Drawing Preconditions of Operation Contracts from Conceptual Schemas

Dolors Costal, Cristina Gómez, Anna Queralt, and Ernest Teniente

Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya
{dolors,cristina,aqueralt,teniente}@lsi.upc.edu

Abstract. Conceptual schemas include the definition of integrity constraints which must be satisfied in each state of the Information Base. Integrity constraints have a considerable impact on the specification of operations since operations should preserve the Information Base consistency. In this paper, we present an approach that automatically generates the preconditions that basic operations must include to ensure that a set of predefined integrity constraints is satisfied after their execution. Our approach is independent of the conceptual modelling language used. We also describe a prototype tool that implements our proposal for UML conceptual schemas.

Keywords: conceptual modelling, operation contracts, integrity constraints.

1 Introduction

An information system must include a representation of the *knowledge* of the domain, i.e. the Conceptual Schema (CS), and of the *state* of that domain, i.e. the Information Base (IB), to perform its functions.

The goal of automating information systems building was already stated in the late sixties [1]. However, and thanks to the definition and standardization of the MDA [2], this goal has revived and seems now more feasible than ever. For this reason, there has recently been a significant amount of work aimed at providing an automatic generation of (parts of) the software system from its specification.

In this context, we may find several proposals that provide an automatic definition of the basic operations (such as entity insertion or deletion, attribute modification, etc.) from a conceptual schema which allow updating the contents of the IB [3, 4, 5, 6]. Their main drawback is that either they do not take into account the integrity constraints to be preserved during the automatic generation of the operations or they consider them only up to a limited extent. Nevertheless, the automatic generation of the software elements required to ensure that the IB always satisfies the constraints of the CS is a crucial issue in software automation [7].

Our approach in this paper represents a step forward in this direction. Given a set of basic operations that update the contents of the IB (which may be either manually or automatically generated), a conceptual schema and a set of predefined integrity constraints, we are able to automatically determine the weakest precondition that must be considered for each basic operation so that integrity constraints are never violated when the operation is executed. Since we only consider adding preconditions,

Z. Bellahsène and M. Léonard (Eds.): CAiSE 2008, LNCS 5074, pp. 266–280, 2008.

integrity enforcement is achieved by avoiding the operation execution when its precondition is not satisfied. Our approach is independent of the conceptual modelling language used, although we will use UML and OCL in our examples.

In this way, our approach facilitates the automatic model-driven development of the information system from its initial specification since it simplifies the manual computation of the operation preconditions during software development. We have also developed an implementation of our approach which is integrated in a CASE tool.

As an example, consider the conceptual schema of Figure 1 which contains information about the employees assigned to projects and their supervisors. The schema contains three textual and two graphical constraints.

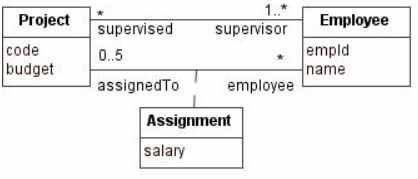

Fig. 1. Conceptual schema of our example application

Figure 2 shows a natural specification of the operation that assigns employees to projects. We assume that the parameters are provided as objects but their identifiers could be used as well.

Operation: newAssignment(e: Employee, p: Project, sal: Float)
Pre: *--the employee is not assigned to the project*
e.assignedTo -> excludes(p)
Post: *--a new instance of Assignment is created*
Assignment.allInstances()->exists(a | a.oclIsNew() and
a.salary = sal and a.project=p and a.employee=e)

Fig. 2. A sample partial contract for the operation *newAssignment*

It can be easily seen that the previous contract does not take integrity constraints into account since its precondition does not ensure that all constraints are satisfied. For instance, it allows assigning an employee to a project even if he is its supervisor. Therefore, this precondition must be extended to guarantee that the operation execution always leads the IB to a consistent state. Doing this by hand is time-consuming and error prone since it is not easy to identify the integrity constraints that may be violated by the operation execution and the additional required preconditions.

The contract of *newAssignment* that incorporates all the knowledge provided by the integrity constraints is shown in Figure 3 and it can be automatically obtained with our approach.

An automatic computation of the preconditions required to ensure that the operation contracts do not violate any integrity constraint provides two important contributions. First, it improves the quality of the specified operations since human

Operation: newAssignment(e: Employee, p: Project, sal: Float)
Pre:
--the employee is not assigned to the project e.assignedTo -> excludes(p)
--the salary is greater than 100 sal > 100
--the employee does not supervise the project p.supervisor -> excludes(e)
--the employee is not assigned to five projects e.assignedTo -> size()<5
Post:
--a new instance of Assignment is created
Assignment.allInstances()->exists(a | a.oclIsNew() and
a.salary = sal and a.project=p and a.employee=e)

Fig. 3. The full contract for the operation *newAssignment*

mistakes can be completely avoided. Second, software development is accelerated since integrity-preserving contracts can be automatically obtained.

The rest of the paper is organized as follows. The next section reviews some preliminary concepts. Section 3 describes a set of basic predefined operations. In section 4, we describe the conflicts that arise between integrity constraints and operations and we present our proposal for the automatic generation of preconditions. Section 5 describes a tool that implements our proposal. Related work is reviewed in section 6 and, finally, section 7 presents some conclusions and points out future work.

2 Preliminary Concepts

A CS consists of a taxonomy of entity types together with their attributes, a set of relationship types, and a set of integrity constraints [8]. A relationship type has several participants, i.e. entity types that play a certain role in the relationship type. In this paper, we deal with relationship types that have two participants (i.e. binary). Some relationship types are reified and, thus, they may have attributes and participate in other relationship types.

An information system maintains a representation of the state of a domain in its IB [9]. The state of the IB is the set of instances of the entity types and relationship types defined in the CS. The integrity constraints of the CS define conditions that each state of the IB must satisfy. Those constraints can have a graphical representation or can be defined through a particular language.

Additionally, a CS includes a set of operations, and the content of the IB changes as a result of their execution. The effect of each operation on the IB is specified by an operation contract. An operation contract is defined by a precondition, which expresses a condition that must be satisfied when the call to the operation is made, and a postcondition, which expresses a condition that the new state of the IB must satisfy [10].

Integrity constraints are closely related to operations, since the former must hold in every state of the IB, and the latter are the ones that change its content. Then, an operation contract must guarantee that the integrity constraints defined in the schema hold after its execution. We consider the following predefined integrity constraints (a more detailed description can be found in [11]).

An *identifier constraint* specifies a set of properties that uniquely identifies each instance of an entity type. Let E be an entity type and $\{p_1,...,p_n\}$ a set of properties, which can be attributes or roles. An identifier constraint specifies that a subset $\{p_i,...,p_j\}$ of these properties uniquely identifies the instances of E.

Recursive relationship type constraints, referred to as ring constraints in [12], are constraints that apply over recursive binary relationship types to guarantee that the relationship type fulfils a certain property. We consider five such constraints: *symmetric*, *asymmetric, antisymmetric, irreflexive* and *acyclic* constraints.

Let E be an entity type and R a recursive relationship type over E. A symmetric constraint over R guarantees that if a and b are instances of E and a is R-related to b, then b is R-related to a. An asymmetric constraint guarantees that if a and b are instances of E and a is R-related to b, then b is not R-related to a. An antisymmetric constraint over R guarantees that if a and b are instances of E, a is R-related to b and b is R-related to a, then a and b are the same instance. An irreflexive constraint over R guarantees that if a is an instance of E then a is never R-related to itself. An acyclic constraint guarantees that if a and b are instances of E and a is R-related to b, then b or instances R-related directly or indirectly to b are not R-related to a.

Path comparison constraints restrict how to relate the population of one role or role sequence (i.e. a path) to the population of another [12]. *Path inclusion, path exclusion* and *path equality* are all examples of this type of constraint and apply to an entity type A related to an entity type B via two different paths $r_1...r_i$, and $r_j...r_n$. A path inclusion constraint guarantees that if a is an instance of A, the set of instances of B related to a via $r_1...r_i$ includes the set of instances of B related to a via $r_j...r_n$. In a similar way, a path exclusion constraint ensures that the intersection between the populations of both paths is empty while a path equality constraint guarantees that both populations contain exactly the same instances.

Value comparison constraints restrict the values of an attribute by comparing it with a constant or with another attribute value [13]. Let E be an entity type, let a_i be an attribute of E, let v be either a constant or the value of an attribute accessible from E, and let op be an operator of type $<$, $>$, $=$, $\neq$, $\leq$, or $\geq$. A value comparison constraint restricts the values of a_i with respect to the value of v according to op.

Cardinality constraints for binary relationship types restrict the number of instances that can be related to another instance through the relationship type. Let R be a binary relationship type such that entity type E_1 plays role p_1 and entity type E_2 plays role p_2 in it. A cardinality constraint from p_1 to p_2 in R indicates the minimum and maximum number of instances of type E_2 that may be related with any instance of type E_1 through R [14]. Cardinality constraint from p_2 to p_1 in R is defined similarly.

Disjointness and *covering* constraints impose restrictions on the population of a set of entity types. A disjointness constraint for entity types $E_1,...,E_n$ indicates that a particular entity can be instance of at most one E_i [15]. A covering constraint between an entity type E and a set of entity types $E_1,...,E_n$ indicates that every instance of E is instance of at least one E_i [15].

3 Basic Operations

A CS must be complemented with a set of operations that define how the users may modify the contents of the IB. In this paper, we deal with basic operations. We describe our basic operations in terms of their postconditions because our approach only depends on them to generate operation preconditions. We consider the following set of basic operations which correspond to the categories identified in [16] to

describe operation postconditions. For the sake of generality, we use external identifiers instead of objects in the operation signatures. Therefore, each instance to be modified is identified by a set of attribute values and not by its object reference.

InstanceCreation. The operation createE(v_1,…,v_n: Set(String)) creates an instance of entity type *E* and gives values $v_1,\ldots,v_n$ to attributes $a_1,\ldots,a_n$ of *E*. The postcondition of this operation can be specified in OCL [17] as follows:

post: E.allInstances()-> exists(e |e.oclIsNew()and e.a_1=v_1 and … and e.a_n=v_n)

As a result of this operation, the new instance belongs to *E* and all its supertypes.

InstanceDeletion. The operation deleteE(id_1,…,id_n: Set(String)) deletes an instance of entity type *E* identified by parameters $id_1,\ldots,id_n$. Its postcondition is:

post: not(E.allInstances()->exists(ele.p_1=$id_1$and …and e.p_n=id_n))

where $p_1,..,p_n$ are the paths that identify the instances of *E*. We assume that all the relationships in which the instance participates are deleted, and that the instance is deleted from *E* and all its supertypes.

AttributeValueModification. The operation modifyAfromE(id_1,…,id_n,nv: Set(String)) modifies attribute *a* of an instance of the entity type *E*. The instance to modify is identified by parameters $id_1,\ldots,id_n$ of the operation. The new value for the attribute is *nv*. Its postcondition is:

post: E.allInstances()-> select(e |e.p_1=id_1 and … and e.p_n=id_n).a = nv

where $p_1,..,p_n$ are the paths that identify the instances of *E*.

RelationshipCreation. The operation createR($id1_1$,...,$id1_n$,$id2_1$,...,$id2_m$: Set(String)) creates an instance of the relationship type *R* between two instances *i1*and *i2* playing roles *r1* and *r2* in *R*. The instances to relate are identified, respectively, by the parameters $id1_1,\ldots,id1_n$ and $id2_1,\ldots,id2_m$, and can be obtained as follows from them:

let i1: E1 = E1.allInstances()-> select(e| e.$p1_1$=$id1_1$ and … and e.$p1_n$=$id1_n$)
let i2: E2 = E2.allInstances()-> select(e| e.$p2_1$=$id2_1$ and … and e.$p2_m$=$id2_m$)

The postcondition of this operation is: **post:** i1.r2->includes(i2)

RelationshipDeletion. The operation deleteR($id1_1$,...,$id1_n$,$id2_1$,...,$id2_m$: Set(String)) deletes the instance of the relationship type *R* between two instances *i1*and *i2* playing roles *r1* and *r2* in *R*. These instances are identified, respectively, by the parameters $id1_1,\ldots,id1_n$ and $id2_1,\ldots,id2_m$, and can be obtained as in the previous operation. The postcondition of this operation is: **post:** i1.r2->excludes(i2)

InstanceGeneralization. The operation generalize$E_i$toE(v_1,…,v_m: Set(String)) establishes that an instance of *E* which is identified by values $v_1,\ldots,v_m$ for paths $p_1,\ldots,p_m$, respectively, is not an instance of E_i after its execution (although it has not been deleted from the IB and it is still an instance of *E*). The OCL postcondition of this operation is:

```
let i1: E = E.allInstances()-> select(e |e.p1=v1 and …
      and e.pm=vm)
post: not (i1.oclIsTypeOf(Ei))
```

We assume that all the relationships in which *i1* participates are deleted.

InstanceSpecialization. The operation `specializeEtoEi(v1,…,vm: Set(String), nv1,…,nvk: Set(String))` establishes that an instance *i1* of *E* is also an instance of E_i after its execution. Additionally, it takes values $nv_1,\ldots,nv_k$ for the attributes $a_1,\ldots,a_k$ of E_i. The instance is identified by values $v_1,\ldots,v_m$ for paths $p_1,\ldots,p_m$, respectively. Its postcondition is:

```
let i1:E=E.allInstances()-> select(e |e.p1=v1 and … and e.pm=vm)
post: i1.oclIsTypeOf(Ei) and i1.a1=nv1 and … and i1.ak=nvk
```

We consider also other basic operations whose postcondition can be stated as a combination of those of the basic operations specified so far. They are the following:

WeakInstanceCreation. The operation `createW(id1,…,idn: Set(String), v1,…,vm: Set(String))` creates an instance of entity type *W*, gives values $v_1,\ldots,v_m$ to attributes $a_1,\ldots,a_m$ of *W* and relates it through a relationship type *R* to an instance *i* of entity type *S* playing role *rs* in *R*. The instance *i* is identified by the parameters $id_1,\ldots,id_n$ and can be obtained as follows from them:

```
let i:S=S.allInstances()->select(e|e.p1=id1 and … and e.pn=idn)
```

The postcondition of this operation is:

```
post: W.allInstances()-> exists(e |e.oclIsNew()and e.a1=v1 and
      … and e.am=vm and e.rs=i)
```

ReifiedRelationshipCreation. The operation `createRR(id11,...,id1n, id21,...,id2m: Set(String), v1,…,vk: Set(String))` creates an instance of the reified relationship type *R* that relates instances *i1* and *i2* playing roles *r1* and *r2* in *R*. Additionally, it takes values $v_1,\ldots,v_k$ for the attributes $a_1,\ldots,a_k$ of *R*. The instances to relate are identified, respectively, by $id1_1,\ldots,id1_n$ and $id2_1,\ldots,id2_m$, and can be obtained as described in the RelationshipCreation operation. Its postcondition is:

```
post: R.allInstances()-> exists(e |e.oclIsNew()and e.r1=i1
    and e.r2=i2 and e.a1=v1 and … and e.ak=vk)
```

We define *WeakInstanceDeletion* and *ReifiedRelationshipDeletion* in a similar way; as well as *InstanceChangeOfSubclass* which mixes *InstanceGeneralization* and *InstanceSpecialization*. We omit their formal definition due to space limitations.

4 Automatic Generation of Operation Preconditions

We describe in this section the approach we propose to automatically generate the weakest preconditions required by our set of basic operations in order to guarantee that their execution does not violate any of the predefined integrity constraints. By weakest we mean the necessary and sufficient conditions that allow ensuring that the constraints will not be violated after applying just the minimum changes specified by the postcondition when the operation precondition is satisfied (i.e. without requiring compensatory actions to restore the IB consistency).

It may happen that the execution of a basic operation postcondition always leads to an integrity constraint violation. Then, no weakest precondition exists. Our approach is able to identify these situations and it discards the definition of such operations.

We identify in section 4.1 the conflicts that arise between predefined constraints and basic operations. Then, in section 4.2, we describe how the weakest preconditions can be automatically obtained.

4.1 Conflicts between Constraints and Operations

The following table summarizes the conflicts that exist between integrity constraints and operations. Columns correspond to the predefined integrity constraints, and rows to the basic operations. A cross in a cell represents that there is a conflict between the corresponding constraint and operation, meaning that the constraint may be violated when the postcondition of the operation is satisfied. Thus, some preconditions must be added to the operation to prevent the violation in these cases.

Table 1. Conflicts between predefined constraints and basic operations

	Identifie	**Irreflexi**	**Symmetr**	**Asymm**	**Antisym**	**Acyclic**	**PathIncl**	**PathExcl**	**PathEq**	**ValueCo**	**Min.**	**Max.**	**Disjoint**	**Covering**
InstanceCreation	×									×	×			×
InstanceDeletion							×		×		×			
AttributeValueModif.	×									×				
RelationshipCreation		×	×	×	×	×	×	×	×			×		
RelationshipDeletion			×				×		×		×			
InstanceGeneralization							×		×		×			×
InstanceSpecialization	×									×	×		×	×
WeakInstanceCreation	×						×	×	×	×	×	×		×
WeakInstanceDeletion							×		×		×			
ReifiedRelationshipCre	×	×	×	×	×	×	×	×	×	×	×	×		×
ReifiedRelationshipDel			×				×		×		×			
InstanceChangeOfSubty	×						×		×	×	×			

The explanation of all marks in the table will be provided in the next section while identifying the preconditions required by the operations in each case.

4.2 Drawing Preconditions

The preconditions that are generated for each basic operation are the following.

InstanceCreation
The operation `createE(`v_1`,…,`v_n`: Set(String))` may violate identifier, value comparison, minimum cardinality and/or covering constraints.

Identifier. The violation of an identifier constraint for the entity type *E* or one of its supertypes occurs when the values for the identifying properties of the created instance are equal to those values for an already existing instance. To prevent it, the following precondition must be added to the operation:

pre: not(E.allInstances()->exists(e|e.a_i=v_i and … and e.a_j=v_j))

where $a_i,\ldots,a_j$ are the identifier attributes and $v_i,\ldots,v_j$ are the new values of the created instance for them.

For example, an instance creation operation `createEmployee(ei:String, nm:String)` for the conceptual schema shown in Figure 1 requires the following precondition since there is an identifier constraint which states that employees are identified by their *empid*:

pre: not(Employee.allInstances()-> exists(e |e.empid=ei))

Value Comparison. A value comparison constraint a_i *op* v for any attribute a_i of *E* or one of its supertypes that is initialized by the operation is violated if the specified comparison is not satisfied by the new instance. Thus, the following precondition is needed for each such a_i attribute:

pre: v_i op v

where v_i is the value of the created instance for a_i.

Minimum Cardinality. Let *R* be a binary relationship type such that entity type *E* plays role *p* and an entity type E_1 plays role p_1 in it. A minimum cardinality constraint from *p* to p_1 in *R* is always violated by the operation, since it creates an unrelated instance. The violation cannot be prevented by means of a precondition and, consequently, the operation cannot be executed in any case. Therefore, the InstanceCreation operation is discarded in this case.

Covering. Any covering constraint between entity type *E* and a set of entity types $E_1,\ldots,E_n$ is violated since the operation creates an instance in a single entity type. Again, the violation occurs in any case and the operation cannot be executed.

InstanceDeletion

An instance deletion operation, deleteE($id_1,\ldots,id_n$: Set(String)), may induce the violation of path inclusion, path equality and/or minimum cardinality constraints.

Path Inclusion. A path inclusion constraint which states that a first path includes a second path can be violated if the operation deletes an instance of one of the entity types that is traversed by the first path. The violation occurs when, after the deletion, the set of instances related to an instance i via the first path does not include the set of instances related to i via the second one. The following precondition is then required:

```
pre: Start.allInstances()-> forAll(s |newPath1(s)->
    includesAll(newPath2(s)))
```

Start is the origin entity type of both paths. *NewPath1(s)* and *newPath2(s)* define the set of instances that are reached from instance *s* by the first and second paths, respectively, assuming that the postcondition of the operation holds.

Path Equality. A path equality constraint between two paths can be violated by an instance deletion if the operation deletes an instance of an entity type in any of the two paths. The violation occurs when, after the deletion, the set of instances related to an instance i via the first path is not equal to the set of instances related to i via the second one. Therefore, the following precondition must be added to the operation:

pre: `Start.allInstances()->forAll(s |newPath1(s)=newPath2(s))`

where *Start*, *newpath1(s)* and *newpath2(s)* are defined as in the path inclusion case.

Minimum Cardinality. Let *E* be an entity type such that one of its instances is deleted by the operation. Let *R* be a relationship type such that an entity type E_1 plays role p_1 and entity type *E* plays role *p* in it. A minimum cardinality constraint from p_1 to *p* in *R* is violated if there is an instance *i* belonging to E_1 that was related to the deleted instance and that, after the deletion, does not satisfy the minimum cardinality any more. The violation can be prevented by the precondition:

pre: `delInst.`p_1`->forAll(e1 |e1.p->size()>min)`

where *delInst* defines the deleted instance.

Note that the previous precondition will always evaluate to false if *R* has a maximum and a minimum cardinality constraints restricted by the same value. Therefore, the operation should be discarded in this case.

For instance, our running example of Figure 1 depicts a minimum cardinality constraint to ensure that all projects have at least one supervisor employee. Therefore, the following precondition must be generated for `deleteEmployee(ei:String)`, aimed at deleting an employee with code *ei*.

pre: `delInst.supervises->forAll(pr|pr.supervisor->size()>1)`

where *delInst* defines the deleted instance:

```
let delInst : Employee = Employee.allInstances()->
    select(e |e.empid=ei)
```

Assuming that we had a subtype *JuniorEmployee* of *Employee* in our example, the basic operation `deleteJuniorEmployee(ei:String)` would also require the previous precondition.

AttributeValueModification

An attribute value modification operation, `modifyAfromE(`id_1`,…,`id_m`,nv: Set(String)`, may violate identifier and/or value comparison constraints.

Identifier. This operation violates identifier constraints if the values of the updated instance for the identifying properties are equal to those values for another instance, after the modification. To avoid it, the following precondition is needed:

pre: `not(E.allInstances()-> exists(e |e.`p_i`=k.`p_i` and … and e.a=nv and … and e.`p_j`=k.`p_j`))`

where p_i,…,*a*,…,p_j are the *E* identifier properties specified by the constraint and *k* is defined as the instance updated by the operation.

Value Comparison. A value comparison constraint for the updated attribute, *a op v*, is violated if the specified comparison is not satisfied by the new value. The following precondition must be added to the operation:

```
pre: nv op v
```

RelationshipCreation

The operation `createR(id1`$_1$`,…,id1`$_n$`,id2`$_1$`,…,id2`$_m$`:Set(String))` creates an instance of a relationship type *R* between instances *i1* and *i2* of entity types E_1 and E_2 playing roles r_1 and r_2 in *R*. As can be seen in table 1, this operation may violate several constraints, many of them when *R* is recursive.

Irreflexive. If *R* has an irreflexive constraint, the violation happens when *i1=i2*. The precondition to be added is:

```
pre: i1 <> i2
```

Symmetric. If *R* has a symmetric constraint, the violation happens if *i2* is not R-related to *i1*, i.e. when an instance that is symmetric to the new one does not exist. Since the IB must be consistent before the execution of any operation, the symmetric instance needed will never exist. Thus, the violation cannot be prevented by means of a precondition and the operation should be discarded.

Antisymmetric. When the relationship has an antisymmetric constraint, the violation happens when *i2* is R-related to *i1,* unless *i1* and *i2* are the same instance. In this case, the following precondition must be added to the operation:

```
pre: i2.r1->includes(i1) implies i2=i1
```

Asymmetric. On the contrary, an asymmetric constraint in a recursive relationship type is violated when *i2* is already R-related to *i1*. The following precondition has to be added to prevent the previous violation:

```
pre: i2.r1->excludes(i1)
```

Acyclic. If the relationship type has an acyclic constraint, it is violated when *i2* is R-related (directly or indirectly) to *i1*, both of them instances of E_1.

```
pre: i2.successors()->excludes(i1)
```

where `successors()` recursively obtains all the instances that are R-related to an instance of E_1. It is defined as follows:

```
context E1 def:
successors():Set(E1) = self.r1->union(self.r1.successors())
```

For relationship types that are not necessarily recursive, the constraints that may be violated are path constraints and maximum cardinality constraints.

Path Inclusion, Equality and Exclusion. A path inclusion constraint that traverses *R* is violated when, after the creation, the set of instances related to an instance *i* via the first path does not include the set of instances related to *i* via the second one. Violations of path exclusion and path equality constraints can be explained analogously. The preconditions to be added are the same than in the instance deletion operation.

Maximum Cardinality. A maximum cardinality from r_1 to r_2 is violated when *i2* is already related to *max* instances of *E1*. The violation can be prevented by adding the following precondition:

```
pre: i1.r2->size() < max
```

If the maximum cardinality constraint is from r_2 to r_1, the precondition needed is:

```
pre: i2.r1->size() < max
```

As before, the operation is discarded if there is a maximum and a minimum cardinality constraint restricted by the same value.

RelationshipDeletion

When `deleteR(id1`$_1$`,…,id1`$_n$`,id2`$_1$`,…,id2`$_m$`:Set(String))` operation deletes the instance of the relationship type *R* between two instances *i1* and *i2* of entity types E_1 and E_2, playing roles *r1* and *r2* in *R*, the constraints that may be violated are the symmetric, path inclusion, path exclusion and minimum cardinality constraints.

Symmetric. If *R* is recursive and symmetric, this constraint is violated when, after the deletion, *i2* is R-related to *i1*. This will always happen, since the operation `deleteR` deletes a single instance. Thus, this violation cannot be prevented in any case.

Path Inclusion and Equality. A path inclusion constraint that traverses *R* is violated when, after the deletion, the set of instances related to an instance *i* via the first path does not include the set of instances related to *i* via the second one. The reason for the violation of path equality is analogous. The preconditions to be added for these cases are the same than in the previous operation.

Minimum Cardinality. A minimum cardinality constraint from r_1 to r_2 is violated when, after the deletion, *i2* is related to less than *min* instances of E_1. The following precondition must be added:

```
pre: i1.r2->size() > min
```

If the minimum cardinality constraint is from r_2 to r_1, the precondition needed is:

```
pre: i2.r1->size() > min
```

Again, the operation is discarded if there is a maximum and a minimum cardinality constraint restricted by the same value.

InstanceGeneralization

An instance generalization may violate path inclusion, path equality, minimum cardinality and/or covering constraints.

In some respects, an instance generalization is similar to an instance deletion since, in both cases, the particular entity affected by the operation is no longer an instance of an entity type after its execution. Thus, violations of path inclusion, path equality or minimum cardinality constraints are like those described above for instance deletions and can be prevented by similar preconditions.

Additionally, an operation `generalizeE`$_i$`toE(v`$_1$`,…,v`$_m$`: Set(String))`, may violate a covering constraint between entity type *E* and a set of entity types $E_1,...,E_n$ that include E_i. The violation occurs if the involved entity is not an instance of any $E_1,...,E_n$ after the execution of the operation. We need the following precondition:

```
let i1: E = E.allInstances()-> select(e |e.p1=v1 and …
    and e.pm=vm)
pre: i1.oclIsTypeOf(E1) or … or i1.oclIsTypeOf(Ei-1) or
    i1.oclIsTypeOf(Ei+1) or … or i1.oclIsTypeOf(En)
```

InstanceSpecialization
It may violate identifier, value comparison, minimum cardinality, disjoint and/or covering constraints.

An instance specialization is similar to an instance insertion because an entity starts to be an instance of a certain entity type after the execution of both operations. Thus, violations of identifier, value comparison, minimum cardinality or covering constraints are like those described above for instance insertions.

The operation `specializeEtoEi(v1,…,vm: Set(String), nv1,…,nvk: Set(String))`, may also violate a disjointness constraint for a set of entity types $E_1,...,E_n$ that include E_i. This happens if the involved entity is an instance of more than one $E_1,...,E_n$ after the execution. The precondition that avoids the violation is:

```
let i1: E = E.allInstances()-> select(e |e.p1=v1 and …
    and e.pm=vm)
pre: not(i1.oclIsTypeOf(E1) or … or i1.oclIsTypeOf(Ei-1) or
    i1.oclIsTypeOf(Ei+1) or … or i1.oclIsTypeOf(En))
```

We omit the description of the preconditions that are generated for the rest of basic operations because those preconditions can be seen as combinations of the cases that have already been described.

For instance, the operation *newAssignment* in our example of the introduction is a ReifiedRelationshipCreation operation whose effect is defined by combining an InstanceCreation and a RelationshipCreation operations. Then, the preconditions added to the contract in Figure 3 correspond to a violation of a *Value Comparison* constraint of the InstanceCreation and a violation of a *Path Exclusion* and a *Maximum Multiplicity* constraints of the RelationshipCreation.

5 Prototype Tool

We have developed a prototype tool that allows the automatic computation of the preconditions of the operation contracts, along the ideas developed in this paper, on top of Poseidon® 4.1 since this CASE tool provides an extension mechanism by means of Java plug-ins.

The designer may specify an operation as provided by Poseidon®. Then, with our plug-in, he may make use of the basic operations to state its postcondition. In Figure 4, we show the specification of an instance creation operation *newPerson* aimed at creating instances of the class *Persona*. Once this is done, he can press the button *Normalize* to automatically obtain the preconditions required for the operation contract. As can be seen in Figure 5, the resulting contract includes a precondition to prevent the violation of the specified identifier constraint.

Our prototype allows the definition and treatment of most of the basic operations considered in this paper. In particular, it is able to handle InstanceCreation, InstanceDeletion, AttributeValueModification, RelationshipCreation, Relationship Deletion, WeakInstanceCreation and WeakInstanceDeletion.

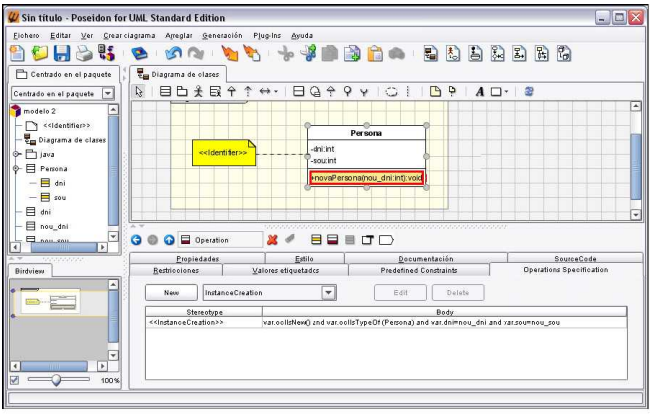

Fig. 4. Specification of an operation contract for *newPerson*

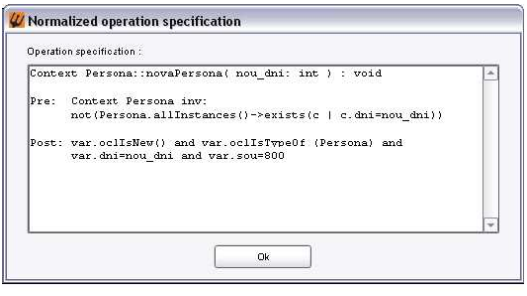

Fig. 5. Automatic generation of the precondition of *newPerson*

6 Related Work

The problem of identifying preconditions of an operation is not new. It has been addressed in the database field and in conceptual modelling of information systems as part of the checking and integrity maintenance problem (see, among others, [4] and [18]). [4] automatically generates elementary operations from an extended ER model of a database application. These operations contain additional manipulations, known as update propagations, to maintain some integrity constraints defined in the conceptual model. Preconditions to guarantee cardinality constraints and other general constraints have to be added to the specification of complex operations (sequence of elementary ones) by the designer. [18] draws automatically a transaction specification from a conceptual model and identifies conditions (preconditions) and repair actions to preserve integrity constraints. This method does not deal with cardinality constraints.

Ackermann and Turowski [13] propose a set of OCL specification patterns that facilitate the definition of some preconditions (as class instance existence, value specification of input parameter and so on). The use of these patterns simplifies the specification of operations although preconditions for each operation must be identified manually by the designer.

In [19] an identification process of preconditions for operations to modify instances of a data model (only a subset of the OMT object model is considered with classes and relations) is defined. This process is not systematic and requires interaction with the designer. An initial precondition for an operation must be provided by the designer and then the *Z-EVES* theorem prover is used to verify whether this precondition is needed for the operation.

A goal similar to ours is addressed in [20], that proposes an approach to identify the weakest preconditions to be added to the operations such that their execution does not violate any integrity constraint. They consider UML class diagrams but they assume that constraints and operation contracts are specified in the B language. Our approach, however, is independent of the conceptual modelling language used. We have shown how to apply it in OCL, which is the language most frequently used. Another difference is that their approach is based on performing general reasoning on the relevant B expressions while we provide an ad-hoc treatment endowed to the particular semantics of each basic operation and predefined constraint.

7 Conclusions and Future Work

Conceptual schemas usually include an important amount of integrity constraints, which must be satisfied in each state of the IB. These constraints may have a graphical representation or can be defined by means of a particular language. The content of the IB changes due to the execution of operations. The effect of an operation is defined by means of a postcondition, which expresses a condition that the IB must satisfy after applying it. Preconditions, which must be satisfied before the execution of the operation, must guarantee that it leaves the IB in a state satisfying all the constraints.

Due to the great amount of constraints that a schema may include, the task of manually determining which preconditions are needed by each operation is time consuming and error prone. To overcome this limitation, we have presented an approach to automatically generate the preconditions needed to guarantee that an operation satisfies the integrity constraints defined in the schema after being executed. Our approach is able to deal with a set of predefined integrity constraints and basic operations and allows to determine the weakest precondition which ensures that the postcondition can be safely applied. As an additional result of this automation, software development will also be performed faster. We have implemented our approach and integrated it in a CASE tool.

Future research may involve drawing preconditions from complex non-basic operations, i.e., operations defined as combinations of the basic ones studied in this work. Additionally, we plan to deal with other types of frequent general constraints. It may also be worth studying how the violation of an integrity constraint may be solved by including some corrective action instead of forbidding the operation execution.

Acknowledgements. We would like to thank Quim Vilà for developing the prototype and the GMC group for helpful discussions on this paper. We are also grateful to the anonymous referees for their useful comments. This work has been partially supported by the Ministerio de Ciencia y Tecnología under project TIN2005-06053.

References

1. Teichroew, D.: Methodology for the Design of Information Processing Systems. In: Proc. Fourth Australian Computer Conference, pp. 629–634 (1969)
2. OMG: MDA Guide Version 1.0.1. (2003)
3. Costal, D., Sancho, M.-R., Olivé, A., Roselló, A.: The Role of Structural Events in Behaviour Specification. In: Tjoa, A.M. (ed.) DEXA 1997. LNCS, vol. 1308, pp. 673–686. Springer, Heidelberg (1997)
4. Engels, G., Gogolla, M., Hohenstein, U., Hüllmann, K., Löhr-Richter, P., Saake, G., Ehrich, H.-D.: Conceptual Modelling of Database Applications Using an Extended ER Model. Data & Knowledge Engineering 9, 157–204 (1992)
5. Laleau, R., Polack, F.: Specification of Integrity-Preserving Operations in Information Systems by Using a Formal UML-based Language. Information and Software Technology 43, 693–704 (2001)
6. Cabot, J., Gómez, C.: Deriving Operation Contracts from UML Class Diagrams. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 196–207. Springer, Heidelberg (2007)
7. Olivé, À.: Conceptual Schema-Centric Development: A Grand Challenge for Information Systems Research. In: Pastor, Ó., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, pp. 1–15. Springer, Heidelberg (2005)
8. Olivé, A.: Conceptual Modeling of Information Systems. Springer, Heidelberg (2007)
9. ISO/TC97/SC5/WG3: Concepts and Terminology for the Conceptual Schema and Information Base. ISO (1982)
10. Meyer, B.: Object-Oriented Software Construction, 2nd edn. Prentice-Hall, Englewood Cliffs (1997)
11. Costal, D., Gómez, C., Queralt, A., Raventós, R., Teniente, E.: Improving the Definition of General Constraints in UML. Software and Systems Modeling (2008) DOI: 10.1007/s10270-007-0078-4
12. Halpin, T.: Information Modeling and Relational Databases: From Conceptual Analysis to Logical Design. Morgan Kaufmann, San Francisco (2001)
13. Ackermann, J., Turowski, K.: A Library of OCL Specification Patterns for Behavioral Specification of Software Components. In: Dubois, E., Pohl, K. (eds.) CAiSE 2006. LNCS, vol. 4001, pp. 255–269. Springer, Heidelberg (2006)
14. Liddle, S.W., Embley, D.W., Woodfield, S.N.: Cardinality Constraints in Semantic Data Models. Data and Knowledge Engineering 11, 235–270 (1993)
15. Lenzerini, M.: Covering and Disjointness Constraints in Type Networks. In: Proc. ICDE 1987, pp. 386–393. IEEE Computer Society Press, Los Alamitos (1987)
16. Larman, C.: Applying UML and Patterns, 3rd edn. Prentice-Hall, Englewood Cliffs (2004)
17. OMG: UML2.0 OCL Specification, OMG Adopted Specification (2005)
18. Pastor, J.A., Olivé, A.: Supporting Transaction Designs in Conceptual Modeling of Information Systems. In: Iivari, J., Rossi, M., Lyytinen, K. (eds.) CAiSE 1995. LNCS, vol. 932, pp. 40–53. Springer, Heidelberg (1995)
19. Ledru, Y.: Idenitfying pre-conditions with the Z/EVES theorem prover. In: Proc. 13th International Conf. on Automated Software Engineering. IEEE Computer Society Press, Los Alamitos (1998)
20. Mammar, A., Gervais, F., Laleau, R.: Systematic Identification of Preconditions from Set-Based Integrity Constraints. In: INFORSID, pp. 595–610 (2006)