Deriving Operation Contracts from UML Class Diagrams

Jordi Cabot[1] and Cristina Gómez[2]

[1] Estudis d'Informàtica, Multimedia i Telecomunicacions, Universitat Oberta de Catalunya
jcabot@uoc.edu
[2] Dept. de 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 describes how users can modify and evolve the system state. To be useful, such a set must be complete (i.e. through these operations, users should be able to modify the population of all elements in the class diagram) and executable (i.e. for each operation, there must exist a system state over which the operation can be successfully applied). Manual specification of these operations is an error-prone and time-consuming activity. Therefore, the goal of this paper is to automatically provide a basic set of system operations that verify these two properties. Operations are drawn from the elements (classes, attributes, etc) of the class diagram and take into account the possible dependencies between the different change events (i.e. inserts/updates/deletes) that may be applied to them. Afterwards, the designer could reuse our proposal to build up more complex operations.

1 Introduction

The specification of an information system must include all relevant static and dynamic aspects of the domain [9]. The static aspects are collected in structural diagrams, class diagrams in the UML. Dynamic aspects are usually specified by means of a behavioral schema consisting of a set of system operations [11] (also known as domain events [14]) 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 basic modifications over the system state that is perceived by the user of the information system as a single change in the domain. We refer to these basic modifications as structural events. Each structural event, such as “create object”, “update attribute” or “delete link”, represents an elementary change to the elements of a class diagram.

Behavioral schemas must be complete [14] and executable [7]. A behavioral schema *bs* is complete when, through the system operations in *bs*, a user can apply all kinds of structural events to any modifiable element of the class diagram (i.e. given an element *e* of the class diagram and a possible structural event *s* over *e*, there is at least an operation in *bs* that includes *s*). It is executable, 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 class diagram’s integrity constraints). Incomplete behavior

G. Engels et al. (Eds.): MoDELS 2007, LNCS 4735, pp. 196–210, 2007.
© Springer-Verlag Berlin Heidelberg 2007

schemas result in information systems that have parts that the user cannot modify since no available operations address their modification. Non-executable behavior schemas result in information systems with operations that can never be successfully executed.

For instance, given the simple example shown in Fig. 1.1, we must specify an operation to create new employees, an operation to delete employees and two operations to update the *name* and *salary* attributes. This behavior schema is complete since all the modifiable elements in the class diagram (*dateOfBirth* is marked as read only) can be created, updated and deleted through the execution of the system operations. Moreover, it is also executable. The deletion operation can be executed in all states with at least one employee instance. The creation and update operations can be applied provided that the argument corresponding to the new salary value is greater than 600, which is the only restriction imposed by the *ValidSalary* constraint.

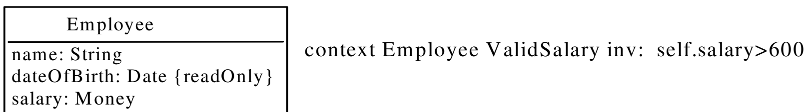

Fig. 1.1. Example of a simple structural schema

For all non-trivial class diagrams the number of required system operations rapidly increases. Therefore, the specification of a complete and executable set of operations becomes an error-prone and time-consuming activity.

We believe that an automatic generation of behavior schemas from UML class diagrams would offer two main benefits. Firstly, it would guarantee the quality (in terms of completeness and executability) of the specified system operations. Secondly, the software development process would be sped up by avoiding a systematic definition of all operations. In this sense, given a class diagram, the main goal of our paper is to provide a method for the automatic generation of a basic behavior schema that satisfies the completeness and executability properties. We define our generated behavior schema as a basic one since we try to keep all defined operations as simple as possible. Operations are declaratively specified by means of OCL contracts. As far as we know, ours is the first approach to provide an automatic generation of a complete and executable behavior schema.

Although our basic schema suffices to cover most common operations appearing in class diagrams, designers may want to generate arbitrary complex operations. Such complex operations may be defined as a combination of our basic ones in order to guarantee their executability as well. Ideally, these more complex operations could also be (semi)automatically generated when additional diagrams (such as the use case diagram [21]) are considered but this is left as further work.

The rest of the paper is organized as follows. Section 2 introduces several preliminary concepts. Section 3 and 4 define the completeness and executability of a behavior schema, respectively. Section 5 presents our generation of a basic complete and executable behavior schema. A case study is shown in Section 6. Finally, Section 7 presents related work and section 8 puts forwards the conclusions and ideas for further research.

2 Preliminary Concepts

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

$$CD = \langle CL, ATT, ASS, AC, GEN, IC \rangle$$

where *CL*, *ATT*, *ASS*, *AC*, *GEN* and *IC* represent the set of classes, attributes, associations, association classes, generalizations and 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. We assume that all associations are binary associations. N-ary associations can easily be expressed in terms of a set of binary ones plus additional constraints [2].

Structural events. The concrete number (and specification) of the system operations required by a class diagram depends on the exact types of structural events provided by the modeling language. The structural event types (and their effect) being considered in this paper are the following:

1. *iCl(x):* inserts a new object (i.e. instance) *x* into class *Cl*. If *Cl* participates in a class taxonomy, *x* is inserted into all (direct or indirect) superclasses as well.
2. *dCl(x):* deletes an existing object *x* from *Cl* and from all its direct and indirect superclasses and subclasses.
3. $uAt_iCl(x,v)$: sets *v* as the new value for the attribute At_i of object *x* (of class *Cl*).
4. $iAs(x_1{:}Cl_1,x_2{:}Cl_2)$: inserts a new link in *As* between objects x_1 of type Cl_1 and x_2 of type Cl_2.
5. $dAs(x_1{:}Cl_1,x_2{:}Cl_2)$: removes the link between objects x_1 and x_2 in *As*.
6. $gCl_cCl_p(x)$: generalizes an object *x* of a (child) subclass Cl_c to a (parent) superclass Cl_p.
7. $sCl_pCl_c(x)$: specializes an object *x* of a superclass Cl_p to Cl_c.

Creation/deletion of instances of association classes requires creating/deleting both the class and association facets of the association class instance with the corresponding events.

Our events are more basic than those proposed in the UML (see the list of *actions* in the UML metamodel [15]). This permits a more fine-grained reasoning. Nevertheless, we could easily define a correspondence between the two sets.

3 Completeness of a Behavior Schema

A behavior schema *bs* is complete when users are able to apply all kinds of changes to the modifiable elements of a class diagram *CD* through the execution of the operations in *bs*, that is, when for each modifiable element *e* in *CD* and each possible structural event *s* over *e*, there is at least one operation in *bs* that includes *s*.

Therefore, completeness is guaranteed if we first compute the set set_{ev} of structural events that may be applied over *CD* and then we ensure that each event *ev*, $ev \in set_{ev}$, is included in one of the system operations in *CD*.

In Section 3.1 we define the notion of modifiability for each kind of model element appearing in a class diagram. Then, in Section 3.2 we compute the set of structural events relevant to a given class diagram (i.e. the set of events that can be possibly

executed over the diagram), taking into account the modifiability of each element in the diagram. To illustrate the process we use the class diagram shown in Fig. 3.1 as a running example.

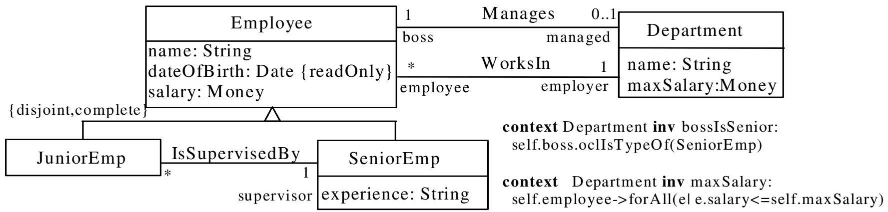

Fig. 3.1. Class diagram used as a running example

3.1 Modifiability of a Model Element

The modifiability of a model element (that is, the possibility of changing the value or population of that element) depends on the type of element and on the (metamodel) properties specified by the designer during its definition.

A class *c* 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 (covering is also known as complete). In a covering generalization set no instances of the supertype can be directly created, they can only be created when one of their subtypes is being instantiated.

An attribute *a* is modifiable when *a* is neither read only nor derived (i.e. *a.isReadOnly=false and a.isDerived=false*, where *isReadOnly* and *isDerived* are properties of the *Property* metaclass).

An association is modifiable when none of its member ends is read only or derived. An association class is modifiable when both its class facet and its association facet are modifiable.

Generalization sets are always modifiable.

All elements in the class diagram in Fig. 3.1 are modifiable except for the *dateOfBirth* attribute, which is marked as *readOnly*.

3.2 Computing the Relevant Structural Events for a Class Diagram

Given a class diagram *CD=<CL, ATT, ASS, AC, GEN, IC>* the set of structural events that may be applied to *CD* are the following:

- *iCl* and *dCl* events for each modifiable class *Cl* in *CD*.
- *iAs* and *dAs* events for each modifiable association *As* in *CD*.
- An uAt_iCl event for each modifiable[1] attribute At_i of a class *Cl*.
- gCl_cCl_p and sCl_pCl_c events for each subclass Cl_c of a superclass Cl_p in a generalization set.

[1] Update events for non-modifiable attributes are only admitted just after the object has been created, as a way of initializing the attribute's value.

In the example of Fig. 3.1, *iJuniorEmp*, *dJuniorEmp* and *gJuniorEmpEmployee* events may be applied over *JuniorEmp*. Similarly, *iSeniorEmp*, *dSeniorEmp, gSeniorEmpEmployee* and *uExperienceSeniorEmp* may be applied over *SeniorEmp*. Relevant events for *Department* are *iDepartment*, *dDepartment, uNameDepartment and uMaxSalaryDepartment* and for *Employee* are *uNameEmployee, uSalaryEmployee, sEmployeeJuniorEmp* and *sEmployeeSeniorEmp*[2]. For *Manages* and *WorksIn*, insertion and deletion events may be applied.

4 Executability of a Behavior Schema

A behavior schema *bs* is executable when for all system operations in *bs* there is at least a system state and a set of arguments for the operation parameters that permit a successful execution of the operation. An operation succeeds when its execution evolves the initial system state to a new state that satisfies all integrity constraints.

Defining an operation as executable does not imply that every time the operation is executed the new system state will be consistent (this depends on the previous state and on the exact arguments passed as parameters for the operation). We just guarantee that it is at least possible to successfully execute it sometime. Otherwise, the operation is completely useless and should be removed.

Executability depends on the set of structural events that the operation applies over the system state. The basic idea is that some events require the presence of other events within the same operation in order to leave the data in a consistent state at the end of the operation execution. As an example, an operation *createDepartment* creating new instances of department (that is, an operation applying the *iDepartment* event) must be in charge of creating a new link in the *Manages* association relating the new department with its boss (*iManages* event). Otherwise, every time this operation is executed the minimum multiplicity of the *boss* role (see Fig. 3.1) becomes violated, and thus, the operation never succeeds.

Therefore, executability is guaranteed if, for each event *ev* included in the effect of a system operation *op,* all other events required by *ev* appear in *op* as well. A behavior schema is executable when all operations are executable.

Dependencies between structural events depend on the type of the event and on the integrity constraints of each particular class diagram. When the dependencies for an event *ev* are being computed, all we need to consider are the minimum multiplicity constraints for associations and attributes[3] and *disjoint* and *complete* constraints (either graphically represented or implicitly induced by textual OCL constraints[4]).

---

[2] *iEmployee* and *dEmployee* events may be applied over *Employee* only when the generalization set in which *Employee* participates as a supertype is defined as incomplete.

[3] Although it is also possible to define minimum multiplicities for the number of objects in a class, they are quite rare. We are therefore not going to consider them in our approach.

[4] Some textual OCL constraints may exactly correspond to minimum multiplicity, disjoint or complete constraints. Also, some may indirectly imply them (for instance, stating that navigating from an object of type *X* to the related *Y* objects we must find more than *N* objects satisfying condition *cond* implies that the minimum cardinality of the *Y* role in the navigated association must be at least *N*).

For other constraints, we can always find a combination of a system state and/or a set of arguments for which the execution of *ev* results in a consistent state. For instance, maximum multiplicity constraints are never violated when *ev* is applied to an empty system state. Constraints restricting the value of the attributes of an object may be satisfied when passing the appropriate arguments as parameters for the event. The same situation occurs with constraints restricting the relationship between an object and related objects. Therefore, all these constraints are ignored when computing the dependencies of *ev*, and thus, when determining the executability of operations including *ev*. As an example, the *maxSalary* constraint (Fig. 3.1) does not affect the executability of operations modifying departments, employees and the links between them. The creation of employees and departments is always successful. Updates of *Salary* and *MaxSalary* attributes may be successful when choosing the right values for the corresponding attributes. The creation of a new *WorksIn* link is successful when the state has at least a department and an employee (who is not already related to a department) that satisfies the *maxSalary* condition.

For the class diagram of Fig. 3.1, several dependencies between the relevant structural events are necessary. For instance, an *iJuniorEmp(x)* event requires the presence of events *uNameEmployee(x,name)*, *uDateOfBirthEmployee(x,date)* and *uSalaryEmployee(x,sal)* to initialize the values of its non-derived attributes. Otherwise, operations that do not include them will always violate the minimum '1' multiplicity of these attributes. Additionally, this event also requires the events *iIsSupervisedBy(x,y)* and *iWorksIn(x,z)* to avoid violating the minimum multiplicity of the *supervisor* and *employer* roles. The complete list of dependencies for this example can be found in Section 5.2.1.

5 Generation of a Complete and Executable Behavior Schema

In this section, we show how to automatically generate a complete and executable behavior schema for a given class diagram *CD*, according to the previous complete and executable properties. Our method has two main phases:

- The assignment of all relevant events for *CD* to a set of new system operations (*completeness*)
- The definition of the actual operation parameters and body in view of the dependencies of the assigned events (*executability*)

In our approach, system operations are assigned to an appropriate class of the class diagram. Other authors argue that it is better to first assign all operations to an artificial class called *System* [11]. The adaptation of our method to this case is straightforward.

Furthermore, operations can be specified in one of two ways: imperatively or declaratively [22]. In an imperative specification, the set of structural events that the operation applies during the operation execution are explicitly defined. 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. In our approach, the imperative version of each operation can be directly deduced from the structural

events we assign to the operation during the generation process. Therefore, we focus on the declarative version.

Note that our pre- and postconditions do not include the verification of the integrity constraints in *CD* (strict interpretation of operation contracts [18]) in order to avoid redundancies between the contracts and the constraints (this improves the quality of the resulting specifications, see [6]). Only those constraints that could potentially affect the executability property of the operations are considered (see the discussion presented in Section 4) and already tackled when reasoning on the dependencies between the structural events assigned to the operation.

5.1 Creating the Required System Operations

Assignment of relevant events for a class diagram *CD* into a set of system operation can be done in many different ways. Since we intend to create a basic behavior schema, our goal is to minimize the complexity of the generated operations. Roughly, we create a different operation in *CD* for each relevant structural event.

Given that set_{ev} is the set of relevant events for *CD* (as computed in section 3.2) the system operations our method generates are the following (operations are assigned to the appropriate class according to the *GRASP* patterns [11]):

- A class operation *Cl::Create* for each *iCl* event in set_{ev}
- A *Cl::Delete* operation for each *dCl* event in set_{ev}
- A Cl_c*::GeneralizeCl* operation for each gCl_cCl_p event in set_{ev}
- A Cl_p*::SpecializeCl* operation for each sCl_pCl_c event in set_{ev}
- A *Cl::UpdateAt*$_i$ operation for each uAt_iCl event in set_{ev}
- Two *P::CreateLinkAs* operations (one for each participant class *P*) for each *iAs* event in set_{ev}.
- Two *P::DeleteLinkAs* operations (one for each participant class *P*) for *dAs* events in set_{ev}

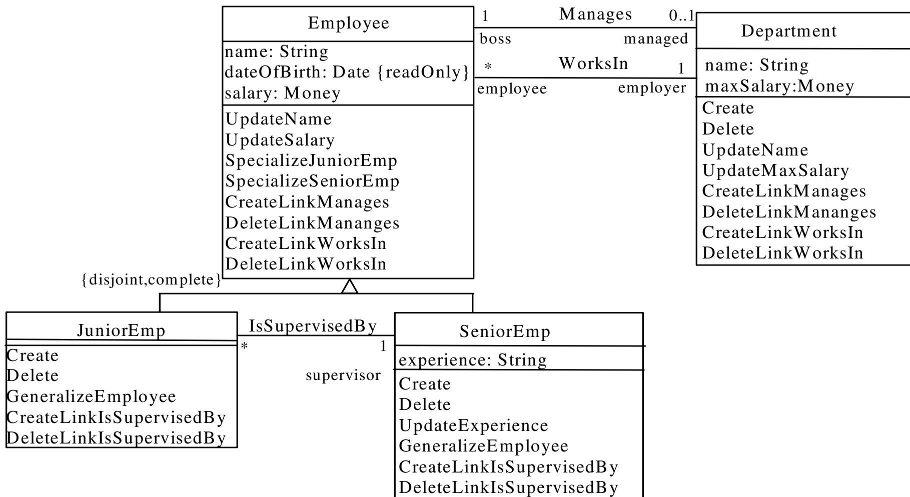

Fig. 5.1. Class diagram with the generated system operations

In UML, operations cannot be assigned to associations (except for association classes). Therefore, operations on associations are assigned to the participants of the association. For recursive associations we can use the name of the opposite role rather than the association name when creating the operation. To satisfy the completeness property, it would be enough to add the operations to one of the participants. However, on behalf of the usability of the generated behavior schema, we prefer to add the operations to both participants. When designing the specified system, designers may add navigability information to the diagram and remove the operations using non-navigable association ends.

Fig. 5.1 shows the running example of Fig. 3.1 once it has been extended to include the set of generated operations.

5.2 Completing the Operation Definition

Following on from the previous section we get a set of operations, each one attached to a class in *CD* and with one of the relevant structural events assigned to it.

To complete the operation definition we need to first determine whether the operation behavior must be extended with new events due to dependencies between them (Section 5.2.1). We will then be able to define the set of parameters for the operation (Section 5.2.2) and its final body (Section 5.2.3).

5.2.1 Computing the Dependencies

A simple dependency for a structural event *ev* is defined as a tuple <*direction, event*> where *event* is the name of the structural event required by *ev* and *direction* indicates whether that event should be executed before *ev* (symbol ←), after *ev* (symbol →) or if the exact position of *ev* is irrelevant (symbol ↑). In fact, the *direction* field is not strictly necessary (in the same way that the exact order of predicates in a postcondition is also irrelevant); as long as all dependent events are applied over the system state, the state at the end of the operation will be consistent. Nevertheless, the *direction* field helps to obtain a more clear and readable contract.

More complex dependencies are expressed as a sequence of simple ones joined with the logical *AND* and *OR* operators (for example, *ev* may require the existence of the events ev_1 and ev_2 or, alternatively, the existence of an event ev_3).

Computing the list of dependencies for an event *ev* is a recursive process. If *ev* requires an event ev_2 we must take into account also the dependencies of ev_2 and so on. Otherwise an operation including *ev* and ev_2 may be non-executable due to the dependencies of ev_2 not being satisfied within the operation. Therefore, to ensure executability, we must compute the *transitive closure* of the dependencies of *ev*. The *transitive closure* can be computed by means of recursively applying the following dependency rules over the new events added to the initially empty list of dependencies for an event *ev* until no more dependencies are added[5].

Note that, due to *OR* dependencies (stating that *ev* depends on an event ev_1 or, alternatively, on an event ev_2) we may obtain different alternative dependency lists at the end of the computation process. Each *OR* dependency is a splitting point. From that point, two new lists are created. The lists are initialized with the contents of the

[5] Termination is guaranteed except in the case of rare multiplicities combinations (as a cyclic sequence of exact one-to-one associations), which require the designer to take part in the process.

original one and then the process continues with each list separately. Each list generates a different operation specification.

The list of dependency rules is the following (in the rules *min(Cl,As)* returns the minimum multiplicity of *Cl* in *As,* i.e. the minimum number of links in *As* in which all instances of *Cl* must participate, and *max(Cl,As)* its maximum multiplicity).:

Rules for computing the dependencies for a structural event *iCl(x)*:

- A dependency dep_{At1} = <→, $uAt_iCl(x,v)$> for each non-derived attribute At_i of *Cl* AND
- A dependency dep_{At2} = <→, $uAt_kCl_p(x,v)$> for each non-derived attribute At_k of Cl_p where Cl_p is a direct or indirect superclass of *Cl* AND
- A number of $min(Cl,As_j)$ dependencies dep_{As1} = <→, $iAs_j(x,y)$> for each non-derived association As_j where *Cl* has a mandatory participation ($min(Cl,As_j) >= 1$) AND
- A number of $min(Cl_p,As_k)$ dependencies dep_{As2} = <→, $iAs_k(x,y)$> for each non-derived association As_s where Cl_p is a direct or indirect superclass of *Cl* and Cl_p has a mandatory participation in As_k.

Dependencies for a *dCl(x)* event :

- A number of $min(Cl,As_j)$ dependencies dep_{As1} = <→, $dAs_j(x,y)$> for each non-derived association As_j where *Cl* has a mandatory participation AND
- A number of $min(Cl',As_k)$ dependencies dep_{As2} = <→, $dAs_k(x,y)$> for each non-derived association As_k where *Cl'* is a direct or indirect superclass or subclass of *Cl* and *Cl'* has a mandatory participation in As_k.

Dependencies for a $sCl_pCl_c(x)$ event:

- A dependency dep_{At} = <→, $uAt_iCl_c(x,v)$> for each non-derived attribute At_i of Cl_c AND
- $min(Cl_c, As_j)$ dependencies dep_{As} = <→, $iAs_j(x,y)$> for each non-derived association As_j where Cl_c has a mandatory participation AND
- A dependency dep_{Spec} = <→, $gCl_{c'}Cl_p(x)$> for a $Cl_{c'}$ class such that $Cl_c \neq Cl_{c'}$ and that *x* is an instance of $Cl_{c'}$. This dependency only applies when the generalization set, for which Cl_p is the supertype, is disjoint and complete; in such a case, specialization of *x* to Cl_c forces the removal (generalization) of *x* from a different subtype $Cl_{c'}$ to satisfy the disjoint constraint. We know that *x* was instance of some $Cl_{c'}$ because the generalization set is complete.

Dependencies for a $gCl_cCl_p(x)$ event:

- $min(Cl_c,As_j)$ dependencies dep_{As} = <→, $dAs_j(x,y)$> for each non-derived association As_j where Cl_c has a mandatory participation AND
- A dependency dep_{Gen} = <→, $sCl_pCl_{c'}(x)$> such that $Cl_c \neq Cl_{c'}$. Again, this dependency only applies when the generalization set, for which Cl_p is the supertype, is disjoint and complete.

Dependencies for an $iAs(x:Cl_1,y:Cl_2)$ event when $min(Cl_1,As)=max(Cl_1,As)$ (the process must be repeated for Cl_2 when $min(Cl_2,As)=max(Cl_2,As)$):

- A dependency dep_{As} = <↑, *dAs(x,z)*> such that <*x,z*> is an existing link in *As*, if $min(Cl_2,As) \neq max(Cl_2,As)$ OR
- A dependency dep_{Ins} = <←, $iCl_1(x)$> OR
- A dependency dep_{Spec} = <←, $sCl_pCl_1(x)$> if Cl_1 has a supertype Cl_p

Dependencies for a $dAs(x{:}Cl_1,y{:}Cl_2)$ event when $min(Cl_1,As)=max(Cl_1,As)$ (the process must be repeated for Cl_2 when $min(Cl_2,As)=max(Cl_2,As)$):

- A dependency $dep_{As} = \langle\uparrow, iAs(x,z)\rangle$ if $min(Cl_2,As) \neq max(Cl_2,As)$ OR
- A dependency $dep_{Ins} = \langle\leftarrow, dCl_1(x)\rangle$ OR
- A dependency $dep_{Gens} = \langle\leftarrow, gCl_1Cl_p(x)\rangle$ if Cl_p is a supertype of Cl_1.

No dependencies are needed for uAt_iCl events since changes on attribute values do not violate cardinality, complete or disjoint constraints. Table 5.1 summarizes the result of the (recursive) application of these rules over the relevant structural events for the class diagram shown in Fig. 3.1.

Table 5.1. Dependencies for the relevant structural events in the class diagram in Fig 3.1

Structural event	Required events (dependencies)
iJuniorEmp(x)	uNameEmployee(x,v_{name}) *AND* uDateOfBirthEmployee(x,v_{date}) *AND* uSalaryEmployee(x,v_{sal}) *AND* iIsSupervisedBy(x,y) AND iWorksIn(x,z)
dJuniorEmp(x)	dIsSupervisedBy(x,y) *AND* dWorksIn(x,z)
sEmployeeJuniorEmp(x)	iIsSupervisedBy(x,y) *AND* gSeniorEmpEmployee(x)
sEmployeeSeniorEmp(x)	uExperienceSeniorEmp(x,exp) *AND* gJuniorEmpEmployee(x) *AND* dIsSupervisedBy(x,y)
gJuniorEmpEmployee(x)	sEmployeeSeniorEmp(x) *AND* dIsSupervisedBy(x,y) *AND* uExperienceSeniorEmp(x,exp)
iSeniorEmp(x)	uNameEmployee(x,name) AND uDateOfBirthEmployee(x,date) AND uSalaryEmployee(x,sal) AND uExperienceSeniorEmp(x,exp) AND iWorksIn(x,z)
dSeniorEmp(x)	dWorksIn(x,z)
gSeniorEmpEmployee(x)	sEmployeeJuniorEmp(x) AND iIsSupervisedBy(x,y)
iDepartment(x)	uNameDepartment(x,name) AND uMaxSalaryDepartment(x,maxSal) AND iManages(x,y)
dDepartment(x)	dManages(x,y)
iManages(x,y)	dManages(x:Department,z:Employee) OR iDepartment(x)
dManages(x,y)	iManages(x:Department,z:Employee) OR dDepartment(x)
iWorksIn(x,y)	dWorksIn(z:Department,y:Employee) OR iEmployee(y)
dWorksIn(x,y)	iWorksIn(z:Department,y:Employee) OR dEmployee(y)
iIsSupervisedBy(x,y)	dIsSupervisedBy(z:SeniorEmp,y:JuniorEmp) OR iJuniorEmp(y) OR sJuniorEmp(y)
dIsSupervisedBy(x,y)	iIsSupervisedBy(z:SeniorEmp,y:JuniorEmp) OR dJuniorEmp(y) OR gJuniorEmpEmployee(y)

5.2.2 Defining the Operation Signature

The signature of an operation *op* depends on the list $list_{ev}$ of structural events the operation consist of (computed as shown in the previous section) and the class where *op* is attached.

Each event $ev \in list_{ev}$ may require the addition of new parameters in the signature. The basic idea is that every variable that appears as a parameter of *ev* must also appear as a parameter (of the same type) in the operation. Four exceptions apply:

1. Variables for *iCl* events are not parameters of the operation. These new objects are created *during* the operation execution.
2. A parameter variable that has already appeared in a previous event does not generate a new operation parameter (i.e. if an operation consists of two events $iAsX(x_1,x_2)$ and $iAsY(x_1,x_3)$ only three parameters x_1, x_2 and x_3 are defined).

3. We use the implicit parameter *self* as a replacement for one of the parameters whose type is the class to which the operation is attached (i.e. if an operation defined in a class *Cl* has the event $uAt_iCl(x,v)$ only a parameter for *v* is generated; the implicit *self* parameter is used whenever *x* appears).
4. Variables for *dAs* events not included in a *DeleteLinkAs* or a *CreateLinkAs* operation are not parameters of the operation. Those deletions are required by *dCl* or gCl_cCl_p events. In those cases, the link/s to be deleted are the ones in which the *self* parameter participates, and thus, they can be determined automatically.

For instance, the operation *JuniorEmp::Create* of Fig 5.1 consists of the *iJuniorEmp(x)* event and all its dependencies defined in Table 5.1 ($uNameEmployee(x,v_{name})$, $uDateOfBirthEmployee(x,v_{date})$, $uSalaryEmployee(x,v_{sal})$, *iIsSupervisedBy(x,y)*, *iWorksIn(x,z))*. From this list of events we may determine the signature of the *Create* operation as follows:

$Create(v_{name}:String,\ v_{date}:Date,\ v_{sal}:Money,\ y: SeniorEmp,\ z:Department)$.

Similarly, the signature of *Department::Delete* is simply *Delete()*. This signature is calculated from the list of events for this operation *(dDepartment(x), dManages(x,y))*. In accordance with the rules above, none of the event variables must be added as an explicit parameter of this operation.

5.2.3 Defining the Operation Body

In an imperative specification of the operation effect, the operation body is simply defined as the ordered list of structural events computed for the operation as shown in the previous sections. However, in a declarative specification we must transform the list of structural events into an OCL contract such that the application of the events over a state satisfying the contract preconditions evolves this initial state into a new state that satisfies the contract postconditions.

As discussed in the introduction of Section 5, our operations do not explicitly include the integrity checking of the class diagram's constraints. Therefore, our operations do not include preconditions and the postconditions refer solely to the operation's own behavior. Constraints that may affect the executability property of the operation will already have been considered when computing the dependencies.

The initial postcondition is obtained by means of translating each single event into an equivalent boolean condition and concatenating the different conditions with *AND* operators (this translation is not unique, see [3]). In the following we provide a possible boolean condition for each event.

1. *iCl(x): x.oclIsNew() and x.oclIsTypeOf(Cl)*
2. *dCl(x):OclAny::allInstances()=OclAny::allInstances()@pre->excluding(x)*
3. $uAt_iCl(x,v)$: $x.At_i=v$
4. $iAs(x_1,x_2)$: $x_1.r_2$*->includes*(x_2) (r_2 is the role corresponding to x_2 in As)
5. $dAs(x_1,x_2)$: $x_1.r_2$*->excludes*(x_2) (r_2 is the role corresponding to x_2 in As)
6. $gCl_cCl_p(x)$: *x.oclIsTypeOf*(Cl_p)
7. $sCl_pCl_c(x)$: *x.oclIsTypeOf*(Cl_c)

Note that *OclAny* is the supertype of all types in the UML class diagram [16]. Using *OclAny* instead of *Cl* in the definition of the *dCl(x)* event condition guarantees that the object *x* is completely removed from the system (and that it does not remain, for example, as an instance of a supertype of *Cl*).

The resulting postcondition may need to be refined depending on the combination of translated structural events. For instance, if several sCl_pCl_c events are applied over an instance *x*, only the translation for the event over the more specific class is necessary. Translation for events $dAs(x_1,x_2)$ can be discarded when $dCl(x_1)$ and/or $dCl'(x_2)$ events also appear (usually, the deletion of links is implicitly assumed).

As an example, we provide the contract for the operation *JuniorEmp::Create* and for the operation *Department::Delete*.

context JuniorEmp::Create(v_{name}:String, v_{date}:Date, v_{sal}:Money,y:SeniorEmp, z:Department)
post: x.oclIsNew() and x.oclIsTypeOf(JuniorEmp) and x.name=v_{name} and x.dateOfBirth=v_{date} and x.salary=v_{sal} and x.supervisor->includes(y) and x.employer->includes(z)

context Department::Delete()
post: OclAny::allInstances()=OclAny::allInstances()@pre->excluding(self)

6 Case Study

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

In particular, we analyze a system for a Conference Management Application as specified in [19]. 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 associations, 2 non-covering generalization sets and several constraints. The proposed behavior schema includes 29 operations.

Our method is able to completely generate 13 of the 29 operations (6 creation operations, 3 deletion operations and 4 update operations). Seven additional operations (each one assigning one or more constant values to attributes of the class diagram) can be directly mapped to our generated system operations by passing these constant values as parameters of our $UpdateAt_i$ operations. The rest of the system operations, 9 out of 29, can only be 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 proposal helps designers by reducing by 69% (20 of 29) the number of operations to be defined and by providing at least an initial contract specification for the remaining ones.

Moreover, our approach generates several operations that did not appear in the manually specified schema (for instance, all *GeneralizeCl* and *SpecializeCl* and some $UpdateAt_i$ operations). Designers could use this information to detect whether some

required operations must be added to the class diagram or if the specification of the class diagram is incomplete (for instance, attributes are not marked as *readOnly* or *derived*, completeness and/or disjointness of generalizations sets is not defined, etc).

7 Related Research

As far as we know, ours is the first approach to study the application of the completeness and executability properties to the automatic generation of a basic behavior schema.

[10] partially determines the set of possible structural events to be applied to a class diagram (generalizations are not considered). However, in this approach system operations must be manually defined as a combination of a set of structural events. Therefore, the completenesss and executability of these operations must be guaranteed by designers. In this approach, operations are specified using the formal notation B.

[8] derives a set of basic operations (similar to our concept of structural events) and a set of elementary operations from an EER diagram. These latter operations are not necessarily executable since cardinality constraints are not considered in any case.

Alternatively, other approaches try to generate system operations from the information provided in different diagrams, such as the use case diagram. For instance, [21] presents a method for generating system operations from use cases specifications. Nevertheless, this method is not automatic and completeness and executability properties of the generated behavior schema are not analyzed.

The idea of dependencies between structural events is not new. This problem has been addressed in the (deductive) database field as part of the more general problem of integrity maintenance at compile-time (see [20], [12], [17]). In those cases, the goal was similar: to extend a (predefined) given transaction/operation with additional events to always ensure its successful execution. However, their expressivity regarding the definition of the structural diagram and the set of admitted structural event types is more restricted than in our method.

Regarding OCL contracts, [1] provides some patterns to help designers in their manual definition but automatic generation of contracts is not studied.

Some IDEs (as [5] or [13]) are able to automatically generate basic getter/setter and creator methods for classes. However, the methods do not take into account the possible dependencies among the included events.

8 Conclusions and Further Research

The complete definition of the behavior schema is one of the most important tasks in the analysis and design stages of an information system. The method presented in this paper facilitates this task by automatically generating an initial set of system operations. Operations are drawn from the structure of the class diagram.

The executability and completeness properties of this set of operations guarantee the quality of the behavior schema. Designers are free to directly use our operations (avoiding the manual definition of the behavior schema) or to reuse our method in order to create more complex operations (while maintaining the previous properties).

We believe our method is useful even when the designer is not interested in a complete and automatic generation of the behavior schema. If integrated in an OCL editor, our method could assist the designer during the definition of OCL contracts by means of *suggesting* additional predicates to complete the postconditions. These suggestions would be provided based on our dependencies computation.

As a further research, we would like to study how we can reuse the information of use cases (as in [21]) and state diagrams to automatically derive more complex system operations. We are also interested in studying how to integrate the efficient verification of all constraints that may be violated by the operation execution (the relevant constraints can be determined with [4]) into the preconditions of our generated operation contracts so that a successful execution of the operation is *always* guaranteed (providing that the precondition is satisfied). Additionally, we plan to apply the completeness and executability properties to the verification of existing behavior schemas.

Acknowledgments

We would like to thank the people of the GMC group and the anonymous reviewers 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 Behavioral Specification of Software Components. In: Dubois, E., Pohl, K. (eds.) CAiSE 2006. LNCS, vol. 4001, pp. 255–269. Springer, Heidelberg (2006)
2. Andrew, J., Mcallister, D.S.: An approach for decomposing N-ary data relationships. Software: Practice and Experience 28, 125–154 (1998)
3. Cabot, J.: From Declarative to Imperative UML/OCL Operation Specifications. In: ER 2007. LNCS, Springer, Heidelberg (to appear, 2007)
4. Cabot, J., Teniente, E.: Determining the Structural Events that May Violate an Integrity Constraint. In: Baar, T., Strohmeier, A., Moreira, A., Mellor, S.J. (eds.) UML 2004. LNCS, vol. 3273, pp. 173–187. Springer, Heidelberg (2004)
5. CincomSmalltalk. VisualWorks, http://www.cincomsmalltalk.com/
6. Costal, D., Sancho, M.-R., Teniente, E.: Understanding Redundancy in UML Models for Object-Oriented Analysis. In: Pidduck, A.B., Mylopoulos, J., Woo, C.C., Ozsu, M.T. (eds.) CAiSE 2002. LNCS, vol. 2348, pp. 659–674. Springer, Heidelberg (2002)
7. Costal, D., Teniente, E., Urpí, T., Farré, C.: Handling Conceptual Model Validation by Planning. In: Constantopoulos, P., Vassiliou, Y., Mylopoulos, J. (eds.) CAiSE 1996. LNCS, vol. 1080, pp. 255–271. Springer, Heidelberg (1996)
8. 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. Data & Knowledge Engineering 9, 157–204 (1992)
9. ISO/TC97/SC5/WG3: Concepts and Terminology for the Conceptual Schema and Information Base. ISO (1982)

10. Laleau, R., Polack, F.: Specification of Integrity-Preserving Operations in Information Systems by Using a Formal UML-based Language. Information and Software Technology 43, 693–704 (2001)
11. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, 2nd edn. Prentice-Hall, Englewood Cliffs (2001)
12. Link, S.: Consistency Enforcement in Databases. In: Bertossi, L., Katona, G.O.H., Schewe, K.-D., Thalheim, B. (eds.) Semantics in Databases. LNCS, vol. 2582, pp. 139–159. Springer, Heidelberg (2003)
13. Microsoft. Visual Studio 2008, http://msdn2.microsoft.com/en-us/vstudio/default.aspx
14. Olivé, A.: Conceptual Modeling of Information Systems. Springer, Heidelberg (2007)
15. OMG: UML 2.0 Superstructure Specification. OMG Adopted Specification (ptc/03-08-02) (2003)
16. OMG: UML 2.0 OCL Specification. OMG Adopted Specification (ptc/03-10-14) (2003)
17. Pastor, J.A., Olivé, A.: Supporting Transaction Design in Conceptual Modelling of Information Systems. In: Iivari, J., Rossi, M., Lyytinen, K. (eds.) CAiSE 1995. LNCS, vol. 932, pp. 40–53. Springer, Heidelberg (1995)
18. Queralt, A., Teniente, E.: Specifying the Semantics of Operation Contracts in Conceptual Modeling. Journal on Data Semantics 7, 33–56 (2006)
19. Raventós, R.: A conceptual schema for a conference management application. UPC, LSI Technical Report, LSI-05-1-R (2005)
20. Schewe, K.-D., Thalheim, B.: Towards a theory of consistency enforcement. Acta Informatica 36, 97–141 (1999)
21. Sendall, S., Strohmeier, A.: From use cases to system operation specifications. In: Evans, A., Kent, S., Selic, B. (eds.) UML 2000. LNCS, vol. 1939, Springer, Heidelberg (2000)
22. Wieringa, R.: A survey of structured and object-oriented software specification methods and techniques. ACM Computing Surveys 30, 459–527 (1998)