Determining the Structural Events that May Violate an Integrity Constraint

Jordi Cabot and Ernest Teniente

Universitat Politècnica de Catalunya
Dept. Llenguatges i Sistemes Informàtics
Jordi Girona 1-3, 08034 Barcelona (Catalonia)

e-mail: [jcabot|teniente]@lsi.upc.es

Abstract. Any implementation of an information system must ensure that an operation is only applied if its execution does not lead to a violation of any of the integrity constraints defined in its conceptual schema. In this paper we propose a method to automatically determine the operations that may potentially violate an OCL integrity constraint in conceptual schemas defined in the UML. This is done by determining the structural events that may violate the constraint and checking whether those events appear in the operation specification. In this way, our method helps to improve efficiency of integrity checking since its results can be used to discard many irrelevant tests.

1. Introduction

A complete conceptual schema (CS) must include the definition of all relevant integrity constraints [5]. An integrity constraint states a condition that must be satisfied in each state of the information base (IB). Some constraints are inherent in the conceptual model in which the language is based but almost all constraints require an explicit definition [12, ch. 5]. Many constraints cannot be expressed using only the graphical constructs provided by the conceptual modeling language and require the use of a general-purpose (textual) sublanguage [3, ch.2]. In the UML this is usually done by means of invariants written in the OCL language [10].

The content of the IB changes due to the execution of the operations provided by the information system. Therefore, it must be guaranteed that the IB state resulting from these operations is consistent with regards to the set of integrity constraints specified over the CS. Moreover, if the application of an operation leads to an IB state where some integrity constraint is violated then the operation should be rejected and the contents of the IB should remain unchanged.

In general, the effect of an operation over the IB may be specified by means of a set of structural events (see for instance [7, 13]). A structural event is an elementary change in the population of an entity type (i.e. a class) or relationship type (i.e. an association) such as: create object, reclassify object, create link, etc. In particular, in the UML, structural events are a subset of the actions defined in the Actions Package [8, p.203+].

We may assert that a given operation will not violate a certain integrity constraint if we know that none of its structural events may induce the violation of such constraint. In this context, our work is aimed at proposing a method that automatically determines the structural events that may violate a constraint.

Our approach allows detecting those constraints that are irrelevant to a given operation and thus must not be taken into account during the process of checking integrity constraints after the operation execution. Hence, we may substantially improve the efficiency of this process since only the constraints we know can actually be violated by the operation must be taken into account.

Roughly, the rationale of our method is to find out the set of potentially violating structural events (PSEs, from now on) for each constraint and then compare this set with the set of structural events included in the operation to see if the operation includes some of them, and thus, its execution can violate that constraint.

The knowledge provided by our method may also be useful in the area of schema validation. For instance, if after applying our method we realize that a particular constraint can never be violated (none of the operations affects it) we can think of removing the constraint.

To our knowledge, ours is the first proposal to address the problem of determining the exact set of PSEs for an integrity constraint in conceptual schemas defined in UML and OCL.

Previous work addressing similar problems can be found in the fields of deductive or relational databases. Therefore, an alternative approach to solve this problem could consist of translating the OCL constraints into logic or SQL (using [2], for instance) and then make use of the algorithms developed for those technologies to determine the set of PSEs of the constraint.

Unfortunately, algorithms for deductive databases are not powerful enough for dealing with the expressiveness of OCL since this language allows negation, recursion, bag semantics (also known as duplicate semantics) and aggregation operations (like *size* or *sum*) which are hardly handled by those algorithms (see [4] for a discussion of their limitations). On the other hand, algorithms for relational databases (like [1]) support the required OCL constructs but lack precision when determining the relevant PSEs that may violate a constraint. For this reason, we believe that using an ad-hoc method to reason directly about the OCL expression that defines the integrity constraint (like the one we propose in this paper) is the best solution to deal with this problem in conceptual models defined in UML.

Our work could be included in any architecture aimed at generating automatically the implementation of an information system from its specification like [6]. It is also helpful in the context of the MDA [9] when deriving platform specific models from a platform independent model.

The structure of the paper is as follows. The next section reviews structural events in the UML. Section 3 outlines how to obtain the simplified OCL expressions that our method will deal with. Section 4 presents our method to determine PSEs that may violate an integrity constraint. Finally, we present our conclusions and point out future work in Section 5.

2. Structural Events in the UML

The main goal of this paper is to determine the set of structural events that may (potentially) violate an integrity constraint defined over the conceptual schema by means of OCL. A structural event is an elementary change in the population of an entity type or a relationship type such as create object, reclassify object, create link, etc.

The precise number and meaning of structural events depends on the particular conceptual modeling language used. In UML, structural events are a subset of the actions defined in the Actions Package [8, p.203+]. Each action is a fundamental unit of behaviour specification. An action takes a set of inputs and converts them into a set of outputs. Structural events correspond to the actions that modify the contents of the IB. They are the only actions that, when applied, may cause the violation of a constraint.

In fact, we are only interested in base structural events. A structural event is base if its effect may not be specified by means of other (base) structural events. Intuitively, it is not difficult to see that determining whether a non-base structural event may violate an integrity constraint can be performed just by considering whether one of the events that define it may violate such constraint.

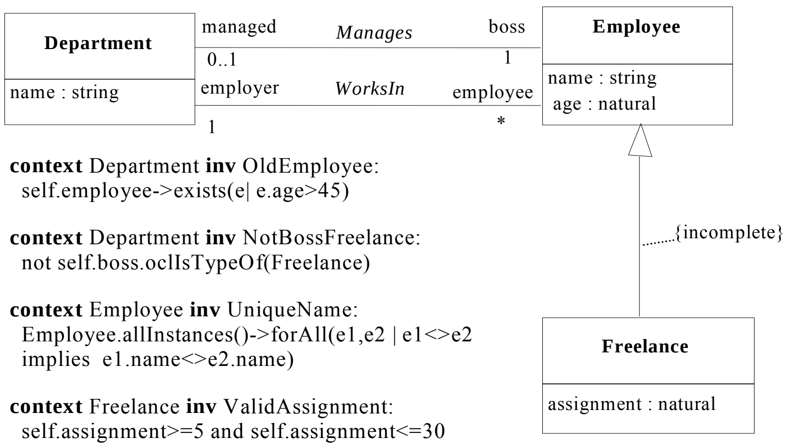
The base structural events that we find in the UML are the following (for more details about them see [8, p.203+]):

- *AddStructuralFeatureAction*: it adds values to a structural feature. It supports also the update of the current value of a structural feature by a new one. A structural feature may represent either an attribute or an association end.
- *CreateLinkAction*: it creates a new link of an association between a set of participants.
- *CreateLinkObjectAction*: it creates a new link when the association is an association class.
- *CreateObjectAction*: it creates a new object as an instance of a specified classifier.
- *DestroyLinkAction*: it destroys links and link objects
- *DestroyObjectAction*: it destroys an object.
- *ReclassifyObjectAction*: it replaces the current classifiers of an object by a new set of classifiers.
- *RemoveStructuralFeatureValueAction*: it removes a value from a structural feature.

We show in Figure 2.1 an example that will be used throughout the paper. It contains a conceptual schema that represents information about the departments of a company and their employees, which may be either freelance or not, and four integrity constraints specified in OCL. Those constraints ensure that each department has a worker older than 45 (constraint *OldEmployee*), that the department boss may not be a freelance (*NotBossFreelance)*, that two different employees may not have the same name (*UniqueName*), and that the assignment of each freelance must be between 5 and 30 hours (*ValidAssignment*).

We may be interested to specify also two different operations to update the contents of the previous conceptual schema. Since UML provides only a metamodel for structural events but no concrete syntax to define them, we have used in our examples the syntax proposed in [11]. Next to each concrete action we add the equivalent structural event of the UML.

ContractFreelance is aimed at contracting a new freelance for a given department. It creates a new instance for the classifier *Freelance*, initializes its values and relates it with the department passed as a parameter to the operation. On the other hand, *FireEmployee* fires an employee (either a normal employee or a freelance), deleting also its relationship with the department.



```
context Department inv OldEmployee:
  self.employee->exists(e| e.age>45)

context Department inv NotBossFreelance:
  not self.boss.oclIsTypeOf(Freelance)

context Employee inv UniqueName:
  Employee.allInstances()->forAll(e1,e2 | e1<>e2
  implies  e1.name<>e2.name)

context Freelance inv ValidAssignment:
  self.assignment>=5 and self.assignment<=30
```

```
context System::ContractFreelance(name: string, age: natural, assig: natural, department: string)
   create object instance f of FREELANCE; -- CreateObjectAction
   f.name=name; --AddStructuralFeatureAction
   f.age=age; --AddStructuralFeatureAction
   f.assignment=assig; --AddStructuralFeatureAction
   select one d from instances of DEPARTMENT where selected.name=department;
   relate f with d across WORKSIN; -- CreateLinkAction

context System::FireEmployee(name:string)
  select one e from instances of EMPLOYEE where selected.name=name;
  select one d related by e->DEPARTMENT;
  unrelate e from d across WORKSIN; -- DestroyLinkAction
  delete object instance e; -- DestroyObjectAction
```

Figure 2.1 - Example of a conceptual schema

3. Simplifying OCL Expressions

Our method assumes a simplified representation of the OCL expression that defines an integrity constraint. Such a simplified representation may be automatically obtained from the original OCL expression and it does not entail a loss of expressive power on the constraints we may deal with.

First, we reduce the number of different operations that appear in an OCL expression by using some of the equivalences among operations already defined in the OCL Standard Libray [10, ch. 11]. Second, we simplify the structure of the OCL expressions by transforming them into conjunctive normal form. We provide in the appendix the list of substitutions we perform to simplify the OCL expression and describe the rules to transform the expression into conjunctive normal form.

As a result of this transformation, each integrity constraint is a conjunction of disjunctions, where each OCL expression appearing in a disjunction is an expression that evaluates to a boolean type. Obviously, to satisfy the constraint the IB must satisfy each disjunction. A disjunction is satisfied if at least one of its literals is satisfied. The literal of a disjunction may be only a *forAll* iterator over an expression, an arithmetic comparison, an equality comparison between objects or sets, a boolean attribute, the not operator and the *oclIsTypeOf* and *oclIsKindOf* operators over an expression.

Applying the first step to our example, we get the following new expressions to define the constraints *OldEmployee* and *UniqueName* (the rest of constraints remain unchanged):

context *Department* **inv** *OldEmployee:*
self.employee->select(e| e.age>45)->size()>0

context *Employee* **inv** *UniqueName:*
Employee.allInstances()->forAll(e1,e2|not e1=e2 implies not e1.name=e2.name)

As a result of the second step, the expression defining *UniqueName* is converted to conjunctive normal form, resulting in:

context *Employee* **inv** *UniqueName:*
Employee.allInstances()->forAll(e1,e2 | e1=e2 or not e1.name=e2.name)

4. Our Method

A naïve approach to solve the problem of determining the structural events that may violate an OCL integrity constraint would conclude that any insertion, update or deletion over an entity or relationship type referenced within the OCL expression may violate the constraint since, obviously, any modification (insert/update/delete) of a model element not appearing in the expression may not cause its violation.

However, such naïve approach is not precise enough since it includes in the result events that will never violate the integrity constraint. In other words, the set of structural events provided by this naïve solution is a superset of the structural events that may actually violate the constraint. For instance, following this approach we would determine that nine base structural events may violate the constraint *OldEmployee*: insert / delete / update entity type Department, insert / delete / update relationship type WorksIn and insert / delete / update entity type Employee.

However, only four structural events may actually violate the constraint: insertion of Department, deletion and update of WorksIn and update of Employee. Intuitively, it is not difficult to see the other five events may never violate *OldEmployee*. So, the most precise solution contains just a 44% (4 of 9) of the events of the naïve one.

The goal of our method is to substantially improve the results obtained with the naïve solution by determining, at definition time, the exact set of base structural events that may actually violate an OCL integrity constraint. As we will see, our method will obtain in the previous example just the four structural events that may really violate the constraint.

We must note that the events we obtain may violate the constraint but this does not necessarily imply that the constraint is violated every time those events are executed (it depends on the exact parameters of the event at execution time). For this reason, we call the events obtained by our method *potentially violating structural events* (PSEs).

We consider that a state of the IB satisfies an integrity constraint *Ic* if *Ic* does not evaluate to false in that state. Then, we have that a base structural event is a PSE for a given constraint when such event can modify the state of the IB in a way that after the execution of the event the constraint evaluates to false. In a similar way, we assume that a *select* expression selects those elements that evaluate the select condition to true (but not when they evaluate to false nor undefined).

Our method assumes that the OCL expression that defines an integrity constraint is represented as an instance of the OCL metamodel [10, ch.8]. For this reason, we treat the OCL expression as a binary tree where each node represents an atomic subset of the OCL expression (an instance of any metaclass of the OCL metamodel: an operation, an access to an attribute or an association …).

The left child of a node is the source of the node (the part of the OCL expression previous to the node). The right child of a node is the argument of the operation (i.e. the second argument, we can think of the left child, the source, as the first argument) if the node represents a binary operation (such as '>', *union*, '+',…) or the body [1] of the iterator if the node represents a loop expression (a forAll, select…).

We show in Figure 4.1 the constraint *OldEmployee* (*self.employee->select(e| e.age>45)->size()>0)* as an instance of the OCL metamodel.

The operator '>' (represented as an instance of the metaclass *OperationCallExp* having as a referred operation an operation called '>') is the root of the tree. The first child of the root is the source of the operator (*self.employee->select(e| e.age>45)->size()*) whereas the second child is the argument of the operation (the integer literal 0). The first child of the child node is an operation (*size*) with a single child (*select*). The *select* node has two children. The first one is its source, an access to an association end (*employee*) with a last child (the access to the variable *self*). The second one, its body, is the operation '>' between the attribute *age* (left child) and the integer *45* (right child).

[1] The expression that is evaluated for each element in the collection

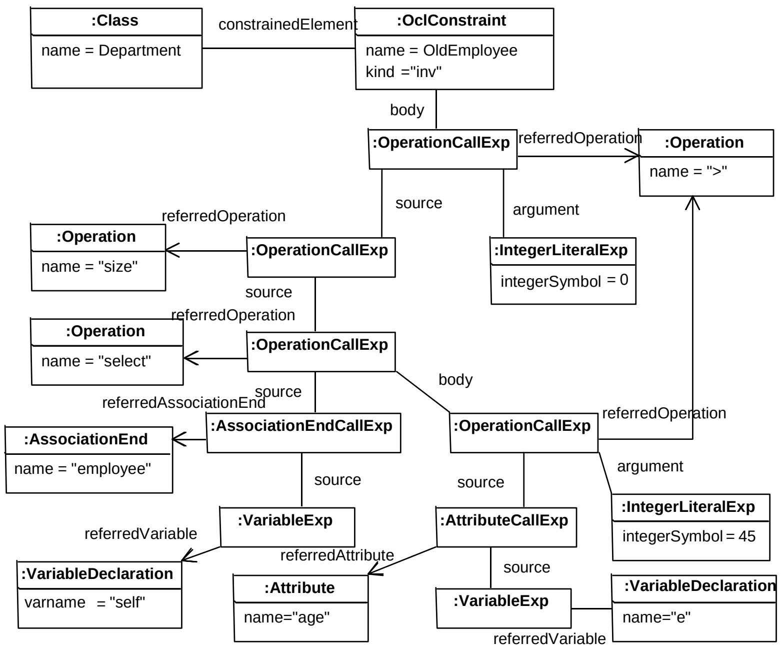


Figure 4.1 – Constraint *OldEmployee* as an instance of the OCL metamodel

Given the binary tree that represents the OCL constraint, our method performs two different steps to determine the base structural events that may violate it:

1. Marking the tree. It is needed to mark each node (i.e. each atomic subset of the OCL expression) with information about its context. This information allows us to discard the events that may not actually violate the constraint.
2. Drawing base structural events. It determines the PSEs by taking the mark and the subexpression corresponding to each node into account.

This section is aimed at explaining these two steps in detail. We need first to introduce a set of internal events we use to determine the set of UML base structural events that may violate an integrity constraint.

We deal with a representative subset of possible OCL expressions. In particular, we cover the whole range of model element, boolean, collection and set operations and loop expressions. However, due to space limitations, we do not address expressions that contain operations over integers, reals or strings (except for the operation '+' chosen as a representant of this group of operations) nor specific operations for Bags, OrderedSets and Sequences.

We assume that taxonomies over relationship types are represented by converting both relationship types to a reified entity type (i.e. an association class) and then defining the taxonomies over them. We also assume that multivalued attributes are represented by means of a binary relationship between the entity type where the attribute is defined and the datatype of the attribute. In this way we do not need to provide a specific reasoning to deal with these two particular constructs since they are already dealt as taxonomies over entity types and as relationship types, respectively.

4.1 Internal Events

To determine the PSEs that may violate an OCL integrity constraint our method reasons about a set of internal events that do not correspond exactly to the base structural events of the UML. Nevertheless, the result of our method in terms of those internal events can be easily translated into the base structural events of the UML. The internal events we use are more basic and precise than those of UML. Besides this, their independence of a particular language allows us to incorporate our results to different sets of structural events providing that we define the correspondence between our internal events and those different sets.

The internal events we use are the following. We state in each case its correspondence with the base structural events of the UML.

- InsertET: insertion over an entity type *et*. It creates a new instance of *et*. The new object can have its attributes initialized but it does not participate in any relationship. It corresponds to a CreateObjectAction over the entity type, a CreateLinkObject (if *et* is an association class), any of them over a subtype of *et* (which induces an insertion over *et*) or by a reclassify action that adds the classifier *et* to an existing instance, plus several AddStructuralFeatureActions to initialize the attributes of the new object. An example is an insertion in the entity type *Employee*, which could be caused either by a CreationObjectAction over *Employee* or *Freelance*.
- UpdateAttribute: it updates the value of an attribute of an entity type *et*. It corresponds to an AddStrucuturalFeature for that attribute (possibly preceded by a RemovalStructuralFeatureAction) over an instance of *et* or any of its subtypes or supertypes. Example: a change in the salary of an employee.
- DeleteET: it deletes an instance of an entity type *et*. The corresponding actions are: a DestroyObjectAction over *et*, a DestroyLinkAction (if *et* is an association class), any of them over a supertype or subtype of *et* (both induce the deletion of the instance over *et*) or by a reclassify action that removes the classifier *et* from an existing instance. Example: the deletion of an employee.
- SpecializeET: it specializes an instance of a supertype of an entity type *et* to *et*. It is equivalent to a ReclassifyObjectAction with an empty set of old classifiers and only the entity type *et* in the set of new classifiers. Example: an employee becoming a freelance.
- GeneralizeET: it generalizes an instance of a subtype of an entity type *et* to *et*. It is equivalent to a ReclassifyObjectAction with an empty set of new classifiers and a direct subtype of *et* in the set of old classifiers. Example: an employee that finishes working as a freelance but remains as employee.
- InsertRT: creation of a new link in a relationship type *rt*. It can be produced by a CreateLinkAction or a CreateLinkObject (if *rt* is an association class) over *rt*. Example: the assignment of an employee to a department.

- UpdateParticipant: it updates one of the participants of a link of a relationshiptype *rt*. It corresponds to an AddStructuralFeature for the association end of that participant over a link of *rt*. Example: a change of the department boss.
- DeleteRT: it deletes a link of a relationship type *rt*. The equivalent action is a DestroyLinkAction over *rt*. Example: to remove an employee from a department.

4.2 Marking the Tree

To compute the set PSEs it is not enough to examine each part of the OCL expression separately. For instance, to determine whether constraint *OldEmployee* may be violated either by an employee assignment or by an employee dismission we may not take into account just the subexpression *self.employee->select(…)->size()*. In fact, both events may change the resulting value of evaluating this subexpression. However, since after the *size* operation we find the '>' operator, only firing an employee may induce the violation of this constraint. On the contrary, if we had used '<=' instead, the expression could be violated by employee assignments.

Each node of the binary tree that represents the OCL expression is marked to indicate which information of the node must be propagated to its children. There are four different symbols to propagate:

- '+': it indicates that the constraint can be violated by an increase in the value or in the number of items of the expression
- '-': it indicates that the constraint can be violated by a decrease in the value or in the number of items of the expression
- '**u**': it indicates that the constraint can be violated by a change in the value or in the items of the expression
- '**und**': it indicates the node does not propagate any kind of information

As an example of the first two symbols consider the operation '>'. A node representing a call to this operation propagates the symbol '-' to the left child (i.e. the first argument) and the symbol '+' to the right child (the second argument). The semantics of this operation justifies this propagation. To violate an expression like *'A > B'* there are two options: decrease the value of *A* (this is why we propagate the symbol '-' to the left) or increase the value of *B* (this explains the '+').

The symbol 'u' is used, for instance, when accessing an attribute. A reference to an attribute can be violated due to an update of the value of the attribute. The same happens with the *select* iterator. The result of a *select* can differ not only because an insertion or deletion on the collection where we apply the *select*. It can also change if we replace any of the objects of the collection (maybe the old object was not selected for the *select* expression but the new one is or vice versa). As an example, imagine that we replace the employee *e1* by the employee *e2* in the department *d1*. The number of employees of *d1* remains constant but if *e1* was the only employee older than 45 years old and *e2* is younger than that age, constraint *OldEmployee* will be violated since no employee will be returned by the select expression.

Finally, the symbol 'und' is used by operations like *'and'* or *'or'*. It denotes that the node does not influence its children expressions at all. The events that can violate '*A and B'* are the same that violate *A* plus those of *B* stand-alone.

To mark the nodes of the tree we traverse the tree in preorder. In a preorder traversal we process all nodes of the tree by first processing the root and then, recursively, processing in preorder the children subtrees. In each node, we take into account the kind of node and the information received from its parent node to decide which information propagates the node to its children.

Table 4.1 shows the symbol propagation for each kind of node and symbol. Sometimes we propagate more than one symbol. In such a case, the final value is obtained by applying the table information to each received symbol. When a cell contains *n/a* (not applicable) it means no constraint exists that includes such combination. When a node has two children the cell states the symbol (or symbols) for each child.

	1. and,or	2. >=,>	3. <,<=	4. =	5. not	6.oclI sType	7. attribute	8. forAll	9.assEnd, assClass[2]
1. und	und und	- +	+ -	+-u +-u	und	+u	+u	+u und	n/a
2. +	n/a	n/a	n/a	n/a	n/a	n/a	+u	n/a	+
3. -	n/a	n/a	n/a	n/a	n/a	n/a	+u	n/a	-
4. u	n/a	n/a	n/a	n/a	n/a	n/a	u	n/a	u

	10. select	11. size	12. sum	13. collect	14 U,∩ count	15. - (set)	16. allInstances	17.var or ct[3]	18. + (Integers)
1. und	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
2. +	+u und	+	+u	+ +	+ u +u	+u -u	und	und	+-[4] +-
3. -	-u und	-	-u	- -	-u -u	-u +u	und	und	+- +-
4. u	u und	n/a	n/a	u u	u u	u u	und	und	u u

Table 4.1 - Marking the binary OCL tree

We show in Figure 4.2 the result of marking the binary tree that represents the constraint *OldEmployee* (a simplified version of the tree of Figure 4.1). Next to each node we add information about the cell (or cells) used to process that node. *CX.Y* means that we access the cell at row *X* and column *Y*.

We start with the operation '>'. Since it is the root of the tree it does not receive any initial information. To mark its children we use the cell 1.2 (row 1, column 2) which states that the left child must be marked '-' while the right must be '+'. The *size* operation receives the symbol '-'. Thus, cell 3.11, we propagate the '-' to its child (the *select*). The *select* sends (cell 3.10) the symbols *–u* to its source (*employee*) and *und* to its body. *Employee*, in its turn, propagates *–u* (column 9 rows 3 and 4) to the variable self. The rest of the tree is marked in a similar way.

[2] A navigation through an association end or an association class
[3] Node that represents any variable or constant (literal) appearing in the constraint.
[4] If we add natural values we only need to propagate the symbols '+' on row 2 and '–' on row 3. However, when adding integers, since they can be negative, we propagate both symbols.

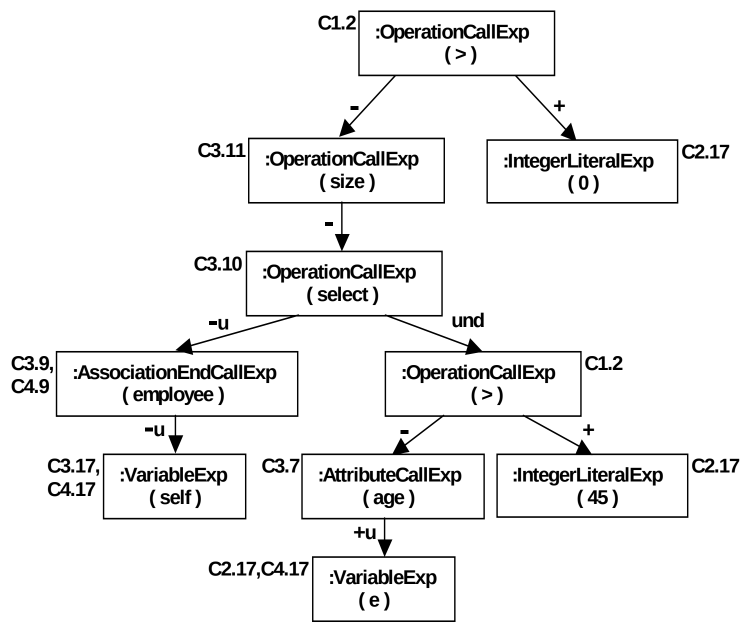


Figure 4.2 - Marking the constraint *OldEmployee*

4.3 Drawing Base Structural Events

Once the tree is marked as explained in the previous section, we may determine the PSEs that may violate the integrity constraint. They are computed as the set of PSEs for the root of the constraint plus, recursively, the set of PSEs for each child node of the root. Hence, we need now to traverse the tree in postorder which implies to process all nodes of the tree by first recursively processing in postorder the children subtrees and then the root.

Table 4.2 describes the set of PSEs we determine for each node in terms of the node type and its mark. Notation: *s(c1)* indicates that the set of PSEs for that node is the sum of the PSEs for its child, and so *s(c1,c2)* when the node has two children. *s(c1) + X* denotes that the node adds the event *X* to the set of PSEs of that node. Blank cells indicate the node does not affect the computation of the PSEs. In addition, to make the table clearer, we use shorthands to indicate the internal events: iET (insertET), uAt (updateAttribute), dET (deleteET), iRT (insertRT), uPa (updateParticipant), dRT (deleteRT), speET (specializeET), genET (generalizeET).

Of particular interest is the function *opp(X)*. This function is used to denote that the set of PSEs for a node is the opposite of the set of PSEs returned by its children. This is the case of nodes representing the *not* operator. The set of PSE for a *not* operator over an expression is defined as just the opposite of the set of PSEs for the expression stand-alone.

A similar thing happens with a *select* expression. An event that can violate the body of a *select* may decrease the number of elements returned by the *select* expression. Therefore, when we need to obtain the set of PSEs that increase the number of selected elements we apply the opposite function to the set of PSEs of the *select* body. The opposite of an insertion is a deletion and vice versa. The opposite of an update event is the event itself. Therefore, the opposites for each event are: opp(iET)=dET, opp(uAt)=uAt, opp(dET)=iET, opp(iRT)=dRT, opp(dRT)=iRT, opp(uPa)=uPa.

	1. and,or	2. >=,>	3. <,<=	4. =	5. not	6.oclIs Type	7. attribute[5]	8. forAll	9.assEnd , assClass
1.und	s(c1,c2)	s(c1,c2)	s(c1,c2)	s(c1,c2)	opp(c1)	s(c1)+ genET	s(c1)+ uAt	s(c1,c2)	n/a
2. +	n/a	n/a	n/a	n/a	n/a	n/a	s(c1)+ uAt	n/a	s(c1)+iRT[6]
3.-	n/a	n/a	n/a	n/a	n/a	n/a	s(c1)+ uAt	n/a	s(c1)+dRT[7]
4.u	n/a	n/a	n/a	n/a	n/a	n/a	s(c1)+ uAt	n/a	s(c1)+uPa

	10. select	11. size	12. sum	13. collect	14. U,∩ count	15. - (Set)	16. allIn stances	17.var or ct	18. +
1. und	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
2. +	s(c1, opp(c2))	s(c1)	s(c1)	s(c1,c2)	s(c1,a1)	s(c1,c2)	iET		s(c1,c2)
3.-	s(c1,c2)	s(c1)	s(c1)	s(c1,c2)	s(c2,a1)	s(c1,c2)	dET		s(c1,c2)
4. u		n/a	n/a						

Table 4.2 - Determining the set of PSEs

As an example, we discuss the possible options when the node represents a reference to an association end involved in a navigation of the OCL expression (column 9 of Table 4.2). When the association end is labeled with a '+' the constraint may be violated by an insertion over the association (event insertRT) where the association end belongs. Remember that '+' pointed out that the constraint can be violated due to an increase of the number elements obtained through a navigation over the association using that association end, and thus, the events that can cause the violation are those events that increase such number. That is precisely what the event insertRT does.

In a similar way, if it is labeled with a '-', the critical event is the deletion of a link of the association (event deleteRT) since we are interested in reducing the number of links of the association. Finally, if it is labeled with the symbol 'u' the problematic event is the replacement of a participant in that association end (updateParticipant) even if the total number of elements does not vary.

Figure 4.3 applies table 4.2 to our example. Since the traversal is in postorder we start by processing the leaves of the tree. First, we process the access to the variable self, already marked with the symbols '-' and 'u'. We must then consider column 17 rows 3 and 4 of Table 4.2 which states that this subexpression does not produce any PSE. After this step, we consider the association end (*employee)* using row 3 and 4 of column 9 to initialize the set of PSE with the events insertET(*Department*, in this case), deleteRT(*WorksIn*) and updateParticipant (association end *WorksIn-Employee*). Next, we process the second child of the *select* expression (its body, e.*age>45*),

[5] When the attribute is preceded by *self* the event iET is added to the set of PSEs.

[6] If the association end is referenced inside the body of a select expression, the event dET is added to the set of PSEs

[7] When *self* precedes the the association end the event iET is added to set of PSEs

which produces the event updateAttribute (attribute *age* of entity type *Employee*). After that, we analyse the *select* itself adding the events generated by its two children. The execution will continue with the process of the *size* operation (no additional events) and the integer constant 0, to end up with the root of the tree, the operation '>' that returns the final set of PSEs for the whole expression.

At the end of this process, we have that the constraint *OldEmployee* may be violated by the internal events: updateAtribute(*age*), updateParticipant(*WorksIn-Employee*), deleteRT(*WorksIn*) or insertET(*Department*). Note that a deleteET over *Employee* does not violate the constraint. It is the deletion of the link between the employee and the department (deletion that can be a preliminary before deleting the employee itself), which may violate it.

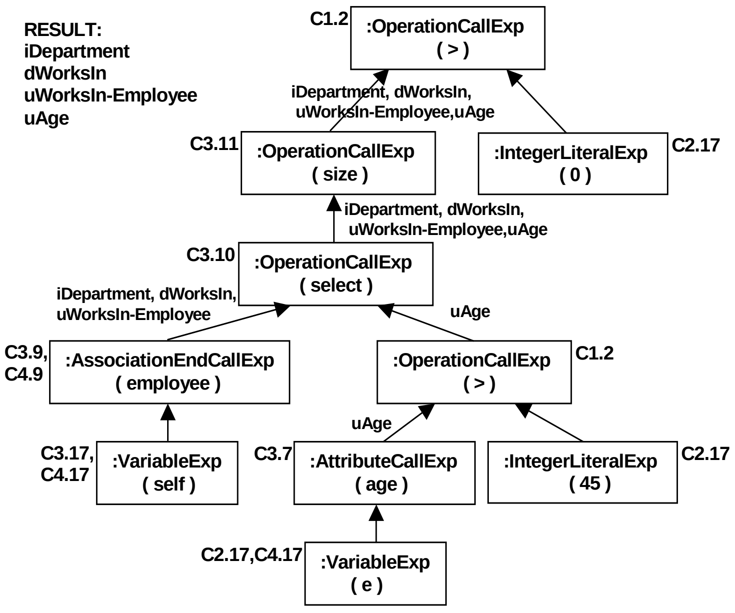

RESULT:
iDepartment
dWorksIn
uWorksIn-Employee
uAge
C1.2
:OperationCallExp (>)
iDepartment, dWorksIn, uWorksIn-Employee,uAge
C3.11
:OperationCallExp (size)
:IntegerLiteralExp (0)
C2.17
iDepartment, dWorksIn, uWorksIn-Employee,uAge
C3.10
:OperationCallExp (select)
iDepartment, dWorksIn, uWorksIn-Employee
uAge
C3.9, C4.9
:AssociationEndCallExp (employee)
:OperationCallExp (>)
C1.2
uAge
C3.17, C4.17
:VariableExp (self)
C3.7
:AttributeCallExp (age)
:IntegerLiteralExp (45)
C2.17
C2.17,C4.17
:VariableExp (e)

Figure 4.3 - Computing PSEs for *OldEmployee*

4.4 Applying the Method

After computing the set of PSEs for each constraint we compare its set of events with the events that appear in the operation specification to see which is the exact set of constraints each operation may violate.

There is only one step missing. The set of PSEs is described using our internal events while the system operation is specified using the set of external events provided by the specific conceptual modelling language used, UML in this case. Therefore, before doing the comparison, we transform the set of PSE into the set of corresponding external events using the rules described in section 4.1

As an example, we apply the method to the whole CS of figure 2.1, obtaining the following results:

1. Set of PSE for each constraint:
    a. *OldEmployee*: updateAtribute(*Age*), updateParticipant(*WorksIn-Employee*), deleteRT(*WorksIn*), insertET(*Department*)
    b. *NotBossFreelance*: specializeET(*Freelance*), insertRT(*Manages*), updateParticipant(*Manages-Boss*).
    c. *UniqueName*: insertET(*Employee*), updateAttribute(*Name-Employee*)
    d. *ValidAssignment*: insertET(*Freelance*), updateAttribute(*Assignment*).
2. Once transformed into the structural events of the UML, the set of events is:
    a. *OldEmployee*: AddStructuralFeature over the attribute *age*, and AddStructuralFeature over the association end *Employee*, a DestroyLinkAction over the association *WorksIn* and a CreateObjectAction over *Department*.
    b. *NotBossFreelance*: ReclassifyObjectAction adding the classifier *Freelance* to a non-freelance *Employee*, CreteLinkAction over *Manages* and an AddStructuralFeature over the associationEnd *boss*.
    c. *UniqueName*: CreateObjectAction over *Employee*, CreateObjectAction over *Freelance* and AddStructuralFeatureAction over the attribute *name*.
    d. *ValidAssignment*: CreateObjectAction over *Freelance* and AddStructuralFeatureAction over the attribute *assignment*.
3. Finally, with this information we can determine which constraints may be violated by each operation
    a. *ContractFreelance* may only violate *ValidAssignment* and *UniqueName* but not the other two constraints.
    b. *FireEmployee* may only violate *OldEmployee*

We would like to remark that with our method we provide an important efficiency improvement to integrity checking. As seen in the previous example, instead of checking all four constraints after the execution of each operation, we have that after *ContractFreelance* we only have to check two of them and just one after *FireEmployee*.

5. Conclusions and Further Work

We have proposed a new method to determine whether the execution of a given operation may potentially violate an integrity constraint. This is done by determining the structural events that may violate the constraint and comparing them with those events that appear in the operation specification.

The main contribution of our work is the use of this knowledge to check only those constraints that can actually be violated by the execution of an operation. We think this is an important contribution since all existing strategies that pursue an automatic code generation of the information systems from their specification can benefit from this knowledge (since they do not have to consider all constraints but just the relevant ones after each operation execution) to provide more efficient implementations.

Checking integrity constraints efficiently requires at least solving two different problems. The first one is the one we have addressed in this paper. The second one is to provide efficient (incremental) algorithms to check an integrity constraint when we know it can be violated by the execution of an operation. This is a direction in which we plan to continue our work.

Acknowledgements

We would like to thank people of the GMC group for their many useful comments to previous drafts of this paper. This work has been partially supported by the Ministerio de Ciencia y Tecnologia and FEDER under project TIC2002-00744.

References

[1] S. Ceri, J. Widom, "Deriving Production Rules for Constraint Maintenance" , Proc. of the 16th VLDB Conference (VLDB'90), Morgan Kauffman, 1990, pp.566-577.

[2] B. Demuth, H. Hussmann, S. Loecher, "OCL as a Specification Language for Business Rules in Database Applications", Proc..of the 4th UML Conference (UML'01), LNCS 2185, Springer, pp. 104-117.

[3] D.W.Embley, B.D. Kurtz, S.N. Woodfield, "Object-Oriented Systems Analysis. A Model-Driven Approach", Yourdon Press, 302 p.

[4] A. Gupta, I.S. Mumick, "Maintenance of Materialized Views: Problems, Techniques and Applications", IEEE Data Engineering Bulletin, Volume 18, Number 2, June 1995, pp. 3-18

[5] ISO/TC97/SC5/WG3, "Concepts and Terminology for the Conceptual Schema and Information Base", J.J. van Griethuysen (ed.), March.

[6] S.J. Mellor, "Executable UML: A Foundation for Model Driven Architecture", Addison-Wesley, 2002

[7] A. Olivé. "Time and Change in Conceptual Modeling of Information Systems". In Brinkkemper, S.; Lindencrona, E.; Solvberg, A. "Information Systems Engineering. State of the Art and Research Themes", Springer, 2000, pp. 289-304.

[8] OMG, "UML 2.0 Superstructure Specification", OMG Adopted Specification.

[9] OMG."MDA Guide Versión 1.0.1", http://www.omg.org/docs/omg/03-06-01.pdf

[10] OMG, "UML 2.0 OCL", http://www.omg.org/docs/ptc/03-10-14.pdf, OMG Adopted Specification

[11] Project Technology, Object Action Language Manual, http://www.projtech.com/pdfs/bp/oal.pdf, visited March 2004.

[12] B. Thalheim, "Entity-Relationship Modeling", Foundations of Database Technology, Springer, 627 p.

[13] R. Wieringa, "A survey of structured and object-oriented software specification methods and techniques". ACM Computing Surveys, 30(4), 1998, pp. 459-527.

Appendix A

To obtain the simplified representation of the OCL expression that defines an integrity constraint we perform the following steps. First, we reduce the number of different operations that appear in an OCL expression by using some of the equivalences among operations already defined in the OCL Standard Libray [10, ch. 11]. Second, we simplify the structure of the OCL expressions by transforming them into conjunctive normal form.

A.1 Reducing the number of different operations

The following equivalences (taken mainly from the OCL Standard Libray [10, ch. 11]) allow to simplify the OCL expressions by reducing the number of operations we consider. In each case, we replace the left part of the equivalence by the expression appearing in the right part.

- Boolean type:
  - <> ↔ not =
  - X = true ↔ X
  - Y = false ↔ not Y
  - X=Y ↔ (X and Y) or (not X and not Y)
- Collection type:
  - collection->includes(Obj) : Boolean ↔ collection->count(Obj)>0
  - collection->excludes(Obj): Boolean ↔ collection->count(Obj)=0
  - collection->includesAll(c2:collection): Boolean ↔ c2->forAll(x| collection->count(x)>0)
  - collection->excludesAll(c2:collection): Boolean ↔ c2->forAll(x | collection->count (x)=0)
  - collection -> isEmpty() ↔ collection->size()=0
  - collection->notEmpty() ↔ not collection->size()=0
- Predefined iterators over collection types:
  - collection->exists(expr) ↔ collection->select(expr)->size()>0
  - collection->reject(expr) ↔ collection->select(not expr)
  - collection->one(expr) ↔ collection->select(expr)->size()=1
- Set type:
  - set->including(Obj): Set ↔ set->union(Set{Obj})
  - set->excluding(Obj): Set ↔ set->- (Set{Obj})

We also define two general syntactical equivalences:

- Self inclusion: we explicitly add the variable self wherever it is omitted
- Collect operation: we use the collect operation when possible. So, we replace expressions of the form *self.A.b* (where *A* represents a navigation with multiplicity higher than 1 and *b* is an attribute) by *self.A>collect(b).*

A.2 Transforming to conjunctive normal form

A logical formula is in conjunctive normal form if it is a conjunction (sequence of ANDs) consisting of one or more clauses, each of which is a disjunction (OR) of one or more literals (or negated literals).

OCL expressions that form the body of OCL constraints can be regarded as a kind of logical formula since they can be evaluated to a Boolean value. Therefore, we can define a conjunctive normal form for the OCL expressions exactly in the same way as that of the logical formulas. The only difference is the definition of a literal. We consider a literal any subset of the OCL constraint that can be evaluated to a Boolean value and that does not include a Boolean operator (*or*, *xor*, *and*, *not* and *implies*). We say that an OCL constraint is in conjunctive normal form when the OCL expression that appears in its body is in conjunctive normal form. This also applies for OCL expressions appearing in the body of *forAll*, and *select* iterators. For instance, constraint *UniqueName* is not in CNF since it includes an implies operator, and thus, it needs to be transformed.

Any logical formula can be translated into a conjunctive normal form by applying a well-known set of rules. We use the same rules in the transformation of OCL expressions with the addition of a new rule to deal with the *if-then-else* construct. The rules are the following:

1. Eliminate the *if-then-else* construct and the *implies* and *xor* operators using the rules:
   a. A *implies* B ↔ *not* A *or* B
   b. *if* A *then* B *else* C ↔ (A *implies* B) and (*not* A *implies* C) ↔ (*not* A *or* B) *and* (A *or* C)
   c. A *xor* B ↔ (A *or* B) *and not* (A *and* B) ↔ (A *or* B) *and* (*not* A *or not* B)
2. Move *not* inwards until the negations be immediately before literals by repeatedingly use the laws:
   a. *not* (*not* A) ↔ A
   b. Morgan's laws: *not* (A *or* B) ↔ *not* A *and not* B
      not (A *and* B) ↔ *not* A *or not* B
3. Repeteadly distributive *or over and* by means of:
   a. A *or* (B *and* C) ↔ (A *or* B) *and* (A *or* C)

It is important to note that in this transformation we do not need to use any Skolemization process to get rid of existencial quantifiers since all free variables that appear in OCL expressions are assumed to be universally quantified.