

Automatic Generation of Basic Behavior Schemas from UML Class Diagrams

Manoli Albert¹, Jordi Cabot², Cristina Gómez³, Vicente Pelechano¹

¹Department of Information Systems and Computation, Technical University of Valencia
Camino de Vera s/n 46022 Valencia (Spain)
{malbert, pele}@dsic.upv.es

²Estudis d'Informàtica, Multimedia i Telecomunicacions, Universitat Oberta de Catalunya
Rambla del Poblenou, 156, E08018 Barcelona (Spain)
jcabot@uoc.edu

³Dept. de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya
Campus Nord, Edif. Omega, Jordi Girona 1-3, 08034 Barcelona (Spain)
crisrina@lsi.upc.edu

Abstract: The specification of a software system must include all relevant static and dynamic aspects of the domain. Dynamic aspects are usually specified by means of a behavioral schema consisting of a set of system operations that the user may execute to update 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). A manual specification of these operations is an error-prone and time-consuming activity. Therefore, the aim of this paper is to present a strategy for the automatic generation of a basic behavior schema. Operations in the schema are drawn from the static aspects of the domain as defined in the UML class diagram and take into account possible dependencies among them to ensure the completeness and executability of the operations. We believe our approach is especially useful in a Model-Driven Development setting, where the full implementation of the system is derived from its specification. In this context, our approach facilitates the definition of the behavioral specification and ensures its quality obtaining, as a result, an improved code generation phase.

Keywords: Behavior schema, operation, structural event, class diagram, UML, OCL

1 Introduction

Model Driven Development (MDD) [40] is currently one of the most popular approaches in the Software Engineering discipline. In MDD, models become the primary artifacts of the software development process. The full implementation of a software system is aimed to be (semi-)automatically derived from them.

MDD processes start by specifying all the knowledge of the domain the software system must know in order to perform its functions. In conceptual modeling, the

resulting domain model is known as the Conceptual Schema (CS) of the system. A CS must include all relevant static (i.e. structural) and dynamic (i.e. behavioral) aspects of the domain [31].

Clearly, a sound definition of the CS is necessary to guarantee the functional equivalence between the real-world domain and the generated code. However, the conceptual modeling phase is a manual step that requires the implication of several people with different roles (domain experts, analysts, end-users and so forth), which makes this task error-prone and time-consuming.

Therefore, any proposal able to introduce some kind of automation in the modeling phase will help to minimize the errors and reduce time and costs during the development process. Moreover, it will have a direct impact on the quality of the final system implementation by avoiding the propagation of errors from the CS to the generated code.

This paper contributes to the improvement of this conceptual modeling phase by defining a method for the automatic generation of the behavioral part of a CS from an initial structural subschema expressed as a UML class diagram. Behavioral schemas consist of a set of system operations [22] (also known as domain events [31]) 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 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.

With our method, analysts can 1) avoid a systematic definition of all operations and 2) ensure the quality of the behavior schema in terms of their completeness [31] and executability [14] properties. As far as we know, ours is the first approach to provide the automatic generation of a complete and executable basic behavior schema. Additionally, our method is also able to deduce the OCL contracts corresponding to the declarative version of each operation [42] from the set of structural events that define its effect.

Roughly, 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 that performs some kind of change on the population of 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 schemas result in information systems with parts that the user cannot modify since no available operations tackle their modification. Non-executable behavior schemas result in information systems with operations that can never be successfully executed and thus that are completely useless.

As an initial result of our method, we obtain a basic behavior schema that suffices to cover the typical modification operations commonly needed in any software system and that it is specially suitable for systems that need to maintain a representation of the state of the domain in order to perform their functions (as all Information Systems in

charge of processing the data and the information of an organization). Moreover, designers may want to generate additional arbitrary complex operations. Such complex operations may be defined as a combination of our basic ones in order to guarantee their completeness and executability as well.

The work reported here extends our previous work [9] in several directions. First, we depart from an enriched class diagram, where several new properties for classes and associations are considered. This permits to generate a set of operations closer to the ones envisaged by the designer. Second, we have extended and refined the study of the dependencies between the different structural events. Third, we have adapted our method to the generation of arbitrary complex operations. Fourth, we have validated our method by comparing the behavior schema we get with the ones manually specified by the designer for three different applications. Finally, we have developed a preliminary version of a prototype tool that implements our approach.

The rest of the paper is organized as follows. The next section provides a quick overview of our method. Section 3 reviews some basic concepts of UML structural schemas. Our generation method is explained in detail in section 4. Section 5 shows how the method can be used to help designers to define arbitrary complex operations. Several case studies and a prototype tool are shown in section 6 and 7, respectively. Finally, section 8 presents related work and section 9 puts forwards the conclusions and ideas for further research.

2 Method Overview

This section gives an overview of the method proposed in this paper. This overview allows the reader to have a global perspective of its different steps, which are detailed in the next sections.

Our method works at the conceptual-schema level. Conceptual schemas may be expressed as the combination of two main components: the structural part and the behavioral part. In particular, the input of our method is a conceptual schema $CS = \langle S, \emptyset \rangle$ (i.e., a conceptual schema with S as structural part and no behavioral component yet). As a result, our method returns a new $CS' = \langle S, B \rangle$ that maintains all elements in the original structural schema S and creates a first behavioral schema B for CS . B is generated based on the information in S , and thus, it contains those system operations that can be derived from the information that classes, associations, cardinalities and other elements in S provide. CS' is a completely standard conceptual schema. Therefore, it can be processed by the same CASE tool used in the specification of the original schema. As an example, once we have generated the complete CS' we could use the typical code-generation functionalities of current CASE tools to automatically generate an implementation of the system, including the new operations in B created by our method.

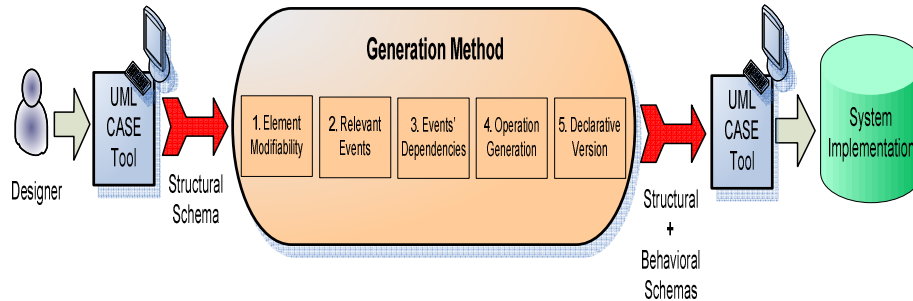


Fig. 2.1. Overall picture of the method

Our method consists of five basic steps. In what follows we briefly describe these steps. To illustrate each step we will (partially) show their effect over the simple structural schema of Fig. 2.2.

Ordered list of steps:

1. *Element modifiability*: Computation of the set of model elements of the structural schema for which some kind of modification (insertions, deletions, and so forth) may be applied. Some elements may be only modifiable at certain times or after the execution of certain events. Non-modifiable elements are discarded during the rest of the process.

All elements in the example are modifiable. Even the attribute birth is modifiable since immediately after a new employee is created its birth date must be initialized (though afterwards that value cannot be changed)

2. *Relevant events*: Computation of all types of structural events that may be applied over the previous modifiable elements. Again, some of these events can only be applied under certain circumstances, depending on the properties of the model elements.

The relevant events for the example are:

- Insertions and deletions on Employee, Department and Manages
- Updates on Employee-name and Department-name
- Updates on Employee-birth (due to its read-only property this event can only be applied after insertions on Employee; this will be ensured by the dependencies computed in the next step)

3. *Events' dependencies*: Detection of all dependencies among the relevant events. To ensure the executability of the operations, we need to determine the dependencies among the different events so that, every time one of the events is executed all its dependencies are applied as well.

For instance, we have that, in the example, the insertion of a new department must be followed by the initialization of the department name (the name is not an optional attribute) and the creation of a new

link between the new department and an employee (to satisfy the minimum cardinality constraint of the Manages association; otherwise the new system state would not be consistent with the constraints).

4. *Operation generation*: Creation of all necessary operations in the schema. To ensure completeness, all the relevant events must be included in at least one operation. Our basic strategy will be to generate a new operation (with the appropriate body and signature) for each relevant event (plus its dependencies) that may be freely applied at any time by a user.

One of the operations that must be generated is Department::create (to cover the event Insertion on Department, determined as relevant in step 2). Its body will be the following (note that additional events are needed because of the dependencies computed in step 3):

```
Department::create(vname:String, vboss: Employee)
{InsertDepartment(d);
 UpdateNameDepartment(d, vname);
 InsertManages(d, vboss);}
```

5. *Declarative version*: At the end of step 4, the operations' bodies are defined as a sequence of structural events (imperative version). Optionally, the designer may prefer to translate this imperative definition into a declarative one by means of expressing the operations as a set of pre and postconditions.

The declarative version of Department::create would be:

```
Department::create(vname:String, vboss: Employee)
post: d.oclIsNew() and d.oclIsTypeOf(Department) and
d.name=vname and d.boss=vboss
```

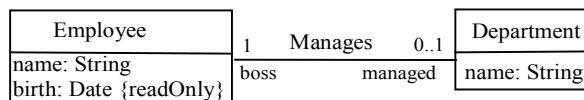


Fig. 2.2. Simple conceptual schema showing departments and their managers

3 Basic Concepts

The aim of this section is to introduce the basic terminology and definitions regarding structural schemas and, in particular, UML class diagrams [35], relevant to the goal of this paper. The model elements in the diagram (and their characteristics) will determine the exact set of operations generated by our method.

We represent a class diagram CD as a tuple:

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

where CL, ASS, ATT, AC and GEN represent the set of classes, associations, attributes, association classes and generalization sets of the class diagram CD, respectively. We will assume that *CD* is syntactically and semantically correct, meaning that all elements in *CD* satisfy the well-formedness rules of the UML metamodel and the basic quality properties (*satisfiability*, *liveliness*,...) that can be expected for all class diagrams [6].

As an example, consider the class diagram of Fig. 3.1 representing information about companies of a country, their departments, employees and investors. Each department is owned by a company and has a manager and a set of employees working in it. Employees may be junior or senior. In this diagram, *Employee*, *Department*, *Company*, ... would be members of the set CL; *Manages*, *WorksFor*, *Owns*, *IsSupervisedBy* and *LocatedIn* would be members of ASS and so forth.

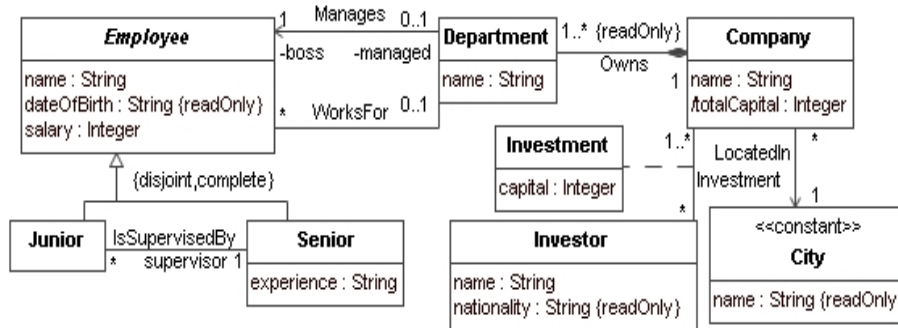


Fig 3.1. Class diagram used as a running example

3.1 Classes

A class *Cl* of CL describes the common characteristics of a set of objects of the domain. A class is an abstract class when it does not provide a complete declaration and can typically not be instantiated [35]. For instance, in Fig. 3.1, *Employee* is an abstract class. All objects of *Employee* must also be either instance of the *Senior* class or of the *Junior* class. We denote by *isAbstract(Cl)* the fact that a class *Cl* is an abstract class. *isAbstract(Cl)* returns *true* when *Cl* is abstract and *false* otherwise.

We also consider *constant* and *permanent* classes [31], [7]. A class is constant if its population does not change during the lifespan of the system. A class is permanent if its objects never cease to be instances of it, that is, the population of a permanent class cannot decrease. The stereotypes <<constant>> and <<permanent>> will be used to identify permanent and constant classes. For example, *City* of Fig. 3.1 is a constant class because it is assumed that the number of cities does not change over time. Boolean functions *isConstant(Cl)* and *isPermanent(Cl)* will be used to identify constant and permanent classes.

3.2 Associations

A binary association As of ASS has a name and two association ends¹ defining the role each participant class plays in the association. Each link between the two participants represents a semantic connection between the objects.

An association may be *derived* or *basic*. If As is derived then its instances can be drawn from other information available in the diagram. Otherwise, As is basic and its values must be provided by the users. We denote by $isDerived(As)$ the boolean function that informs about the derivability of As .

For the sake of clarity, associations will be formally represented as $As(p_1:Cl_1, p_2:Cl_2)$, denoting an association named As with the class participants Cl_1 and Cl_2 playing the roles p_1 and p_2 , respectively. When in As the association ends p_1 and p_2 do not have an explicit name, the name of the participant class in lowercase is used. For instance, in Fig. 3.1, the association $Manages(boss:Employee, managed: Department)$ represents the relationship between departments and their respective bosses. $isDerived(Manages)=false$.

3.2.1 Association ends

Apart from a name, association ends may have several properties that are relevant for the generation of our behavior schema:

Minimum and maximum cardinality. An association end p_2 in $As(p_1:Cl_1, p_2:Cl_2)$ may have a minimum and maximum cardinality.

- The minimum cardinality min for p_2 indicates that all objects of Cl_1 must be related at least with min objects of Cl_2 .
- A maximum cardinality max for p_2 defines that an object of Cl_1 cannot be related with more than max objects of Cl_2 .

We denote by $min(p_2; As)$ and $max(p_2; As)$ these cardinality constraints. For instance, $WorksFor$ association establishes that an employee works at most in one department, so $min(department; WorksFor)=0$ and $max(department; WorksFor)=1$.

Changeability. An association end p_2 in $As(p_1:Cl_1, p_2:Cl_2)$ may have a changeability value that specifies whether objects of Cl_1 can be dynamically connected or disconnected (by creating or destroying links of As) with objects of Cl_2 . The values of changeability are:

- *readOnly*: indicates that insertions and deletions of links of As over an object o of Cl_1 is not possible except for the moment immediately following the creation of o .
- *changeable*: indicates that insertions and deletions of links for objects of Cl_1 is always possible.

The function $changeability(p_i; As)$ returns the changeability value for the association end p_i in As . Default value of changeability is *changeable*. In the example of Figure 3.1, the association end *company* of the association *Owns* is the only

¹ We assume that all associations are binary associations. N-ary associations can be easily expressed in terms of a set of binary ones plus additional constraints [24].

readOnly association end (meaning that departments cannot be moved from one company to another) since composite ends cannot be *changeable* [2].

Aggregation. An association end p_i in $As(p_1:Cl_1, p_2:Cl_2)$ may have an aggregation kind value that specifies whether p_i is an aggregate or a composite [35]. Associations with one aggregate (composite) end are called aggregation (composition) associations. An aggregation is a whole-part relationship that is transitive and antisymmetric across all their links. Composition is a stronger form of aggregation in which the composite is an aggregate with the additional constraint that establishes that a part must be included in at most one composite at a time. If a composite is deleted, all its parts are usually deleted with it. A particular interpretation of the association, aggregation and composition concepts may be found in [2]. The function $aggregationKind(p_i; As)$ returns the value of the type of aggregation for association end p_i in As . Their values are:

- *none*: indicates that p_i is not an aggregate or a composite.
- *shared*: indicates that p_i is an aggregate but not a composite.
- *composite*: indicates that p_i is a composite.

For shared and composite ends the other end must be defined as *none*.

As an example, in Fig. 3.1, $aggregationKind(company; Owns) = composite$ since companies are a composite of departments. Deletion of a company implies the deletion of its departments.

Navigability. An association end p_2 in $As(p_1:Cl_1, p_2:Cl_2)$ is navigable when instances of Cl_1 may access the related objects in Cl_2 . The boolean function $isNavigable(p_i, As)$ will be used to determine the navigability of an association end p_i in As . In our example all association ends except for *managed* of the *Manages* association and *company* of *LocatedIn* association are navigable.

3.1.3 Attributes

An attribute *at* of ATT is a named element within a class that describes the values that instances of the class may hold [35]. Similarly to the case of association and association end properties, an attribute may have multiplicity (denoted by $min(at; Cl)$ and $max(at; Cl)$), changeability ($changeability(at; Cl)$) and derivability ($derived(at; Cl)$) values with similar semantics.

For the sake of simplicity we will assume that all attributes in ATT have a maximum cardinality of 1. Multi-valued attributes can be represented as a binary association between the class owning the attribute and the corresponding data type. Therefore, they can be treated following exactly the same procedure we propose for dealing with associations.

In the running example, the attribute *nationality* of an investor is marked as *readOnly* (i.e. $changeability(nationality; Investor) = readOnly$) because its value may not change after the creation of the investor object and the attribute *totalCapital* of a company is derived since its value may be calculated as the sum of the capital of each investor.

3.4 Association Classes

An association class Ac of AC is both an association and a class. It can be seen as an association that also has class properties, or as a class with association properties. We will assume that class and association properties do not contradict each other.

Instances of association classes are called *link objects* and may contain attributes as well as references to other objects. A link object takes its identity from the set of objects referenced by it. Attribute values are not involved in providing identity [35]. For instance, in the example of Fig. 3.1, *Investment* is an association class whose instances are links between companies and their investors. Each link holds the amount of capital invested by the investor in the company.

3.5 Generalizations

A generalization set Gen in GEN , denoted by $Gen(E;E_1,\dots,E_n)$, is a taxonomic relationship between a more general class Cl (superclass) and a set of more specific classes Cl_1,\dots,Cl_n (subclasses). Each individual relationship between the supertype and one of the subtypes is called a generalization. Generalization sets may be *disjoint* (every object of Cl is instance of at most one Cl_i) and/or *complete* (every object of Cl is instance of at least one Cl_i). *Complete* property is also known as *covering*. It is worth to note that a supertype s of a covering and disjoint generalization set is always abstract (i.e. $isAbstract(s)=true$). No instances of s can be directly created. They can only be created when one of their subtypes is being instantiated.

A disjoint and complete generalization set of *Employee* into *Senior* and *Junior* is shown in the running example.

4 Our Method

This section presents our method for the generation of a complete and executable behavior schema drawn from a structural schema represented as a UML class diagram.

Completeness of a behavior schema. We say that a behavior schema bs is **complete** when users are able to apply all kinds of changes (as insertions, deletions and updates) to the population (i.e. values or instances) of the modifiable elements of a class diagram CD by means of executing the operations in bs , that is, when for each modifiable element e in CD and each possible structural event s over the population of 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 operations in CD .

Executability of a behavior schema. A behavior schema bs is **executable** when for each operation op in bs there is at least a system state and a set of argument values for the operation parameters that permit a successful execution of op . An operation succeeds when its execution evolves the initial system state to a new state that satisfies all integrity constraints. Note that defining an operation as executable does not imply

that every time the operation is executed the new system state is consistent with the constraints (this depends on the previous state and on the concrete argument values passed as parameters for the operation). We just guarantee that it is at least possible to successfully execute the operation. Otherwise, the operation is completely useless and should be removed. Afterwards, the designer could optionally extend our operations to include additional conditions (e.g. derived from the OCL constraints defined in the class diagram) to be satisfied by the initial system state in order to always ensure a successful execution of the operation². This part is not covered in this paper and left as further work.

As we will see in Section 6, these two basic conditions suffice to automatically generate a set of operations that highly correspond to the operations that designers would specify themselves for the same class diagram.

Next subsections explain in detail each step of our method, as sketched in Section 2. We use the class diagram of Fig. 3.1 to illustrate our proposal. The complete behavior schema generated for this running example can be found in the Appendix.

4.1 Determining the modifiability of a model element

The modifiability of a model element is defined as the possibility of changing the value or population of that element at run-time. Modifiability depends on the type of the element and on its properties, specified by the designer during its definition.

A class Cl is modifiable iff *not isAbstract(Cl)* and *not isConstant(Cl)* (i.e. when Cl is not an abstract nor a constant class).

An attribute at of Cl is modifiable iff *not isDerived(at;Cl)*. A *readOnly* attribute is modifiable but only after the creation of a new Cl instance so that its value may be initialized. After that, the value cannot be changed.

An association end p_i of $As(p_1:Cl_1,p_2:Cl_2)$ is modifiable iff *not isDerived(As)*. Similarly to *readOnly* attributes, *readOnly* association ends can be modified only as part of the creation and initialization of a participant instance.

An association is modifiable iff *not isDerived(As)*.

An association class is modifiable when both its class facet (i.e. its properties when regarding the association class as a class) and its association facet (i.e. when using it as an association) are modifiable.

Generalization sets are always modifiable.

All elements in the class diagram in Fig. 3.1 are modifiable except for the *totalCapital* attribute (which is marked as *derived*) and the *City* class.

4.2 Computing the relevant structural events for a class diagram

In this step we identify the set of structural events that users must be able to apply over the modifiable elements of a class diagram CD .

² The inclusion of these additional conditions at the analysis level is sometimes considered redundant [13]

Clearly, this set of events depends on the types of structural events admitted by the modeling language we plan to use. In what follows, we introduce the structural event types (and their effect) we consider in this paper³:

1. $iCl(x)$: inserts a new object x into class Cl . If Cl participates in a class taxonomy, x is inserted into all (direct or indirect) superclasses as well. For example, the structural event $iDepartment(x)$ inserts a new department x into class *Department* of Fig. 3.1.
2. $dCl(x)$: deletes an object x from class Cl and from all its direct and indirect superclasses. As an example, $dDepartment(x)$ removes department x from class *Department*.
3. $uAt_iCl(x,v)$: sets v as the new value for the attribute At_i of object x (of class Cl). As said before, we handle multi-valued attributes as associations.
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 . As an example, the event $iManages(x,y)$ inserts a new link between employee x and department y .
5. $dAs(x_1:Cl_1,x_2:Cl_2)$: removes the link between objects x_1 and x_2 in As . The event $dManages(x,y)$ removes the link between objects x and y .
6. $gCl_cCl_p(x)$: generalizes an object x of a (child) subclass Cl_c to a (parent) superclass Cl_p . This event is applicable if x is a direct instance of Cl_c . The structural event $gJuniorEmployee(x)$ generalizes a junior x to an employee.
7. $sCl_pCl_c(x)$: specializes an object x of a superclass Cl_p to an immediate subclass Cl_c . The event is only applicable if x is an instance of Cl_p and not of Cl_c . The event $sEmployeeJunior(x)$ specializes an employee x into a junior.
8. $iAsCl(x:AsCl,y_1:Cl_1,y_2:Cl_2)$: inserts a new link object x in the association class $AsCl$. This new object is related with objects y_1 of type Cl_1 and y_2 of type Cl_2 . The event $iInvestment(x,co,in)$ inserts a new link object x between company co and investor in .
9. $dAsCl(x_1:Cl_1,x_2:Cl_2)$: removes the link object between objects x_1 and x_2 in $AsCl$. The event $dInvestment(co,in)$ removes the link object between company co and investor in .

Given these possible kinds of structural events, Table 4.1 identifies the events that may be applied over a given class diagram CD , depending on the properties of its model elements. A *comments* column is included in the table in order to clarify particular executability characteristics of some of the events.

³ Our events are more basic than the list of actions proposed in the UML [29] (for instance, we split the *ReclassifyAction* in two different events: *specialize* and *generalize*). This permits a more fine-grained reasoning. Nevertheless, we can easily define a correspondence between both sets of events.

Table 4.1. Determining the relevant events for each model element

Element	Properties	Events	Comments
A class Cl	isConstant(Cl) or isAbstract(Cl)	\emptyset	
	isPermanent(Cl)	i Cl	
	Other	i Cl ,d Cl	
An attribute at of Cl	changeability(at ; Cl)=changeable and not isDerived(at ; Cl)	uat Cl	
	changeability(at ; Cl)=readOnly and not isDerived(at ; Cl)	uat Cl	Only possible just after a new Cl object is created
	isDerived(a ; Cl)	\emptyset	
An association $As(p_1:Cl_1,p_2:Cl_2)$	changeability(p_1 ; As) = changeable and changeability(p_2 ; As) = changeable and not isDerived(As)	i As ,d As	
	changeability(p_1 ; As) = readOnly and changeability(p_2 ; As) = changeable and not isDerived(As) (similarly to p_1 changeable and p_2 readOnly)	i As ,d As	i As only admitted immediately after the creation of a Cl_2 object. d As only after the removal of Cl_2 object
	isDerived(As)	\emptyset	
A generalization set $Gens(Cl_p,Cl_c)$	Always	s Cl_pCl_c , g Cl_cCl_p	
An association class $Ac(p_1:Cl_1,p_2:Cl_2)$	- i $AsCl$ can be applied when both iCl (considering the properties of the class facet of Ac) and iAs (considering its association properties) events could be applied. - d $AsCl$ can be applied when both dCl and dAs could be applied		

With the information provided in Table 4.1, we may determine (Table 4.2) the relevant events for the example of Fig. 3.1.

Table 4.2. Relevant structural events for the class diagram of Fig. 3.1

Element	Event
<i>Company</i>	iCompany, dCompany, uNameCompany
<i>Department</i>	iDepartment, dDepartment, uNameDepartment
<i>Investor</i>	iInvestor, dInvestor, uNameInvestor, uNationalityInvestor
<i>Employee</i>	uNameEmployee, uDateOfBirthEmployee, uSalaryEmployee, sEmployeeJunior, sEmployeeSenior
<i>Junior</i>	iJunior, dJunior, gJuniorEmployee
<i>Senior</i>	iSenior, dSenior, uExperienceSenior, gSeniorEmployee
<i>City</i>	\emptyset
<i>Manages</i>	iManages, dManages
<i>WorksFor</i>	iWorksFor, dWorksFor
<i>IsSupervisedBy</i>	iIsSupervisedBy, dIsSupervisedBy
<i>Owns</i>	iOwns, dOwns
<i>Investment</i>	iInvestment, dInvestment
<i>IsLocatedIn</i>	iIsLocatedIn, dIsLocatedIn

4.3 Specifying the Events' Dependencies

In the previous section, we have computed the events that the designer may apply to evolve the system state. However, when doing so, the user cannot freely choose any combination of events because, in general, the execution of an event ev requires or depends on the presence of other events in order to leave the data in a consistent state (i.e. in a state where all integrity constraints are satisfied).

This implies that if an event ev depends on a set of events $ev_1...ev_n$, every time ev is executed, then all other $ev_1...ev_n$ events must be applied as well. As an example, consider the event $iDepartment$ for creating departments in our running example. The execution of this event requires the creation of a new link in the *Manages* association relating the new department with its boss ($iManages$ event), the creation of a new link in the *Owns* association relating the new department with its company ($iOwns$ event) and the insertion of a new value for the *name* attribute of the department ($uNameDepartment$). Otherwise, every time this event is executed the minimum multiplicity of the *boss* end, *company* end and *name* attribute (see Fig. 3.1) becomes violated, and thus, the creation of the new department fails.

Therefore, computing the dependencies among the events is a key issue in ensuring the executability of the operations in the behavior schema.

We say that an event ev depends on an event ev' if executing ev but not ev' always leaves the system in an inconsistent state, i.e. in a state where at least one of the integrity constraints becomes violated. Note that, according to our definition of the *executability* property (beginning of section 4), applying both ev and ev' does not guarantee that the resulting system state will be always consistent with all constraints, it just guarantees that there exist at least an initial system state (and/or a set of arguments for the events' parameters) for which ev and ev' can be successfully applied.

Dependencies between structural events depend on the type of the event and on the integrity constraints of each particular class diagram. In fact, when computing the dependencies for an event ev we just need to focus on basic structural constraints as minimum multiplicity constraints for associations, *disjoint* and *complete* constraints and constraints related to the composite relationships of the diagram, either graphically represented or implicitly derived from the set of textual OCL constraints. For all other constraints, we can always find at least 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 cardinality constraints are never violated when ev is applied to an empty system state. Constraints restricting the value of the attributes of an object o may be satisfied when passing appropriate argument values as parameters for the events modifying o .

A dependency for a structural event ev is defined as a tuple $\langle direction, event \rangle$ where *event* is the name of the structural event required by ev and *direction* indicates whether that event should be executed before ev (symbol \leftarrow), after ev (symbol \rightarrow) or if the exact position of ev is irrelevant (symbol \uparrow). 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).

In what follows we list the dependencies for each kind of structural event.

Dependency rules for $iCl_1(x)$ events:

- IC1. A dependency $dep_{iCl_pAt_i} = \langle \rightarrow, uAt_iCl_1(x,v) \rangle$ for each non-derived attribute At_i of Cl_1 where $min(At_i, Cl_1) = 1$ AND
- IC2. A dependency $dep_{iCl_1At_i} = \langle \rightarrow, uAt_iCl_p(x,v) \rangle$ for each non-derived attribute At_i of Cl_p where Cl_p is a direct or indirect superclass of Cl_1 and where $min(At_i, Cl_p) = 1$ AND
- IC3. A number of $min(p_2; As_k)$ dependencies $dep_{iCl_1As_k} = \langle \rightarrow, iAs_k(x,y) \rangle$ for each non-derived association or association class $As_k(p_1:Cl_1, p_2:Cl_2)$ where Cl_1 has a mandatory participation (i.e, $min(p_2; As_k) \geq 1$) AND
- IC4. A number of $min(p_2; As_k)$ dependencies $dep_{iCl_pAs_k} = \langle \rightarrow, iAs_k(x,y) \rangle$ for each non-derived association or association class $As_k(p_1:Cl_p, p_2:Cl_2)$ where Cl_p is a direct or indirect superclass of Cl_1 and Cl_p has a mandatory participation in As_k

As an example, consider the application of these dependencies over the relevant event $iJunior(x)$ that inserts a junior employee x (Fig. 3.1). Rule IC1 does not generate any dependency since the *Junior* class has no attributes. Rule IC2 determines that the events $uNameEmployee(x,vname)$, $uDateOfBirthEmployee(x,vbirth)$ and $uSalaryEmployee(x,vsal)$ are required to initialize the name, the birth date and the salary of the new junior employee since those attributes are not optional (i.e, $min(name;Employee)=1$, $min(dateOfBirth;Employee)=1$ and $min(salary;Employee)=1$). Besides, an event $iIsSupervisedBy(x,y)$ to link the new junior with an existing senior employee y is needed to maintain the multiplicity of the *IsSupervisedBy* association, as stated by rule IC3. Summing up, the execution of the event $iJunior(x)$ requires the execution of $uNameEmployee(x,vname)$, $uDateOfBirthEmployee(x,vbirth)$, $uSalaryEmployee(x,vsal)$ and $iIsSupervisedBy(x,y)$ events as well.

Dependency rules for $dCl_1(x)$ events:

- DC1. A number of $min(p_2; As_k)^4$ dependencies $dep_{dCl_1As_k} = \langle \rightarrow, dAs_k(x,y) \rangle$ for each non-derived association⁵ or association class $As_k(p_1:Cl_1, p_2:Cl_2)$ where Cl_1 has a mandatory participation AND
- DC2. A number of $min(p_2; As_k)$ dependencies $dep_{dCl_1As_k} = \langle \rightarrow, dAs_k(x,y) \rangle$ for each non-derived association or association class $As_k(p_1:Cl_p, p_2:Cl_2)$ where Cl_p is a direct or indirect superclass of Cl_1 and Cl_p has a mandatory participation in As_k AND
- DC3. A number of $min(p_2; As_k)$ dependencies $dep_{dCl_1Comp1} = \langle \rightarrow, dCl_2(y) \rangle$ for each non-derived association or association class $As_k(p_1:Cl_1, p_2:Cl_2)$ where Cl_1 has a mandatory participation and $aggregationKind(p_1; As_k)=composite$ AND
- DC4. A number of $min(p_2; As_k)$ dependencies $dep_{dCl_1Comp2} = \langle \rightarrow, dCl_2(y) \rangle$ for each non-derived association or association class $As_k(p_1:Cl_p, p_2:Cl_2)$ where Cl_p has a

⁴ At design time we do not know the number of links of As in which x participates, and thus, we cannot exactly determine how many dAs events should be applied to remove all links of x in As . However we know that at least $min(Cl_1, As_k)$ should be applied.

⁵ Deletion of links in derived association ends should be automatically generated by the software component in charge of enforcing the association derivation rule

mandatory participation and it is a direct or indirect superclass of Cl_1 and $aggregationKind(p_1; As_k)=composite$

In our running example, the event $dJunior(x)$ that removes a junior employee x requires, by rule DC1, a posterior event $dIsSupervisedBy(x,y)$ to remove the existing link between the junior employee and his/her supervisor. The event $dCompany$ induces, among others, the deletion of the related departments (at least one) due to the semantics of the composite relationship between them.

Dependency rules for $sCl_p Cl_c(x)$ events:

- SC1. A number of $min(At_i, Cl_c)$ dependencies $dep_{SpecAti} = \langle \rightarrow, uAt_i Cl_c(x,y) \rangle$ for each non-derived attribute At_i of Cl_c where $min(At_i, Cl_c) = 1$ AND
- SC2. A number of $min(p_2; As_k)$ dependencies $dep_{SpecAsk} = \langle \rightarrow, iAs_k(x,y) \rangle$ for each non-derived association or association class $As_k(p_1:Cl_c, p_2:Cl_2)$ where Cl_c has a mandatory participation AND
- SC3. A dependency $dep_{Spec} = \langle \rightarrow, gCl_c Cl_p(x) \rangle$ for a Cl_c' class such that $Cl_c' \neq Cl_c$. Cl_c' .⁶ 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 its previous subtype Cl_c' to satisfy the disjoint constraint. We know that x was instance of some Cl_c' because the generalization set is complete.

Consider the event $sEmployeeJunior(x)$ that specializes an employee x as junior employee. The rule SC1 does not generate any dependency since the *Junior* class has no attributes. Due to rule SC2, the structural event $iIsSupervisedBy(x,y)$ is required to insert a link between the specialized junior employee and an existing senior employee y . And finally, considering rule SC3, $gSeniorEmployee(x)$ event is required. This event is necessary to avoid violating the disjointness constraint (employees cannot be junior and senior at the same time).

Dependency rules for $gCl_c Cl_p(x)$ events:

- GC1. A number of $min(p_2; As_k)$ dependencies $dep_{GenAsk} = \langle \rightarrow, dAs_k(x,y) \rangle$ for each non-derived association or association class $As_k(p_1:Cl_c, p_2:Cl_2)$ where Cl_c has a mandatory participation AND
- GC2. A number of $min(p_2; As_k)$ dependencies $dep_{GenAsk} = \langle \rightarrow, dCl_2(y) \rangle$ for each non-derived association or association class $As_k(p_1:Cl_p, p_2:Cl_2)$ where Cl_1 has a mandatory participation and $aggregationKind(p_1; As_k)=composite$ AND
- GC3. A dependency $dep_{Gen} = \langle \rightarrow, sCl_p Cl_c(x) \rangle$ for a Cl_c' class 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.

The dependencies for the event $gJuniorEmployee(x)$ that generalizes a junior employee x as employee are the event $dIsSupervisedBy(x,y)$ (rule GC1) and

⁶ When the generalization has more than two subtypes, Cl_c' should be the specific subclass verifying that $x.oclIsTypeOf(Cl_c') = true$.

sEmployeeSenior(x) (rule GC3). This second event is necessary since if x is an employee then x must be either junior or senior employee in order to satisfy completeness constraint. Therefore, if x is generalized from junior to employee, it must be necessarily specialized as senior employee.

Dependency rules for $iAs(x,y)$ events for an association $As(p_1:Cl_1, p_2:Cl_2)$ when $min(p_2; As) = max(p_2; As)$ or $changeability(p_2;As)=readOnly$ (the process must be symmetrically repeated for the other participant when $min(p_1; As) = max(p_1; As)$ or $changeability(p_1;As)=readOnly$) are:

IA1. A dependency $dep_{iAs} = \langle \hat{\uparrow}, dAs(x,z) \rangle$ if $min(p_1; As) \neq max(p_1; As)$ and $changeability(p_2; As) = changeable$ and $changeability(p_1; As) = changeable$ and $aggregationKind(p_2; As) \neq composite$ OR

IA2. A dependency $dep_{iAsCl_1} = \langle \leftarrow, iCl_1(x) \rangle$ if not $isConstant(Cl_1)$ or not $isAbstract(Cl_1)$ OR

IA3. A dependency $dep_{iAsSpec} = \langle \leftarrow, sCl_p Cl_1(x) \rangle$ if Cl_1 has a supertype Cl_p

The event $iIsSupervisedBy(x,y)$ that inserts a link between the junior x and the senior employee y requires either, by rule IA1, the deletion of the existing link ($dIsSupervisedBy(x,z)$) or, by rule IA2, the previous insertion of a new junior employee x ($iJunior(x)$) or, by rule IA3, the specialization of the employee x into junior ($sEmployeeJunior(x)$). Otherwise, the insertion of the new link will cause a junior to have two supervisors, violating this way the '1' multiplicity in $IsSupervisedBy$.

Dependency rules for $dAs(x,y)$ events for an association $As(p_1:Cl_1, p_2:Cl_2)$ when $min(p_2; As) = max(p_2; As)$ or $changeability(p_2;As)=readOnly$ (the process must be symmetrically repeated for the other participant when $min(p_1; As) = max(p_1; As)$ or $changeability(p_1;As)=readOnly$) are:

DA1. A dependency $dep_{dAs} = \langle \hat{\uparrow}, iAs(x,z) \rangle$, if $min(p_1; As) \neq max(p_1; As)$ and $changeability(p_2; As) = changeable$ and $changeability(p_1; As) = changeable$ and $aggregationKind(p_2; As) \neq composite$ OR

DA2. A dependency $dep_{dAsCl_1} = \langle \leftarrow, dCl_1(x) \rangle$ if $isConstant(Cl_1)=false$ and $isPermanent(Cl_1)=false$ OR

DA3. A dependency $dep_{dAsGen} = \langle \leftarrow, gCl_1 Cl_p(x) \rangle$ if Cl_1 has a supertype Cl_p

The event $dIsSupervisedBy(x,y)$ that deletes a link between the junior x and the senior employee y requires either the insertion of a new link for the junior employee x (rule DA1) or the deletion (rule DA2) or generalization (rule DA3) of the junior employee x in order to maintain the multiplicity of the association.

Dependency rules for $iAsCl(x:Cl, y_1:Cl_1, y_2:Cl_2)$ events are the union of the dependencies for iCl and iAs events.

Dependency rules for $dAsCl(y_1:Cl_1, y_2:Cl_2)$ events are the union of the dependencies for dCl and dAs events.

For instance, in the example of Fig. 3.1 the insertion of an investment ($iInvestment(x,y_1,y_2)$) requires the event $uNameInvestment(x,vname)$, as stated by rule IC1.

UAt:CI(x,v) events do not generate new dependencies since changes on attribute values neither violate cardinality, complete nor disjoint constraints.

The following tables summarize the result of the application of the previous dependency rules over the relevant structural events for our running example.

Table 4.3. Dependencies for relevant iCl events in the class diagram of Fig. 3.1

Struct. event	Rule	Required events (dependencies)
iJunior(x)	IC2	uNameEmployee(x,vname) AND uDateOfBirthEmployee(x,vbirth) AND uSalaryEmployee(x,vsal)
	IC3	ilsSupervisedBy(x,y)
iSenior(x)	IC1	uExperienceSenior(x,vexp) AND
	IC2	uNameEmployee(x,vname) AND uDateOfBirthEmployee(x,vbirth) AND uSalaryEmployee(x,vsal)
iDepartment(x)	IC1	uNameDepartment(x,vname) AND
	IC3	iManages(y,x) AND iOwns(x,z)
iCompany(x)	IC1	uNameCompany(x,vname) AND
	IC3	iOwns(y,x) AND iLocatedIn(x,z)
iInvestor(x)	IC1	uNameInvestor(x,vname) AND uNationalityInvestor(x,vnationality) AND
	IC3	iInvestment(t,y,x)

Table 4.4. Dependencies for relevant dCl events in the class diagram of Fig. 3.1

Struct. event	Rule	Required events (dependencies)
dJunior(x)	DC1	dIsSupervisedBy(x,y)
dSenior(x)	-	∅
dDepartment(x)	DC1	dManages(y,x) AND dOwns(x,z)
dCompany(x)	DC1	dOwns(y,x) AND dLocatedIn(x,z) AND
	DC3	dDepartment(y)
dInvestor(x)	DC1	dInvestment(y,x)

Table 4.5. Dependencies for relevant sClpClc(x) events in the class diagram of Fig. 3.1

Struct. event	Rule	Required events (dependencies)
sEmployeeJunior(x)	SC2	ilsSupervisedBy(x,y) AND
	SC3	gSeniorEmployee(x)
sEmployeeSenior(x)	SC1	uExperienceSenior(x,vexp) AND
	SC3	gJuniorEmployee(x)

Table 4.6. Dependencies for relevant gClcClp(x) in the class diagram of Fig. 3.1

Structural event	Rule	Required events (dependencies)
gJuniorEmployee(x)	GC1	dIsSupervisedBy(x,y) AND
	GC3	sEmployeeSenior(x)
gSeniorEmployee(x)	GC3	sEmployeeJunior(x)

Table 4.7. Dependencies for relevant *iAs* in the class diagram of Fig. 3.1

Struct. event	Rule	Required events (dependencies)
iManages(x,y)	IA1	dManages(z,y) OR
	IA2	iDepartment(y)
iWorksFor(x,y)	-	∅
iIsSupervisedBy(x,y)	IA1	dIsSupervisedBy(x,z) OR
	IA2	iJunior(x) OR
	IA3	sEmployeeJunior(x)
iOwns(x,y)	IA2	iDepartment(x)
iLocatedIn(x,y)	IA1	dLocatedIn(x,z) OR
	IA2	iCompany(x)

Table 4.8. Dependencies for relevant *dAs* events in the class diagram of Fig. 3.1

Struct. event	Rule	Required events (dependencies)
dManages(x,y)	DA1	iManages(z,y) OR
	DA2	dDepartment(y)
dWorksFor(x,y)	-	∅
dIsSupervisedBy(x,y)	DA1	iIsSupervisedBy(x,z) OR
	DA2	dJunior(x) OR
	DA3	gJuniorEmployee(x)
dOwns(x,y)	DA2	dDepartment(x)
dLocatedIn(x,y)	DA1	iLocatedIn(x,z) OR
	DA2	dCompany(x)

Table 4.9. Dependencies for *iAsCl* and *dAsCl* in the class diagram of Fig. 3.1

Structural event	Rule	Required events (dependencies)
iInvestment(t,x,y)	IC1	uCapitalInvestment(t,vcapital)
dInvestment(x,y)	-	∅

4.4 Generating the system operations

Given the information about the relevant events for a class diagram *CD* and their dependencies, we define in this section how to automatically generate a complete and executable behavior schema for *CD*. Among all possible behavior schemas, our goal is to generate a basic one, that is, a behavior schema where all operations are necessary and where each operation is kept as simple as possible (with *simple* understood as “with the minimal number of structural events”). In this way we favor the readability and usability of the schema. Arbitrary complex operations can be later on added by the designer (see Section 5).

The generation of this behavior schema consists of three main steps:

- Identifying the required system operations (*completeness*).
- Computing the set of structural events involved in each operation (*executability*).
- Defining the final signature and body for the operations.

4.4.1 Determining the system operations

Clearly, assignment of relevant events into a set of system operations can be done in many different ways as long as each event is included in at least one operation (to ensure the *completeness* of the schema).

Nevertheless, since we intend to create a basic behavior schema, we have the additional goal of minimizing the complexity of the generated operations.

Therefore, we fulfill completeness by creating a different operation in *CD* for each relevant event except for those events that can only be applied under certain circumstances (e.g. at particular time instants or immediately after/before of other events), as identified in the *comments* column in Table 4.1. It does not make sense to create system operations for them since the user will not be able to execute those operations at his/her will. All these events will be included in other operations as part of the dependencies computation.

Following these premises, given that set_{ev} is the set of relevant events for *CD*, the system operations our method generates are the following (each operation is assigned to the most 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 class operation $AsCl::create$ for each $iAsCl$ event in set_{ev}
- A $AsCl::delete$ operation for each $dAsCl$ event in set_{ev}
- A $Cl_c::generalizeCl_p$ operation for each gCl_cCl_p event in set_{ev}
- A $Cl_p::specializeCl_c$ operation for each sCl_pCl_c event in set_{ev}
- A $Cl::updateAt_i$ operation for each uAt_iCl event in set_{ev} such that $changeability(At_i;Cl)=changeable$ and or, alternatively, a single $Cl::update$ operation for all uAt_iCl events in set_{ev} corresponding to changeable attributes.
- A $Cl_1::createLinkAs^8$ operation for the participant class Cl_1 in $As(p_1:Cl_1, p_2:Cl_2)$ for each iAs event in set_{ev} providing that $isNavigable(p_2;As)$ and $changeability(p_2;As)=changeable$
- A $Cl_1::createLinkObjectAsCl$ operation for the participant class Cl_1 in $AsCl(p_1:Cl_1, p_2:Cl_2)$ for each $iAsCl$ event in set_{ev} providing that $isNavigable(p_2;As)$ and $changeability(p_2;As)=changeable$
- A $Cl_1::deleteLinkAs$ operation for the participant class Cl_1 in $As(p_1:Cl_1, p_2:Cl_2)$ for each dAs event in set_{ev} providing that $isNavigable(p_2;As)$ and $changeability(p_2;As)=changeable$
- A $Cl_1::deleteLinkObjectAsCl$ operation for the participant class Cl_1 in $AsCl(p_1:Cl_1, p_2:Cl_2)$ for each $dAsCl$ event in set_{ev} providing that $isNavigable(p_2;As)$ and $changeability(p_2;As)=changeable$

The application of these rules on our running example generates the extended schema of Fig. 4.1

⁷ In UML, operations cannot be assigned to associations except for association classes. Therefore, operations on associations are assigned to the participants of the association

⁸ For recursive associations we can use the name of the opposite role rather than the association name

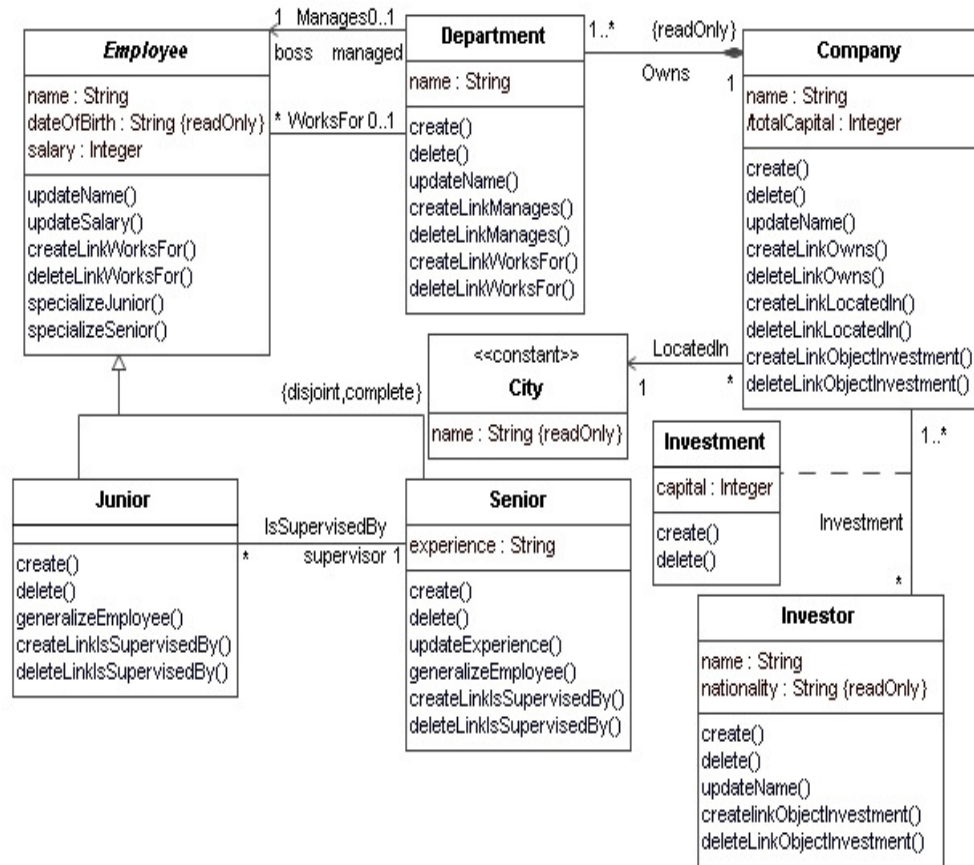


Fig. 4.1. Class diagram with the generated system operations for the running example

4.4.2 Computing the set of events for each operation

Initially, the previous operations just contain the single structural event that has induced the creation of the operation (i.e. the body of a $Cl::create$ operation only includes the event iCl). Nevertheless, as seen in section 4.3, *executability* is only guaranteed when dependencies among the events are considered.

As a consequence, the body of some of the operations must be extended with new events in order to satisfy those dependencies. For instance, the operation *createDepartment* (initially including just the $iDepartment$ event) must also be in charge of creating a new link in the *Manages* association relating the new department with its boss ($iManages$ event) and a new link in the *Owns* association relating the new department with its company ($iOwns$ event).

The computation of the final set of events for an operation op initially consisting of a single event ev is done by determining the *transitive closure* of the dependencies of ev . This is a recursive process. We start by adding to op the events $ev_1 \dots ev_n$ that ev depends on. Then, we take into account the dependencies of ev_1 (and $ev_2, ev_3, \dots ev_n$).

Otherwise an operation including ev and ev_1 may be non-executable due to dependencies of ev_1 not satisfied within the operation. Note that, if an event ev_i in op depends on an event ev_j that is already part of op the dependency is considered satisfied and ev_j it is not added to op again. The process continues until the dependencies for all events in the operation body are satisfied.

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 original one and then the process continues with each list separately. Each list generates a different operation specification. When processing *OR* dependencies we must: 1- select the alternative (if any) that terminates the recursive process (i.e. the one with events that are already part of the operation and thus the one that does not generate additional dependencies) and, in that case, discard the other options and 2 - prune the alternatives that do not preserve the operation effect [36] (i.e. alternatives that neutralize the effect of another event in the operation).

Termination is guaranteed except for cyclic sequences of associations with rare multiplicities combinations (as a cyclic sequence of exact one-to-one associations), which require the designer to take part in the process to avoid an infinite loop.

As an example, consider an operation with the event $gJuniorEmployee$. Dependencies for this event are the deletion of the $IsSupervisedBy$ link ($dIsSupervisedBy$), rule GC1, and the specialization of the generalized junior employee as new senior ($sEmployeeSenior$), as stated by rule GC2. In its turn, $dIsSupervisedBy$ has an *OR* dependency (rules DA1, DA2 and DA3), and thus, an event $iIsSupervisedBy$, $dJunior$ or $gJuniorEmployee$ is required. This later alternative is chosen to stop the recursive process. $sEmployeeSenior$ requires (rule SC1) adding the event $uExperienceSenior$. The process is now completed since this later event does not generate new dependencies to be satisfied.

4.4.3 Final operation specification

In an imperative specification, the final body for the operation is just the ordered list $list_{ev}$ of structural events the operation consist of (computed as shown in the previous section).

The signature of the operation also depends on the events in $list_{ev}$. Each event $ev \in list_{ev}$ may induce 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. Object variables for iCl and $iAsCl$ 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 in non-static operations 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. Similarly for *iAsCl* and *dAsCl* events.

For instance, the list of events for the operation `Junior::create` (Fig 4.1) is *iJunior(x)*, *uNameEmployee(x,vname)*, *iIsSupervisedBy(x,y)*, *uSalaryEmployee(x,vsal)* and *uDateOfBirthEmployee(x,vbirth)*. Its body and signature are defined as follows:

```
Junior::create(vname:String,      vbirth:Date,      vsal:Integer,
y:Senior){
    iJunior(x);
    uNameEmployee(x,vname);
    uDateOfBirthEmployee(x,vbirth);
    uSalaryEmployee(x,vsal);
    iIsSupervisedBy(x,y);}
```

This basic operation generation process suffices to guarantee the executability of our operations. Remember that in our approach, an operation is executable if there is at least an initial system state over which the operation can be successfully executed. Our approach does not pretend to guarantee that all possible executions of the operation will be successful. Nevertheless, for some operations our method can easily improve their applicability (i.e. the number of scenarios where the operation can be successfully applied) by means of adding simple conditional and iterator expressions. Consider for instance the operation *Company::deleteCompany* (events *dCompany*, *dLocatedIn*, *dOwns* and *dDepartment*):

```
Company::delete(){
    dLocatedIn(self,self.city);
    dOwns(self,self.department);
    dDepartment(self.department);}
dCompany(self);}
```

This operation is executable. For system states where the deleted company has a single department, the operation will leave the system in a consistent state. Nevertheless, the following alternative version of this operation:

```
Company::delete(){
    dLocatedIn(self,self.city);
    for each department d in self.department
        dOwns(self,d);
        dDepartment(d);}
endfor
dCompany(self);}
```

extends its applicability. Now the operation can be successfully executed regardless the number of departments (at least one) related to the company. Note that this extension for removing all links of an object being deleted is not necessary when working with declarative operations since there the removal of all links of a deleted object is, in general, implicitly assumed [37].

It may happen that depending on the structure of the class diagram two different operations end up with the same body. To avoid redundancies we can remove one of them in the final behavior schema.

4.5 Declarative Specification

There are two different approaches for specifying the effect of an operation on the system state: the previous *imperative* approach or the alternative *declarative* approach. From a specification point of view, the declarative approach is preferable since it allows a more abstract and concise definition of the operation effect [42]. Therefore, designers may prefer to view the generated operations in a declarative form. The goal of this section is to extend our method to cope with the generation of declarative operations by automatically transforming the previous imperative definitions.

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.

Therefore, 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. When defining the contract we will assume the implicit “nothing else change” semantics [5]. This common assumption allows us to avoid specifying all parts of the system that do not change during the operation.

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 [5]). In the following we provide a possible boolean condition for each event.

1. $iCl(x):x.ocllsNew()$ and $x.ocllsTypeOf(Cl)$
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. $gCl_cCl_p(x):not x.ocllsTypeOf(Cl_c)$
7. $sCl_pCl_c(x): x.ocllsTypeOf(Cl_c)$
8. $iAsCl(x:Cl, y_1:Cl_1, y_2:Cl_2): x.ocllsNew()$ and $x.ocllsTypeOf(Cl)$ and $x.cl_1=y_1$ and $x.cl_2=y_2$
9. $dAsCl(y_1:Cl_1, y_2:Cl_2): OclAny::allInstances()=OclAny::allInstances()@pre->excluding(asCl::allInstances@pre()->select(x|x.cl_1=y_1 and x.cl_2=y_2))$

Note that *OclAny* is the supertype of all types in the UML class diagram [28]. 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*). Also, for gCl_cCl_p we do not

explicitly state that x must remain instance of Cl_p . This is implicitly assumed since, before the event, x was instance of Cl_p and the translation does not state otherwise.

The resulting postcondition may need to be refined depending on the combination of translated 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 $dCl_1(x_1)$ and/or $dCl_2(x_2)$ events also appear [37].

The signature of the operation does not change.

As an example, we provide the contract for the operation *Junior::create* and *Department::delete*.

```
context Junior::create(vname:String, vbirth:Date, vsal:Integer,
y:Senior)
post: x.oclIsNew() and x.oclIsTypeOf(Junior) and x.name=vname
and x.dateOfBirth=vbirth and x.salary=vsal and
x.supervisor-> includes(y)
```

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

All the declarative operation contracts for the running example can be found in the Appendix.

5 Designer-defined Operations

It may happen that designers need to define additional (more complex) operations besides the ones automatically generated with our method. The aim of this section is to show how our method can be easily reused to also ensure the correctness of arbitrary complex designer-defined operations.

To start the process, the designer must inform about the intended effect of the operation op he/she wants to define and the class that will hold op (for complex operations it is not possible to automatically determine its best context type). The effect of op is provided as a list $list_{ev}$ of structural events.

Then, the full (and correct) definition of the operation op is created as follows:

- The computation of the dependencies presented in Section 4.3 is applied to each⁹ event in $list_{ev}$. As a result we obtain an extended list $list'_{ev}$. Note that, in this case, $list'_{ev}$ is initialized with all events in $list_{ev}$ before the computation process. Therefore, an event dependency that has already been covered in the initial list of events provided by the designer does not generate new events in $list'_{ev}$.
- When there are several possible options for $list'_{ev}$ (due to the existence of OR dependencies), the designer selects the one he/she prefers. The process

⁹ We assume that the list of events provided by the designer is not contradictory (for instance, the list does not include an event to update an object and then another to delete it).

can be repeated for the other options to generate alternative operation definitions if so desired.

- The parameters and the body of *op* are determined according to the process defined in Sections 4.4.3 using $list'_{ev}$ as the set of events to consider.

Some of these operations may be used as a replacement of other (more basic) ones. To ensure completeness, a basic operation *op* can only be removed after creating a designer-defined operation *op'* iff *op'* subsumes *op* that is, if all events that define the effect of *op* appear in $list'_{ev}$.

As an example, consider an operation *Junior::AssignSenior* that associates a senior supervisor to a junior employee and updates (increases) the salary of the supervisor as a compensation for the supervision time. Therefore, $list_{ev}$ for this operation consists of two events: $\{iIsSupervisedBy, uSalaryEmployee\}$.

The computation of the dependencies for *uSalaryEmployee* returns an empty list. For *iIsSupervisedBy* we have an OR dependency, and thus, the designer must choose the preferred alternative between the *dIsSupervisedBy* (deleting an existing link between the junior employee and a previous senior supervisor) and the *iJunior* events (creating a new instance of a junior employee to be assigned to the supervisor).

Assuming that the designer prefers the *dIsSupervisedBy* event, the final $list'_{ev}$ is $\{iIsSupervisedBy(x,y), uSalaryEmployee(y,vsal), dIsSupervisedBy(x,z)\}$.

Given this list of events, the (declarative) body of *AssignSenior* becomes:

```
context Junior::AssignSenior(vsen:Senior, vsal:Integer)
post: self.supervisor->includes(vsen) and vsen.salary=vsal and
self.supervisor->excludes(self.supervisor@pre)
```

Once this new operation is defined the designer may want to remove the generated basic operations *CreateLinkIsSupervisedBy()* from the *Senior* and *Junior* classes and replace them with the new *AssignSenior* one.

6 Case Studies

To illustrate the benefits of our proposal, we automatically generated with our method the behavior schema of three real-life applications and compare the results we get with the behavior schema originally specified by the designer for the same application by hand. In particular, we analyze the *osCommerce* system for online stores as specified in [41], a conference management system (CMA) [34] and an employee training management system that has been developed for a car industry company (EmpTrainig) [16].

Table 6.1 summarizes the results. Columns *NCl*, *NAs*, *NG*, *NAt* show the number of classes, associations, generalization sets and attributes of each case study. *NOp* records the number of modification operations in the behavior schema as defined by the designer. Column *T* indicates how many operations of those in *NOp* are directly generated by our method. Column *P* describes the number of operations that are only partially generated (i.e. the specification of these operations should be manually

completed by the designer). Mainly, the difference is that in the original schema these 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 with our method. Operations in $NOp - T - P$ are arbitrary complex operations not directly covered by our method. Nevertheless, these operations could be defined in an assisted way as seen in Section 5.

We would like to remark that the results we get with our method are quite satisfactory since it considerably reduces the number of operations to be defined by designers (in a 69% for *EmpTraining* and CMA and in a 74% for *osCommerce*). Moreover, our method also provides a high percentage of partial contracts for the rest of the operations (21% for *EmpTraining*, 31% for CMA and 7% for *osCommerce*).

As an additional benefit, our method generates some operations that did not appear in the manually specified schemas. Designers could use this information to detect whether some of the required operations are missing in their schema or the specification of the schema is incomplete (for instance, some attributes may need to be marked as read only, some generalization sets as disjoint and complete and so forth). As an example, for the CMA system our method generates all the necessary *generalizeCl* and *specializeCl* and some *updateAt_i* operations that were not present (by mistake) in the original schema defined by the designer.

Table 6.1. Results of applying our method to three case studies

	Size of the CS					Results				
	NCI	NAs	NG	NAt	NOp	T	%T	P	%P	%T+P
osCommerce	61	69	10	181	163	121	74	12	7	81
CMA	13	13	2	49	29	20	69	9	31	100
EmpTraining	21	24	0	103	90	62	69	19	21	90

7 Prototype Tool

We have developed a prototype version of a tool for the automatic generation of a basic behavior schema from a UML class diagram.

The tool accepts as an input a UML class diagram defined using Poseidon[®] 5.0.1 [17] and exported as a XMI file, and generates a set of complete and executable system operations. For each operation, both declarative and imperative versions are provided. Currently, our tool supports a representative subset of features of the model elements that may be represented by Poseidon[®] 5.0.1.

As an example, Figure 7.1 shows the declarative and imperative operation specification for the *Department::create* operation (Fig. 7.1).

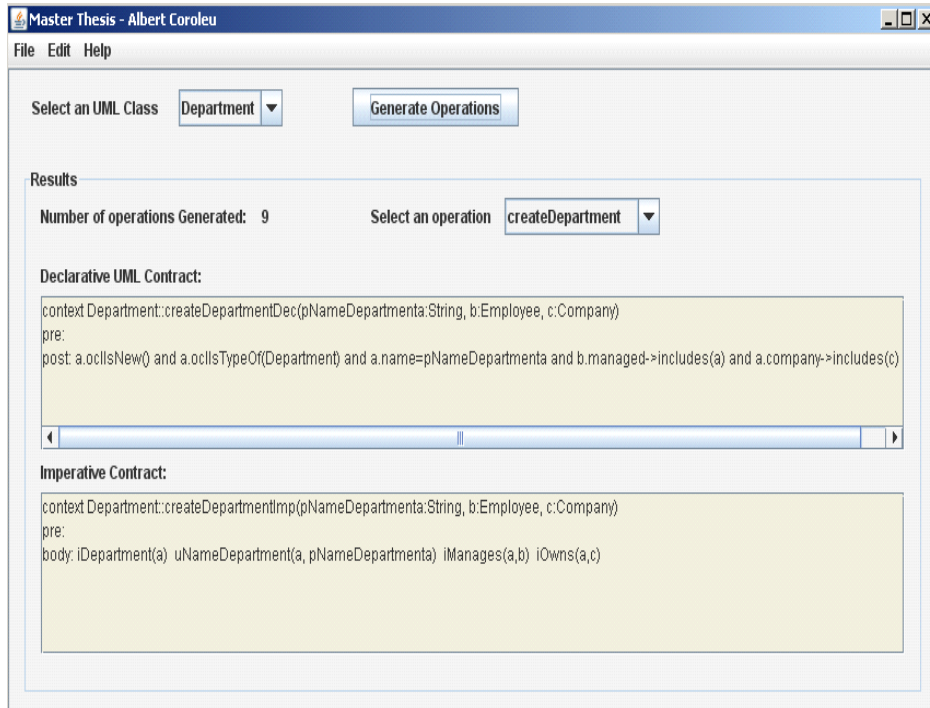


Fig. 7.1. Generation of *Department* operations

8 Related Research

Related approaches can be categorized into two groups. On the one hand, there are several methods providing code generation facilities from UML models and that may include the creation of basic modification operations as part of their code-generation phase. On the other hand, there may be proposals working at a conceptual level (as our own proposal) aimed to the automatic specification of behavior schemas from other UML (or ER, B,...) diagrams.

In what follows we compare our method with the most representative approaches of both groups.

8.1 Comparison with MDD methods

Current MDD methods target specific programming languages and/or technology platforms and thus, their results are hardly reusable to other technologies. Instead, working at the conceptual level, our method can be used regardless the final technology platform where the system is going to be implemented.

Moreover, the expressivity supported by our method is higher than that of current MDD methods, which assume simpler class diagrams (usually, derived elements, read only properties, disjoint and complete constraints and association classes among other constructs are not fully admitted) as a starting point of the code-generation process. This results in a less precise and usable operation specification. Besides, none of them provides any kind of partial support for the definition of complex operations.

Among the most popular MDD methods we have:

- *Executable UML* methods [25] [20] propose to limit the expressiveness of the UML language to an executable language subset. However, these methods force the designer to explicitly define the body of the operations using the proprietary action language they provide. No automatic operation generation is performed.
- OptimalJ [11] and ArcStyler [19] are well-known MDD environments for Java. Both follow the MDA architecture and use a pattern-based approach for their code-generation process. Adaptations for developing extensions for tackling custom platforms are also possible but no generation facilities as the ones presented herein are included.
- *AndroMDA* [3] is an open source code generation framework. Models from UML tools are transformed into deployable components of platforms as J2EE, Spring or .NET. AndroMDA generates blank methods for implementing the business logic of the application but does not assist in the automatic generation of the body of those methods.
- *OO-Method* [33] generates system operations somewhat similar to those defined in the paper but using a concrete implementation language instead of a more abstract one like OCL.
- Many popular CASE tools (as *Together* [4], *Poseidon* [17], *Objecteering/UML* [30], *Rational Rose* [18]) offer some kind of code-generation capabilities, specially for the Java language. However, for the behavior part, they basically map the specified signature of the operations in the class diagram to the signature in the corresponding Java class. A relevant exception is *Objecteering/UML* which generates skeletons for the update, create and destroy operations of each class and uses a predefined library to check that cardinality constraints are not violated. All other possible dependencies are ignored. The UML constructs that may appear in the initial class diagram is also limited.
- Some IDEs (as [10] or [26]) are able to automatically generate basic getter/setter and creator methods for classes. However, they do not take into account the possible dependencies among the events and thus the executability of the operations is not guaranteed.

8.2 Comparison with other conceptual-level methods

The idea of dependencies between structural events stems from the (deductive) database field as part of the more general problem of integrity maintenance at compile-time (see [36], [23], [32]). There, the goal was similar: to extend a

(predefined) given transaction/operation with additional events to ensure its successful execution. However, their expressivity regarding the definition of the structural diagram and the set of admitted structural event types is much more restricted than in our method (due to the different language used but also because they tried to guarantee that an operation is *always* executable, which requires limiting the expressivity of the input model in order to achieve a result). Moreover, in those methods, the initial operations must be defined by the designers, they are not automatically derived. Also, completeness was not addressed.

Other closed approaches are [21], [15] and [12] aimed at partially generating behavior schemas from the information in the class diagram. All of them restrict the expressivity of the input class diagram.

[21] partially determines the set of possible structural events to be applied to a class diagram (generalization sets are not considered). However, in this approach operations must be manually defined as a combination of a set of those structural events. Therefore, the completeness and executability of these operations must be guaranteed by designers. Operations must be specified using the formal notation B [1].

[15] 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.

[12] generates a set of structural events to be applied to a structural model with dynamic features expressed in the ROSES language. Generalization sets, association classes and navigability are not considered.

Alternatively, other approaches try to generate system operations from the information provided in different diagrams, such as the use case diagram. For instance, [38] 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.

Our work is also related to existing work on constraint checking techniques. These techniques are aimed at preventing all constraint violations at the end of the operation execution. For instance, tools for run-time constraint checking (e.g. Octopus [39]) for the OCL language) extend the operation body with a set of *if-then* conditions that raise an exception if some constraint is not satisfied. Other tools try to formally prove at design time that the operation cannot possibly violate any constraint (it is *safe* wrt to the constraints). However, these techniques are unable to generate the operations themselves. The designer has to manually specify them. Moreover, they do not focus on detecting whether an operation is *never* executable. For instance, if an operation *op* is non-executable, when using run-time checking techniques, the execution of *op* will fail at every execution (since *op* is non-executable at least one of the added if-then clauses will fail which will raise an exception) but the designer will not know if *op* has just failed because the input state is not a suitable one for *op* or because there is not any state in which *op* can be successfully applied (i.e. *op* is not executable)

Our method assumes that the input class diagram is *satisfiable* (i.e. it is possible to create valid instances of the diagram). It is not possible to create executable operations for non-satisfiable diagrams. There are several approaches that permit to prove the satisfiability of a UML class diagram (e.g. [6]) before starting with our generation process.

9 Conclusions and Further Research

MDD approaches demand techniques and tools that automate and/or facilitate the different steps of the software development process. The method proposed in this work contributes to this line of research by automating part of the conceptual modeling phase, in particular, the definition of the dynamic aspects of the conceptual schema. Our method facilitates this task by automatically generating an initial set of system operations that users can execute to evolve the system state..

Operations are drawn from the static aspects of the domain as defined in the class diagram and take into account possible dependencies among them to ensure the completeness and executability of the operations.

This automatic generation simplifies the conceptual modeling phase and improves the quality of the obtained conceptual schema. Moreover, we believe our method can be helpful 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 contract postconditions in order to prevent constraint violations. These predicates are determined based on the information provided by our dependency computation process.

This method can be easily integrated as an additional feature in any extensible UML CASE tool. In particular, as a further work we plan to transform our prototype tool implementing the work presented in this paper into an Eclipse plug-in to be integrated into the new MOSKitt Open Source CASE tool [27].

Additionally, we are interested in strengthen the semantics of our characterization of the executability property introduced in this paper to increment the practicability of our method. In this sense we would like to integrate in our operation generation process the *efficient* verification of *all* constraints that may be violated by the operation execution (the relevant constraints can be determined with techniques as constraint checking [8] into the preconditions of our generated operation contracts. Then, a successful execution of the operation could be *always* guaranteed (providing that the system state and the operation parameters satisfy the precondition).

Other future lines of research are studying how we can reuse the information of use cases (as in [38]) and state diagrams to automatically derive more complex system operations, how to apply the completeness and executability properties to the verification of existing behavior schemas and, last but not least, how to incrementally regenerate the behavior schema after evolutions on the the structural schema (i.e. addition of new attributes, changes on the cardinalities, ...) .in order to improve the system's maintenance.

Acknowledgments

We would like to thank the people of the GMC group for helpful discussions and comments on previous drafts of this paper and to Joan Fons for assisting us in analyzing EmpTrainig system. We are also grateful to the anonymous referees (of the journal and of the MoDELS'07 conference) for their interesting suggestions. This

work has been partially supported by the Ministerio de Ciencia y Tecnología under project TIN2008-00444.

References

1. Abrial, J. R.: The B-book: Assigning programs to meanings. Cambridge University Press, (1996)
2. Albert, M., Pelechano, V., Fons, J. et al.: Implementing UML Association, Aggregation, and Composition. A Particular Interpretation Based on a Multidimensional Framework. In: Advanced Information Systems Engineering - CAiSE'03, vol. 2681, pp. 143-158. (2003)
3. AndroMDA: AndroMDA. <http://www.andromda.org/>
4. Borland: Together. <http://www.borland.com/us/products/together/index.html>
5. Cabot, J.: From Declarative to Imperative UML/OCL Operation Specifications. In: Conceptual Modeling - ER'07, vol. 4801, pp. 198-213. (2007)
6. Cabot, J., Clarisó, R., Riera, D.: Verification of UML/OCL Class Diagrams using Constraint Programming. In: MoDeVVA'08, (2008)
7. Cabot, J., Olivé, A., Teniente, E.: Representing Temporal Information in UML. In: The Unified Modeling Language - UML'03, vol. 2863, pp. 44-59. (2003)
8. Cabot, J., Teniente, E.: Determining the Structural Events that may Violate an Integrity Constraint. In: The Unified Modelling Language - UML'04, vol. 3273, pp. 320-334. (2004)
9. Cabot, J., Gómez, C.: Deriving Operation Contracts from UML Class Diagrams. In: Model Driven Engineering Languages and Systems - MODELS'07, vol. 4735, pp. 196-210. (2007)
10. CincomSmalltalk: VisualWorks. <http://www.cincomsmalltalk.com/>
11. Compuware: OptimalJ. <http://frontline.compuware.com/javacentral/tools/25988.asp>
12. Costal, D., Sancho, M., Olivé, A. et al.: The Role of Structural Events in Behaviour Specification. In: Database and Expert Systems Applications - DEXA'07, vol. 1308, pp. 673-686. (1997)
13. Costal, D., Sancho, M., Teniente, E.: Understanding Redundancy in UML Models for Object-Oriented Analysis. In: Advanced Information Systems Engineering - CAiSE'02, vol. 2348, pp. 659-674. (2002)
14. Costal, D., Teniente, E., Urpí, T. et al.: Handling Conceptual Model Validation by Planning. In: Advances Information System Engineering, vol. 1080, pp. 255-271. (1996)
15. Engels, G., Gogolla, M., Hohenstein, U. et al.: Conceptual Modelling of Database Applications using an Extended ER Model. Data & Knowledge Engineering, vol. 9, pp. 157-204. Elsevier Science Publishers B. V, Amsterdam, The Netherlands, The Netherlands (1992)
16. Fons, J.: EmpTraining: An Employee Training Management System. UPV, (2008) http://oomethod.dsic.upv.es/labs/index.php?option=com_content&task=view&id=49&Itemid=35
17. Gentleware: Poseidon for UML. <http://www.gentleware.com/>
18. IBM Software: Rational Rose. www.ibm.com/software
19. Interactive Objects: ArcStyler. OpenArchitectureWare Generator, (2007) <http://www.interactive-objects.com/products/arcstyler>
20. Kennedy Carter: IUML. <http://www.kc.com/products/iuml.php>

21. Laleau, R., Polack, F.: Specification of Integrity-Preserving Operations in Information Systems by using a Formal UML-Based Language. In: Information and Software Technology, vol. 43, pp. 693-704. (2001)
22. Larman, C.: Applying UML and patterns: An introduction to object-oriented analysis and design and the unified process. 2nd edn. Prentice Hall, (2001)
23. Link, S.: Consistency Enforcement in Databases. In: Semantics in Databases, vol. 2582, pp. 201-213. (2003)
24. McAllister, A. J., Sharpe, D.: An Approach for Decomposing N-Ary Data Relationships. Software: Practice and Experience, vol. 28, pp. 125-154. (1998)
25. Mellor, S. J., & Balcer, M. J.: Executable UML. A foundation for model-driven architecture. Object Technology edn. Addison-Wesley, Boston (2002)
26. Microsoft: Visual Studio 2008. <http://msdn2.microsoft.com/en-us/vstudio/default.aspx>
27. Moskitt - Conselleria d'Infraestructures i Transport: Moskitt. <http://www.moskitt.org/>
28. Object Management Group: Object Constraint Language (OCL), Version 2.0, OMG Available Specification (formal/2006-05-01). (2006) <http://www.omg.org/docs/ptc/03-10-14.pdf>
29. Object Management Group: Unified Modeling Language: Superstructure. Version 2.1. vol. ptc/06-04-02,
30. Objecteering Software: Objecteering UML. <http://www.objecteering.com/>
31. Olivé, A.: Conceptual modeling of information systems. Springer-Verlag, (2007)
32. Pastor, J. A., Olivé, A.: Supporting Transaction Design in Conceptual Modelling of Information Systems. In: Advanced Information Systems Engineering - CAiSE'95, vol. 932, pp. 40-53. (1995)
33. Pastor, O., Insfrán, E., Pelechano, V. et al.: OO-METHOD: An OO Software Production Environment Combining Conventional and Formal Methods. In: Advanced Information Systems Engineering - CAiSE'97, vol. 1250, pp. 145-158. Springer-Verlag, London, UK (1997)
34. Raventós, R.: A Conceptual Schema for a Conference Management Application. vol. 05-01-R, UPC, (2005)
35. Rumbaugh, J., Jacobson, I., Booch, G.: The unified modeling language reference manual. 2nd edn, Addison-Wesley, (2005)
36. Schewe, K. -, Thalheim, B.: Towards a Theory of Consistency Enforcement. In: Acta Informatica, vol. 36, pp. 97-141. (1999)
37. Sendall, S., Strohmeier, A.: Using OCL and UML to Specify System Behavior. In: Object Modeling with OCL, vol. 2263, pp. 250-280. Springer-Verlag, London, UK (2002)
38. Sendall, S., Strohmeier, A.: From use Cases to System Operation Specifications. In: The Unified Modeling Language - UML'00, vol. 1939, pp. 1-15. (2000)
39. SourceForge: Octopus: OCL Tool for Precise UML Specifications. <http://octopus.sourceforge.net/>
40. Stahl, T., & Voelter, M.: Model-driven software development: Technology, engineering, management. Wiley, (2006)
41. Tort, A.: OsCommerce Conceptual Schema. UPC, (2007) <http://guifre.lsi.upc.edu/OSCommerce.pdf>
42. Wieringa, R.: A Survey of Structured and Object-Oriented Software Specification Methods and Techniques. ACM Computing Surveys, vol. 30, pp. 459-527. ACM, New York, NY, USA (1998)

Appendix. Complete and executable set of operations for the running example

* Operations for the *Employee* class

```
context Employee::updateName(vname:String)
post: self.name=vname

context Employee::updateSalary(vsal:Integer)
post: self.salary=vsal

context Employee::createLinkWorksFor(vdept:Department)
post: self.department->includes(vdept)

context Employee::deleteLinkWorksFor()
post: self.department->excludes(self.department@pre)

context Employee::specializeJunior(vsen:Senior)
post: self.oclIsTypeOf(Junior) and
       self.supervisor->includes(vsen) and
       not self.oclIsTypeOf(Senior)

context Employee::specializeSenior(vexp:String)
post: self.oclIsTypeOf(Senior) and
       self.experience=vexp and
       not self.oclIsTypeOf(Junior)
```

* Operations for the *Junior* class

```
context Junior::create(vname:String, vdate:Date, vsal:Integer,
vsen:Senior)
post: j.oclIsNew() and j.oclIsTypeOf(Junior) and
       j.name=vname and j.dateOfBirth=vdate and
       j.salary=vsal and j.supervisor->includes(vsen)

context Junior::delete()
post: OclAny::allInstances()=OclAny::allInstances()@pre->
       excluding(self)

context Junior::generalizeEmployee(vexp:String)
post: not self.oclIsTypeOf(Junior) and
       self.oclIsTypeOf(Senior) and
       self.experience = vexp

context Junior::createLinkIsSupervisedBy(vsen:Senior)
post: self.supervisor->includes(vsen) and
       self.supervisor->excludes(self.supervisor@pre)
```

Alternative operation definitions for *createLinkIsSupervisedBy* (created by choosing different options in the OR dependency for *isSupervisedBy* event) are not included

because they are redundant (same body and signature) wrt other *Junior* operations. Likewise with the *deleteLinkIsSupervisedBy* operation.

* Operations for the *Senior* class

context Senior::create(vname:String, vdate:Date, vsal:Integer, vexp:String)

post: s.oclIsNew() and s.oclIsTypeOf(Senior) and
s.experience=vexp and
s.name=vname and s.dateOfBirth=vdate and s.salary=vsal

context Senior::delete()

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

context Senior::updateExperience(vexp:String)

post: self.experience=vexp

context Senior::generalizeEmployee()

post: not self.oclIsTypeOf(Senior) and
self.oclIsTypeOf(Junior) and

context Senior::createLinkIsSupervisedBy(vjun:Junior)

post: self.junior->includes(vjun)

context Senior::deleteLinkIsSupervisedBy(vjun:Junior)

post: self.junior->excludes(vjun)

* Operations for the *Department* class

context Department::create(vname:String, vcmp:Company, vemp:Employee)

post: d.oclIsNew() and d.oclIsTypeOf(Department) and
d.name=vname and
d.company->includes(vcmp) and d.boss-> includes(vemp)

context Department::delete()

post: OclAny::allInstances()=OclAny::allInstances()@pre->
excluding(self) and

context Department::updateName(vname:String)

post: self.name=vname

context Department::createLinkManages(vemp:Employee)

post: self.boss->includes(vemp) and
self.boss->excludes(self.boss@pre)

context Department::createLinkWorksFor(vemp:Employee)

post: self.department->includes(vemp)

context Department::deleteLinkWorkFor(vemp:Employee)

post: self.department->excludes(vemp)

Alternative operation definitions for *createLinkManages* and *deleteLinkManages*, (created by choosing different options in the OR dependencies) are not included because they are redundant wrt other Department operations.

* Operations for the *Investor* class

context Investor::create(vname:String, vnationality:String, vcmp:Company, vcapital:Integer)

post: i.ocIsNew() and i.ocIsTypeOf(Investor) and i.name=vname and i.nationality=vnationality and inv.ocIsNew() and inv.ocIsTypeOf(Investment) and inv.company=vcmp and inv.investor=i and inv.capital=vcapital

context Investor::delete()

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

context Investor::updateName(vname:String)

post: self.name=vname

context Investor::createLinkObjectInvestment (vcapital:Integer, vcmp:Company)

post: self.investment->includes(inv) and inv.ocIsNew() and inv.ocIsTypeOf(Investment) and inv.company=vcmp and inv.capital=vcapital

context Investor::deleteLinkObjectInvestment (vcmp:Company)

post: OclAny::allInstances= OclAny::allInstances()@pre->excluding(Investment::allInstances@pre()->select(x | x.company=vcmp and x.investor=self))

* Operations for the *Company* class

context Company::create(vname:String, vcity:City, vdept:Department)

post: c.ocIsNew() and c.ocIsTypeOf(Company) and c.name=vname and c.city->includes(vcity) and c.department->includes(vdept)

context Company::delete()

post: OclAny::allInstances()=OclAny::allInstances()@pre->excluding(self) and OclAny::allInstances()=OclAny::allInstances()@pre->excluding(self.department)

context Company::updateName(vname:String)

post: self.name=vname

context Company::createLinkOwns(vname:String, vemp:Employee)

post: d.ocIsNew() and d.ocIsTypeOf(Department) and d.name=vname and

```
d.boss->includes(vemp) and
self.department->includes(d)
```

```
context Company::deleteLinkOwns(vdept:Department)
post: OclAny::allInstances() =OclAny::allInstances()@pre->
excluding(vdept)
```

```
context Company::createLinkLocatedIn(vcity:City)
post: self.city->includes(vcity) and
self.city->excludes(self.city@pre)
```

```
context Company::createLinkObjectInvestment(vcapital:Integer,
vinv:Investor)
post: self.investment->includes(inv) and
inv.oclIsNew() and inv.oclIsTypeOf(Investment) and
inv.investor=vinv and inv.capital=vcapital
```

```
context Company::deleteLinkObjectInvestment(vinv:Investor)
post: OclAny::allInstances= OclAny::allInstances()@pre->
excluding(Investment::allInstances@pre()->select(x
x.company=self and x.investor=vinv) |
```

Alternative operation definitions for *deleteLinkLocatedIn* and *createLinkLocatedIn* are not included because they are redundant wrt other Department operations.

* Operations for the *Investment* class

```
context Investment::create(vcapital:integer, vinv:Investor,
vcmp:Company)
post: i.oclIsNew() and i.oclIsTypeOf(Investment) and
i.company=vcmp and i.investment=vinv and
i.capital=vcapital
```

```
context Investment::delete()
post: OclAny::allInstances() = OclAny::allInstances()@pre->
excluding(self)
```