour schema must include a system operation to create instances of *c* and another to remove instances from *c*. To satisfy the minimal property, no additional operations should be defined.

To comply with the correctness property these operations are not provided for non-modifiable classes. For example, a superclass in a covering generalization set cannot be directly instantiated (covering sets force all instances of the superclass to be also instances of at least one subclass; a direct creation of an instance of the superclass would violate this constraint). Instead, its instances are created/deleted when creating/deleting instances of their direct or indirect modifiable subclasses.

Operation to Create an Instance of a Class. A system operation to create an instance of a class *c* has a parameter for each non-derived attribute[1] of *c*. The minimum and maximum multiplicity and the type of the parameter coincide with that of the corresponding attribute, except for attributes with a defined default value, where the minimum multiplicity is always zero (these attributes take the default value when the instance creation operation does not initialize the attribute). This is a static operation.

[1] *ReadOnly* attributes can be initialized during the object creation. Afterwards, their value cannot be modified

Additionally, the creation operations need an additional parameter for each modifiable association in which instances of *c* have a mandatory participation (i.e. associations where instances of *c* must be necessarily related with some instance/s of the other participant of the association). Each parameter represents the set of instances that will be linked with the created instance[2]. The minimum and maximum multiplicity of the parameter depends on the multiplicities of the opposite member end of *c* in the association. Note that the absence of these parameters results in incorrect operations because the created instance would always violate the minimum cardinality constraints of the problematic member ends.

The goal of this operation is to create a new instance of *c*, initialize the non-derived attributes of the new instance and create the links between this new instance and the instances of related classes passed as argument values of the operation. In what follows, we present a template to automatically generate this kind of system operations.

Template 1: System Operation to Create an Instance of a Class. Given a class diagram CD=<CL, ATT, ASS, GEN, IC, modifiable> and a class $c \in$ CL such that *c*.modifiable=true, the system operation to create instances of *c* is:

```
context <c> :: create<c>
(p<at_1>[<at_1.lower>..<at_1.upper>]:<at_1.datatype>
,...,
p<at_i>[<at_i.lower>..<at_i.upper>]:<at_i.datatype>,
p<at_i+1>[0..<at_i+1.upper>]:<at_i+1.datatype> ,...,
p<at_n>[0..<at_n.upper>]: <at_n.datatype>,
p<me_1>[<me_1.lower>..<me_1.upper>]:<p_1>,…,
p<me_m>[<me_m.lower>..<me_m.upper>]:<p_m>)
post: self.oclIsNew() and self.oclIsTypeOf(<c>)
and self.<at_1>=p<at_1> and … and
```

[2] Populating an empty system state when both participants *p1* and *p2* of an association *as* present a minimum multiplicity > 0 require a slightly different version of this operation since the related instance can neither exist (there is a cycle of dependencies, creating *p1* requires the previous existence of some instance/s of *p2* and the other way around). To deal with this initial situation we could provide an operation to create at the same time both participants and the link between them.

self.<at$_i$>=p<at$_i$> and if p<at$_{i+1}$>->size()>0 then self.<at$_{i+1}$>=p<at$_{i+1}$> endif and … and if p<at$_n$>->size()>0 then self.<at$_n$>=p<at$_n$> endif and self.<me$_1$>=p<me$_1$> and … and self.<me$_m$>=p<me$_m$>

where $pat_1,...,pat_n$ are the parameters corresponding to non-derived attributes of *c* (among them, $pat_{i+1},\ldots,pat_n$ represent the attributes with a default value) and $pme_1,...,pme_m$ the parameters corresponding to the non-derived member ends where *c* has a mandatory participation ($p_1,...,p_m$ are the corresponding participant classes). In both cases, we must consider not only the attributes and associations of *c* but also all attributes and associations of the superclasses (direct or indirect) of *c*, if any.

The application of the previous template to the running example of Figure 2 generates three system operations: *Department::createDepartment(pname:String, pmaxSalary:Money, pboss:Employee), Employee::createEmployee(pname:String, pdateOfBirth:Date, psalary:Money)* and *SeniorEmp::createSeniorEmp(pname:String, pdateOfBirth:Date, psalary:Money, pexperience:Natural)*.

As an example, the resulting contract for the *createDepartment* operation is the following:

context Department :: createDepartment (pname:String, pmaxSalary:Money, pboss:Employee)
post: self.oclIsNew() and self.oclIsTypeOf(Department) and self.name=pname and self.maxSalary=pmaxSalary and self.boss=pboss

Operation to Delete an Instance from a Class. Similarly to the creation operation, the system operation to remove from the system an instance of a class *c* is only defined when *c* is modifiable. No parameters are necessary for this operation.

Template 2: System Operation to Delete an Instance from a Class. Given a class diagram CD=<CL, ATT, ASS, GEN, IC, modifiable> and a class $c \in$ CL such that *c*.modifiable=true, the system operation to delete instances of *c* is:

context <c>::delete()
post: not self.oclIsKindOf(OclAny)

where the satisfaction of the postcondition *not self.oclIsKindOf(OclAny)* ensures that the object has been deleted from the system since this expression only holds when *self* is not an instance of *OclAny* (which is the common supertype for all classes in CL). We assume that the removal of an instance implicitly causes the removal of all links where the objects participated [14].

The application of the previous template over the classes in Figure 2 generates three system operations to delete instances of *Department*, *Employee* and *SeniorEmp*.

3.2. System Operations for Associations

For modifiable associations $as \in$ ASS, two basic system operations are required, one to create new links of the association and another to delete existing links[3].

Also, an alternative update operation is required when a member end of *as* presents a maximum multiplicity value equals to its minimum multiplicity value. As an example consider the class diagram of Figure 3. Given a consistent system state (i.e. a state where all *B* instances are related with exactly one *A* instance), we cannot create new links of *R* (nor delete them) without violating the multiplicity constraints. Creating a new link would cause some *B* instance to be related with two *A* instances. Similarly, deleting a link from *R* would cause some *B* instance to not be related with any *A* instance. In this case, creation and deletion of links is only possible when creating/deleting *A* and/or *B* instances at the same time. Therefore, to relate an instance i_b of type *B* with an instance i_a of type *A* we cannot, first, delete the old link between i_b and a previous *A* instance, and, after, create the new link between i_b and i_a. We must perform both changes within the same system operation. For this purpose we provide an update association operation that permits to change the links of *as* while satisfying the multiplicity constraints.

---

[3] UML 2.0 does not clearly states whether it is possible to create new links of associations with read only member ends. In UML 1.4, there was a distinction between *read only* (no creations allowed) and *add only* associations (creations are permitted but not updates nor deletions)

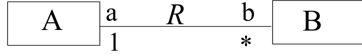


Figure 3. Example of a class diagram

In UML, operations cannot be assigned to associations. Therefore, operations for managing associations must be assigned to the association participants. To satisfy the completeness property, it is enough to add the operations over one of the association participants. However, on behalf of the usability of the generated behaviour schema, we may relax here the minimal restriction and add the operations to both participants.

Operation to Create a Link of an Association. A system operation to create a new link of an association needs as parameters the two instances to be linked. One of the parameters is implicit (i.e. it is the instance over which the system operation is executed). The type of the other parameter is the type of the other participant in the association.

The operation has a precondition that ensures that no other link with the same participants exists (this precondition is necessary unless a member end of the association has the property *isUnique* set to false). Note that this constraint is not an explicit constraint of the class diagram but a constraint required by the semantics of the own modeling language. The postcondition ensures that the new link has been created. The following template is used to generate this type of system operations:

Template 3: System Operation to Create a Link of an Association. Given a class diagram CD=<CL, ATT, ASS, GEN, IC, modifiable> and an association $as \in$ ASS with its corresponding member ends me_1 and me_2, and such that *as*.modifiable=true, the system operation to create new links of *as* is:

context <c_1>::create<me_2>Link(p<me_2>:<c_2>)
pre: self.<me_2>->excludes(p<me_2>)
post: self.<me_2>->includes(p<me_2>)

where c_1 is the participant class next to me_1 ($me_1.class=c_1$) and c_2 the participant next to me_2 ($me_2.class=c_2$).
The application of the previous template to the running example of Figure 2 generates the following system operations: *Employee::createEmployerLink(pemployer:Department)* (and/or *Department::createEmployeeLink(pemployee:Employee)*). Note that operations to create new links for the *Manages* association are not provided due to the multiplicity constraints of the *boss* member end.
As an example, the contract for the *Department::createEmployeeLink* operation is:

context Department::createEmployeeLink (pemployee:Employee)
pre: self.employee->excludes(pemployee)
post: self.employee->includes(pemployee)

Operation to Delete a Link from an Association. Similarly to the previous operation, a system operation that deletes a link needs to know both participants of the link to be deleted.

Template 4: System Operation to Delete a Link from an Association. Given a class diagram CD=<CL, ATT, ASS, GEN, IC, modifiable> and an association $as \in$ ASS with its corresponding member ends me_1 and me_2 and such that *as*.modifiable=true, the system operation to remove a link of *as* is:

context <c_1>::delete<me_2>Link(p<me_2>:<c_2>)
post: self.<me_2>->excludes(p<me_2>)

where c_1 is the participant class next to me_1 and c_2 the participant next to me_2.
For the running example of Figure.3 the following system operation is generated: *Employee::deleteEmployerLink(pemployer:Department)* (and/or *Department::deleteEmployeeLink(pemployee:Employee)*).

Operation to Update a Link of an Association. A system operation to update existing links of an association *as* between the participants p_1 and p_2 receives as a parameter the instance i of the participant p_1 to be updated and the set of instances set_i of p_2 that must be related with i. Then, the operation replaces previous links among and related p_2 instances with the new links among i and the instances of p_2 in set_i. The minimum and maximum cardinality of set_i depends on the multiplicities defined in the member end next to p_2.

Template 5: System Operation to Update a Link of an Association. Given a class diagram CD=<CL, ATT, ASS, GEN, IC, modifiable> and an association *as* ∈ ASS with its corresponding member ends me_1 and me_2 and such that *as*.modifiable=true, the system operation to update links of *as* is:

context < c_1>::update<me_2>Link
(p<me_2>[<me_2.lower>..<me_2.upper>]:<c_2>)
post: self.<me_2> = p<me_2>

where c_1 is the participant class next to me_1 and c_2 the participant next to me_2.

For the example of Figure 2 the system operations *Department::updateBossLink(pboss:Employee)* (and/or *Employee::updateManagedLink(pmanaged[0..1]: Department)*) and *Department::updateEmployeeLink(pemployee[0.. 100]: Employeee)* (and/or *Employee::updateEmployerLink(pemployer[0..1]: Department)* are generated.

3.3. System Operations for Attributes

For each modifiable attributes *at* ∈ ATT, a system operation to update its value/s is required. This operation is assigned to the class owning the attribute. The new value/s is the only parameter of the operation. The multiplicity and type of the parameter coincides with that of the corresponding attribute. The postcondition of the operation ensures that the new value/s has been assigned to the attribute.

Template 6: System Operation to Update the Values of an Attribute. Given a class diagram CD=<CL, ATT, ASS, GEN, IC, modifiable> and an attribute *at* ∈ ATT such that *at*.modifiable=true, the system operation to update the attribute *at* of a class *c* is:

context <c>::update<at>
(p<at>[<at.lower>..<at.upper>]:<at.datatype>)
post: self.<at>= p<at>

The application of the this template to the running example of Figure 2 generates the following system operations: *Employee::updateName(pname:String), Employee::updateSalary(psalary:Money), Department::updateName(pname:String), Department::updateMaxSalary(pmaxSalary:Money)* and *SeniorEmp::updateExperience(pexperience: Natural)*.

Alternatively, we could provide a template to generate a single system operation to update at the same time all attributes of a class (with a parameter for each attribute; all parameters allowing a minimum multiplicity equal to zero). This operation can be regarded as a composition of the application of the previous template over each attribute.

3.4. System Operations for Generalizations

For each modifiable class *c* superclass of a generalization *g* ∈ GEN, a complete behaviour schema must include a system operation to generalize instances of a subclass *sbc* of *c* to *c* (that is an operation that, given an instance *i* of *sbc,* removes *i* from *sbc* but leaves *i* as instance of *c*). To generate a correct schema, we cannot provide this operation for non-modifiable superclasses since then we may end up with an instance *i* of *c* that it is not, at the same time, instance of one of the subclasses of *c*, violating this way the covering constraint of the generalization set or the is*Abstract* property of *c*.

Moreover, for each class *c* superclass of a generalization *g* ∈ GEN, we also need to provide an operation to specialize instances of *c* to subclasses of *c* (that is, an operation that given an instance *i* of *c* and a subclass *sbc* of *c*, adds *i* to the set of instances of *sbc*). Note that this operation is required regardless the modifiability of *c*. When *c* is not modifiable we know that all instances of *c* are also instances of a subclass *sbc* of *c*, but this does not forbid specializing *c* instances to a different subclass as well. The only exception is when *c* is not modifiable and the generalization set is *disjoint*, since then, instances of *c* cannot be instance of more than one subclass at the same time. In that case, an instance *i* must change from a subclass sbc_1 to another subclass sbc_2. Similarly to the update operation for links, we require an operation to change an instance from a subclass to another. This operation first generalizes *i* from sbc_1 to the superclass and then specializes *i* from the superclass to sbc_2.

A class may have several specialize/generalize operations if it participates in different generalizations. The *specialize* operation is attached to the superclass of the generalization while a *generalize* operation is attached to each subclass. Both operations are similar to the *create* and *delete* operations for classes. The main difference is that now the attributes and associations we must consider are just those attributes and association specifics of the subclass.

Operation to Specialize an Instance of a Class. The specialization operation has a parameter for each specific non-derived attribute of the subclass (the attributes defined in the superclass already have a value) as well as a parameter for each specific non-derived association of the subclass where the subclass has a mandatory participation. The operation precondition must check that the instance to be specialized is not yet instance of the subclass.

The template for the specialization operation is the following:

Template 7 System Operation to Specialize an Instance of a Class. Given a class diagram CD=<CL, ATT, ASS, GEN, IC, modifiable> and a generalization $g \in$ GEN relating a superclass *sp* with a subclass *sb*, the system operation to specialize an instance of *sp* into *sb* is:

context <sp>::specializeTo<sb>
(p<at_1>[<at_1.lower>..<at_1.upper>]:<at_1.datatype>
,...,
p<at_i>[<at_i.lower>..<at_i.upper>]: <at_i.datatype>,
p<at_{i+1}>[0..<at_{i+1}.upper>]:<at_{i+1}.datatype> ,...,
p<at_n>[0..<at_n.upper>]: <at_n.datatype>,
p<me_1>[<me_1.lower>..<me_1.upper>]:<p_1>,…,
p<me_m>[<me_m.lower>..<me_m.upper>]:<p_m>)
pre: not self.oclIsTypeOf(<sb>)
post: self.oclIsTypeOf(<sb>) and
self.<at_1>=p<at_1> and … and self.<at_i>=p<at_i>
and if p<at_{i+1}>->size()>0 then
self.<at_{i+1}>=p<at_{i+1}> endif and … and
if p<at_n>->size()>0
then self.<at_n>=p<at_n> endif and
self.<me_1>=p<me_1> and … and
self.<me_m>=p<me_m>

Note that in the postcondition we do not require the instance to be new, we just enforce that the instance becomes an instance of the subclass. The semantic of the different parameters of the operation is the same as in template 1.

Given the class diagram of Figure 2, only the system operation *Employee::specializeToSeniorEmp (pexperience:Natural)* is generated when applying this template. Its contract is:

context Employee::specializeToSeniorEmp
(pexperience:Natural)
pre: not self.oclIsTypeOf(SeniorEmp)
post: self.oclIsTypeOf(SeniorEmp) and
self.experience=pexperience

Operation to Generalize an Instance of a Class. The generalization operation does not have parameters. The postcondition ensures that the generalized instance is no longer instance of the subclass but remains as instance of its superclass.

Template 8: System Operation to Generalize an Instance of a Class. Given a class diagram CD=<CL, ATT, ASS, GEN, IC, modifiable> and a generalization $g \in$ GEN relating a modifiable superclass *sp* with a subclass *sb*, the system operation to generalize an instance of *sb* into *sp* is:

context <sb>::generalizeTo<sp>()
post: self.oclIsKindOf(<sp>) and not
self.oclIsTypeOf(<sb>)

For the example of Figure 2, only the system operation *SeniorEmp::generalizeToEmployee()* is generated.

4. Case study

To show the benefits of our proposal, in this section we compare our generated behaviour schema for a real-life application with the behaviour schema originally specified by the designer for the same application by hand.

In particular, we have analyzed a system for a Conference Management Application (CMA) as specified in [12]. This system provides functionalities to support paper submissions, assignment of papers to reviewers and the evaluation process. The class diagram consists of 13 classes, 13 binary association, 2 non-covering generalization sets and several constraints. The proposed behaviour schema includes 29 system operations.

The application of our templates (templates 1 to 6) to the CMA completely generates 13 of the 29 system operations. Seven additional operations (each one assigning one or more constant values to attributes of the class diagram) can be directly mapped to our templates by passing these constant values as parameters of the operations generated with template 6. The rest of system operations, 9 of 29, are only partially generated by our method. This means that the designer must manually complete their specification. Mainly, the difference is that in the original schema these nine operations include some ad hoc if-else conditions that restrict the applicability of the operations depending on the system state. Clearly, it is not possible to automatically generate these conditions.

From the results presented above, we see that the application of our templates helps designers by reducing in a 69% (20 of 29) the number of system operations to be defined and by providing at least an initial contract for the rest. Moreover, our approach generates some operations that did not appear in the manually specified schema (for instance, all operations corresponding to templates 7 and 8 and some corresponding to template 6). Designers could use this information to detect whether some of the required system operations are missing or the specification of the class diagram is incomplete.

5. Conclusions and further work

The complete definition of the behaviour schema of an information system is one of the most important tasks in the analysis and design stages. The method we have presented in this paper facilitates this task by means of automatically generating an initial set of system operations. The operations are drawn from the structure and constraints of the class diagram. Our approach could be easily integrated into current UML CASE tools.

The correctness and completeness properties of this set of operations guarantee the quality of the behaviour schema. Designers are free to directly use our operations (avoiding the manual definition of the behaviour schema) or to combine them in order to create more complex operations (while keeping the previous quality properties).

As a further work we would like to study how we can use the information of use case and state diagrams to automatically generate more complex system operations (combinations of the ones presented herein).

Acknowledgments

We would like to thank the people of the GMC group for their many useful comments in the preparation of this paper. This work has been partially supported by the Ministerio de Ciencia y Tecnología under project TIN2005-06053

References

[1] Ackermann, J., Turowski, K.: A Library of OCL Specification Patterns for Behavioural Specification of Software Components. CAiSE'06, LNCS, 4001 (2006) 255-269

[2] Costal, D., Sancho, M.-R., Teniente, E.: Understanding redundancy in UML models for object-oriented analysis. CAiSE'02, LNCS, 2348 (2002) 659-674

[3] Engels, G., Gogolla, M., Hohenstein, U., Hüllmann, K., Löhr-Richter, P., Saake, G., Ehrich, H.-D.: Conceptual Modelling of Database Applications Using an Extended ER Model. DKE 9 (1992) 157-204

[4] ISO/TC97/SC5/WG3: Concepts and Terminology for the Conceptual Schema and Information Base. ISO, (1982)

[5] Laleau, R., Polack, F.: Specification of Integrity-Preserving Operations in Information Systems by Using a Formal UML-based Language. Information and Software Technology 43 (2001) 693-704

[6] Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process. 2nd edn. Prentice Hall (2001)

[7] Olivé, A.: Conceptual Modeling of Information Systems. Springer Verlag (2007), in press

[8] Olivé, A., Raventós, R.: Modeling events as entities in object-oriented conceptual modeling languages. DKE 58 (2006) 243-262

[9] OMG: UML 2.0 OCL Specification. OMG Adopted Specification

[10] OMG: UML 2.0 Superstructure Specification. OMG Adopted Specification
[11] Queralt, A., Teniente, E.: Specifying the Semantics of Operation Contracts in Conceptual Modeling. Journal on Data Semantics 7 (2006) 33-56
[12] Raventós, R.: A conceptual schema for a conference management application. Technical Report, LSI-05-1-R (2005)
[13] Sendall, S., Strohmeier, A.: From use cases to system operation specifications. UML'00, LNCS, 1939 (2000) 1-15
[14] Sendall, S., Strohmeier, A.: Using OCL and UML to Specify System Behavior. In: Object Modeling with the OCL, The Rationale behind the Object Constraint Language. Springer-Verlag (2002) 250-280
[15] Wieringa, R.: A survey of structured and object-oriented software specification methods and techniques. ACM Computing Surveys 30 (1998) 459-527