Computing the Relevant Instances that May Violate an OCL constraint

Jordi Cabot[1,2] and Ernest Teniente[2]

[1]Estudis d'Informàtica i Multimèdia, Universitat Oberta de Catalunya
jcabot@uoc.edu
[2] Dept. Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya
teniente@lsi.upc.edu

Abstract: Integrity checking is aimed at efficiently determining whether the state of the information base is consistent after the application of a set of structural events. One possible way to achieve efficiency is to consider only the relevant instances that may violate an integrity constraint instead of the whole population of the information base. This is the approach we follow in this paper to automatically check the integrity constraints defined in a UML conceptual schema. Since the method we propose uses only the standard elements of the conceptual schema to process the constraints, its efficiency improvement can benefit any implementation of the schema regardless the technology used.

1. Introduction

A conceptual schema (CS) must include the definition of all relevant integrity constraints (ICs) [5] since they state conditions that each state of the information base (IB) must satisfy.

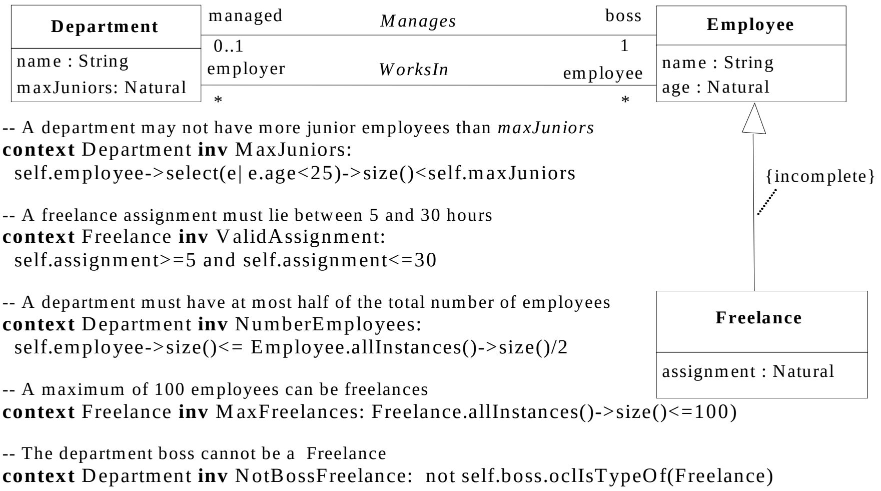
The content of the IB changes due to the execution of operations. In general, the effect of an operation over the IB may be specified by means of a set of structural events (see for instance [6], [10]). A structural event is an elementary change in the population of an entity type or relationship type such as: create object, delete object, update attribute, create link, etc.

The state of the IB resulting from the execution of an operation must be consistent with regards to the set of ICs specified over the CS. The traditional approach to deal with this problem is to reject those operations whose application would lead to an inconsistent state of the IB. This approach is usually known as integrity constraint checking and it requires verifying efficiently that the IB state obtained as a consequence of an operation execution does not violate any integrity constraint. For the sake of simplicity, we assume, without loss of generality, that each operation constitutes a single transaction and use both terms indistinctly.

In this paper we propose a new method to improve efficiency of integrity constraint checking in CSs specified in the UML [9] where constraints are written in the OCL [8]. Note that, as shown in [3], OCL can also be used to represent the graphic constraints expressed in the UML diagrams.

Constraints in OCL are defined in the context of a specific entity type, the *context entity type* (CET), and must be satisfied by all instances of that entity type. However, when verifying an IC not all instances must be taken into account since, assuming as usual that the IB is consistent before the update, only those that have been modified by the set of structural events applied over the IB may violate the IC.

Consider the running example of Fig. 1.1 to illustrate these ideas. After executing an operation that hires a junior employee (i.e. an employee under 25), not all departments must be taken into account to verify the constraints *MaxJuniors* and *NumberEmployees*. In fact, since the IB is assumed to be consistent before the operation execution, only the department where the employee starts working in may induce a violation of one of those integrity constraints.



```
-- A department may not have more junior employees than maxJuniors
context Department inv MaxJuniors:
  self.employee->select(e| e.age<25)->size()<self.maxJuniors

-- A freelance assignment must lie between 5 and 30 hours
context Freelance inv ValidAssignment:
  self.assignment>=5 and self.assignment<=30

-- A department must have at most half of the total number of employees
context Department inv NumberEmployees:
  self.employee->size()<= Employee.allInstances()->size()/2

-- A maximum of 100 employees can be freelances
context Freelance inv MaxFreelances: Freelance.allInstances()->size()<=100)

-- The department boss cannot be a  Freelance
context Department inv NotBossFreelance:  not self.boss.oclIsTypeOf(Freelance)
```

Fig. 1.1 - Example of a conceptual schema

Given a conceptual schema CS with a set of integrity constraints IC, our method generates a CS' with additional entity types, required to record the structural events issued by the operation, and where the definition of the original ICs has been modified to be able to verify them only in terms of the relevant instances. Moreover, the way we compute the relevant instances ensures that a constraint will not be verified if no structural event that may violate it has been issued by the operation.

In addition to the efficiency improvement, the main advantage of our method is that it works at the conceptual level, i.e. it is technology-independent, since the result of our method is a standard conceptual schema. Then, any architecture able to treat a CS to generate automatically its implementation can benefit from our method, no matter the target technology platform it generates. Pre-processing the original CS with our method allows any code-generation architecture to automatically generate efficient integrity constraints that are verified only in terms of its relevant instances.

The problem of efficient integrity checking has been widely addressed. However, as far as we know, ours is the first proposal to cope with this issue at a specification level. Previous work addressing similar problems is always particular for a given technology. The best approaches are proposed in the fields of deductive [4] or relational [2] databases.

The work presented here extends our previous work in [1] where we proposed a method to compute the structural events that may violate an integrity constraint. However, that work did not care about how to check integrity constraints efficiently when one such structural event was issued by a transaction. This is precisely the main concern of this paper.

The paper is organized as follows. Section 2 introduces some basic concepts. Section 3 classifies the different kind of constraints according to the efficiency level our method can provide. In particular, our method improves the efficiency of instance constraints (section 4) and partial instance constraints (section 5). Finally, section 6 presents some conclusions and further work.

2. Determining the structural events that may violate an IC

The first thing we need to take into account when computing the relevant instances of an integrity constraint is to determine the set of structural events that may cause its violation, i.e. its set of potentially violating structural events (PSE).

To compute the set of PSEs we consider the following kinds of structural events:

- InsertET(*ET*): insertion over the entity type *ET*.
- UpdateAttribute(*Attr*,*ET*): it updates the value of the attribute *Attr*.
- DeleteET(*ET*): it deletes an instance of the entity type ET.
- SpecializeET(*ET*): it specializes an instance of a supertype of the entity type *ET* to *ET*.
- GeneralizeET(*ET*): it generalizes an instance of a subtype of *ET* to *ET*.
- InsertRT(*RT*): creation of a new link in the relationship type *RT*.
- DeleteRT(*RT*): it deletes a link of the relationship type *RT*.

For instance, the event *InsertET(Freelance)* is a PSE for *ValidAssignment* since the new freelance may have an assignment below 5 or over 30, and thus, it may violate the constraint. On the contrary, the event *DeleteET(Freelance)* is not a PSE for that constraint since it may never induce a violation of *ValidAssignment*.

To compute the PSEs and the relevant instances that may violate an integrity constraint we assume that OCL constraints are represented as an instance of the OCL metamodel [8, ch. 8]. Therefore, we treat the OCL expression of the constraint as a binary tree where each node represents an atomic subset of the OCL expression (an operation, an access to an attribute or an association, etc.).

The root of the tree is the most external operation of the OCL expression. 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 (if any). As an example, Fig. 2.1 shows the constraint *MaxJuniors* (*self.employee->select(e| e.age<25)->size()<self.maxJuniors*) as an instance of the OCL metamodel. The operator '<' is the root of the tree. The left child is the source of the operator (*self.employee->select(e| e.age<25)->size()*) whereas the right child is the access to the attribute *maxJuniors* with a child representing the access to the *self* variable. The rest of the constraint is represented in a similar way.

We use the method proposed in [1] to determine the set of PSEs that may violate an integrity constraint. This method draws those PSEs from the nodes of the OCL expression that defines the constraint by means of examining the elements and operations referred in the constraint as well as its syntactic structure.

As an example, we have that the application of this method over the tree representing the constraint *MaxJuniors* would result in the marked tree of Fig. 2.1. Each node is marked with the set of structural events that may violate the subexpression it represents. For instance, the access to the attribute *maxJuniors* is marked with *UpdateAttribute(maxJuniors,Department)* and *InsertET(Department)* since an update of the attribute *maxJuniors* (in particular a decrease of its value) or the creation of a new department (with more junior employees than permitted) may violate the constraint. Other PSEs for *MaxJuniors* are: *InsertRT(WorksIn)* and *UpdateAttribute(Age,Employee)*.

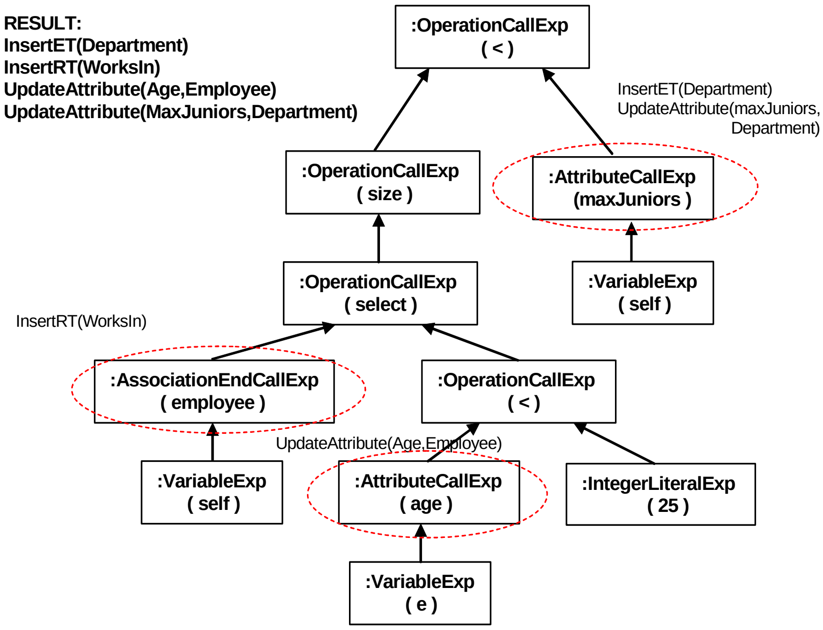

RESULT:
InsertET(Department)
InsertRT(WorksIn)
UpdateAttribute(Age,Employee)
UpdateAttribute(MaxJuniors,Department)
:OperationCallExp (<)
InsertET(Department)
UpdateAttribute(maxJuniors, Department)
:OperationCallExp (size)
:AttributeCallExp (maxJuniors)
:OperationCallExp (select)
:VariableExp (self)
InsertRT(WorksIn)
:AssociationEndCallExp (employee)
:OperationCallExp (<)
UpdateAttribute(Age,Employee)
:VariableExp (self)
:AttributeCallExp (age)
:IntegerLiteralExp (25)
:VariableExp (e)

Fig. 2.1 – Computing the set of PSEs for *MaxJuniors*

3. Constraint classification

After executing a set of operations over the IB, we must verify all constraints having as PSEs some of the structural events included in them. A direct computation of the OCL expression defining a constraint would evaluate it over all instances of the context entity type (CET). However, this is not always necessary and many times we can check a constraint by considering only the relevant instances of its CET (those affected by the set of structural events).

For instance, consider again the constraint *MaxJuniors*. After changing the age of an employee, instead of checking all departments, we only need to verify the departments where the modified employee works in.

In general, we may distinguish three different types of integrity constraints: *instance*, *partial instance* and *class* constraints. We classify a constraint as instance if we can always compute the exact subset of the instances of its CET we need to take into account to check it. A constraint is a class constraint if we always have to consider the whole population of the CET to check the constraint. Finally, in some cases we may need to consider the whole CET population or just a subset depending on the structural events issued during operation execution. In this case we say the constraint is partial instance.

MaxJuniors is a good example of instance constraint. We also have that *MaxFreelances* (*context Freelance inv: Freelance.allInstances()->size()<=100*) is a class constraint. The reason is that after inserting a new freelance we need to access all instances of the entity type *Freelance* to verify the number of freelances is still less than 101. Finally, *NumberEmployees* (*context Department inv: self.employee->size()<= Employee.allInstances()->size()/2*) is a partial instance constraint. Note that if we assign a new employee to a department, we only need to check the constraint over that particular department. However, if we remove an employee, we need to verify all departments, including those where the removed employee did not work.

A constraint can be classified into exactly one of those types just by examining the syntactic structure of the OCL expression defined in the body of the constraint. Intuitively, a constraint will be classified as *instance* if it is defined by means of a contextual instance (i.e. using, implicitly or explicitly, the *self* variable). A constraint will be a *class constraint* if it is defined using the *allInstances* operation. A *partial instance constraint* is a constraint that includes in its definition both the *self* variable and the *allInstances* operation.

To formally classify a constraint within the above categories, we need to introduce the concept of subexpression. In short, a subexpression is a sequence of nodes of the tree representing the OCL expression of the constraint. An OCL expression can consist of several subexpressions. In particular, each node representing an access to a variable begins a different subexpression. The reference to an entity type that precedes the *allInstances* operation is also considered a variable

Then, we can define a subexpression as the sequence of nodes that starts with this initial node and includes all its consecutive parent nodes that are traversed up to the last node of the subexpression. The last node is a node whose parent represents a call to an arithmetic operation, arithmetic or boolean comparison or a loop expression having the last node as its right child.

Fig. 3.1 shows the different subexpressions of the *MaxJuniors* and *NumberEmployees* constraints. An ellipse circles each subexpression.

We distinguish between two kinds of subexpressions: *instance* and *class* ones. A subexpression is considered an instance subexpression when it begins, directly or indirectly, with the *self* variable. A subexpression begins indirectly with the *self* variable when begins with a variable *v* where *v*<>*self* and *v* is defined within a loop expression (*select*, *forAll*…) included in an instance subexpression. Otherwise, the subexpression is considered a class subexpression. The same Fig. 3.1 classifies each subexpression for the example constraints.

Given a constraint *c* we define that *c* is an *instance constraint* when all the subexpressions of *c* including nodes with PSEs are instance subexpressions. We define that *c* is a *class constraint* when all the subexpressions of *c* including nodes with PSEs are class subexpressions. Finally, we define *c* as a *partial instance constraint* when it is neither an instance constraint nor a class constraint, and thus, *c* includes an instance subexpression and a class subexpression, at least. Our method improves the verification of instance and partial instance constraints but not the verification of class constraints (where we always need to examine all the instances).

Applying the previous definitions over the example constraints (Fig. 3.1) we obtain that the constraint *MaxJuniors* is an instance constraint and *NumberEmployees* is a partial instance constraint.

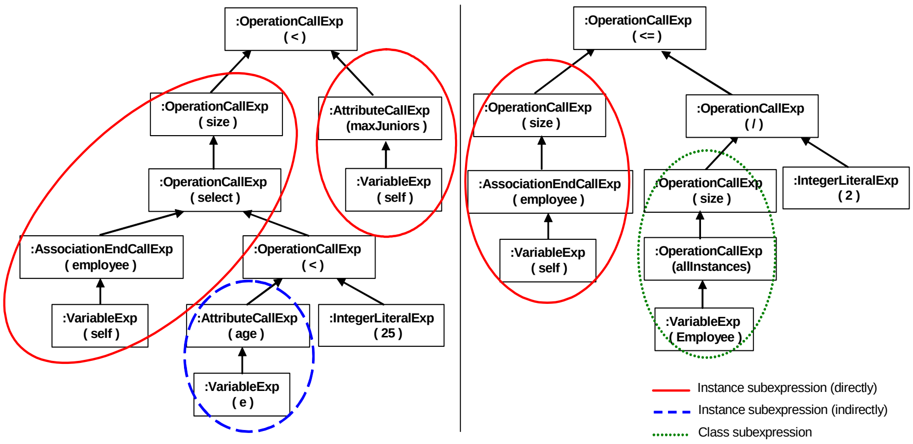

:OperationCallExp (<)
:OperationCallExp (size)
:AttributeCallExp (maxJuniors)
:VariableExp (self)
:OperationCallExp (select)
:AssociationEndCallExp (employee)
:OperationCallExp (<)
:VariableExp (self)
:AttributeCallExp (age)
:IntegerLiteralExp (25)
:VariableExp (e)
:OperationCallExp (<=)
:OperationCallExp (size)
:OperationCallExp (/)
:AssociationEndCallExp (employee)
:OperationCallExp (size)
:IntegerLiteralExp (2)
:VariableExp (self)
:OperationCallExp (allInstances)
:VariableExp (Employee)
Instance subexpression (directly)
Instance subexpression (indirectly)
Class subexpression

Fig. 3.1 – Subexpressions for MaxJuniors (left) and NumberEmployees (right) constraints

4. Processing instance constraints

We explain now the transformation we propose for *instance constraints* to evaluate them only over the relevant instances of the IB. As we have just seen, *instance constraints* must only be evaluated over those instances of the context entity type that may have been affected due to the structural events issued during the transaction, since these are the only instances that can violate the constraint.

Given a constraint *c* defined over a context entity type CET, the basic idea of our transformation process is to create a new derived entity type meant to contain only those instances of the CET that need to be evaluated. More specifically, the population of the new type will be the set of instances of CET affected by the structural events. To compute its population we record that set of structural events in a special kind of entity types, the *structural event types*. Then, the context of the constraint *c* is changed from CET to the new derived type, and thus, the constraint is evaluated only over the relevant instances.

In the following, section 4.1 explains the definition and treatment of the structural event types while section 4.2 explains the creation of the new derived entity type, the computation of its population and the redefinition of the constraint.

4.1 Definition of structural event types

We need to define *structural event types* to record explicitly the structural events. More concretely, these types are devoted to record the information about the modifications produced by the structural events issued during the transaction.

In general, we need to define a *structural event type* for each possible structural event. Therefore we define the following types for each entity type of the CS (see section 2): iET (to record insertion events over the entity type *ET*), dET (for deletion events over the entity type *ET*), gET (a generalize event over *ET*) and sET (an specialize event over *ET*). Additionally, for each attribute of *ET*, we define a structural event type uETAttribute to record the changes in the attribute value. Moreover, for each relationship type *RT* we need to define: iRT (insertion of a new link in *RT*) and dRT (a deletion of a link of *RT*).

Nevertheless, since we simply use these types for dealing with instance constraints, we are only interested in defining the types corresponding to structural events that may be a PSE for that kind of constraints. Therefore, if a structural event cannot violate any of the ICs of the CS, we do not define its corresponding structural event type.

As an example, the list of structural event types we will define for the constraint *MaxJuniors*, according to its set of PSEs (see section 2), is the following: *iDepartment* (insertion of a new department), *iWorksIn* (insertion in the relationship *WorksIn*), uDepartmentMaxJuniors (update of the attribute *MaxJuniors*), and *uEmployeeAge* (update of the attribute *age*).

Note that we never need to define a structural event type for deletion events over entity types since this event is never included in the set of PSEs of an instance constraint. In fact, this kind of structural event can never appear in an instance subexpression. Since an instance subexpression begins (directly or indirectly) with the *self* variable, it is obvious that can not contain the event deleteET when ET=CET (we only evaluate the constraint over existing instances). Moreover, when ET<>CET, ET is accessed from a navigational expression starting with the *self* variable. In such cases it is the deletion of the link between the instance of ET and the previous instance in the navigation that can violate the constraint, not the deletion of the instance itself.

4.1.1 Structure of structural event types

The next question we need to ponder is the internal structure (attributes and relationship types) of the structural event types. They are stereotyped with the stereotype <<structural event>> to differentiate them from the entity types of the CS.

We distinguish between structural event types recording structural events that modify entity types and those that modify relationship types.

The structural event types recording structural events that modify entity types are defined as types without attributes and with just one relationship type relating the structural event type with the corresponding entity type. Through this reference we can access the entity modified by the structural event

The multiplicity of the relationship type between the structural event type and the entity type is 0..1:1. The reason is that an instance of the structural event type must necessarily refer to an instance of its entity type while an instance of the entity type may appear, at most once, in a structural event type. For the sake of simplicity, the role next to the entity type in all those relationship types is always named as *ref*.

Fig. 4.1 shows, as an example, the structural event type for the event insertET(Department). Note that the only information recorded for each instance of *iDepartment* is a reference to the corresponding new department instance in the *Department* type to access its information when needed.

We can opt for this kind of structure because there are no structural event types for deletion events (see section 4.1), and thus, we can always relate the instance of the structural event type with the corresponding entity in the entity type.

In the definition of these types we assume that the IB corresponding to the CS is updated at execution time with the modifications produced by the structural events. Thus, we can avoid redundancies by not including in the structural event type the information about the changes produced by the event over the affected entity (i.e. in the type *iDepartment* we do not include the information about the attribute values of the new department, we just use the reference towards the *Department* type to obtain this information).

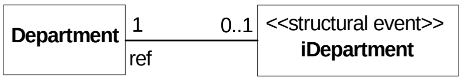


Fig. 4.1 Structural event type for the event insertET over *Department*

In a similar way, the structural event types for structural events over relationship types do not contain attributes either. However, their instances do not refer to the corresponding link of the relationship type but to the set of participants of that link.

Therefore, a structural event type corresponding to a structural event over a relationship type *RT*, contains as many relationship types as the number of participants in *RT*. Each one of these relationship types relates the structural event type with one of the participants of *RT*. Note that the types dRT (deletion of a link of *RT*) are perfectly possible since their instances do not point to the deleted link (which does not already exist in the IB) but to their participants.

As an example, Fig. 4.2 shows the structural event type for the event insertRT(WorksIn). The type *iWorksIn* presents two relationship types, with *Department* and *Employee*, since these entity types are the participants of *WorksIn*.

When defining the multiplicity of the relationship types between the structural event type and the set of participants we distinguish between types for deletion events (dRT) and types for insertion (iRT) events.

For iRT types, the multiplicity of the relationship type is 1:* since, in general, an entity of a participant entity type can participate in many links of the relationship type (for instance, if we assign a set of employees to the same department, several instances of *iWorksIn* will refer to the same department entity) and every instance of the iRT type must be related to an existing entity of the participant entity type.

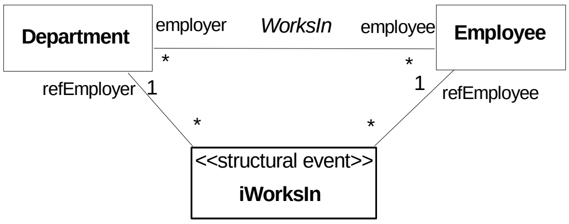


Fig. 4.2 Structural event type for the event insertRT over WorksIn

For dRT types, the multiplicity may become 0..1:*, because, after deleting the link, and thus, creating a new instance in the dRT type, it may happen that other events delete also some of the participants of the link. This is not possible for iRT types since we cannot delete the participant without deleting before the link itself.

Note that we cannot remove the instance of dRT when deleting one of the participants since we may still need the information about the deleted link to compute the relevant instances for constraints including the deletion event as PSE. We can only delete it when all participants are deleted.

The constraints including as a PSE the event deleteRT over a relationship type *R*, either navigate *R* from E_1 to E_2 or from E_2 to E_1, where E_1 and E_2 are the participant entity types of *R*. When, after deleting a link of *R*, the participant E_1 is also deleted, the information about the deleted link is irrelevant for constraints that navigate R from E_1 to E_2. In such a case, it is the deletion of E_1 what must be taken into account. However, for constraints navigating *R* from E_2 to E_1, the deleted link is used to navigate through the affected E_2 participant to obtain the relevant instances for the constraint.

For instance, consider a constraint stating that all departments must have at least three employees. The constraint can be violated by a deletion over *WorksIn*. If we delete the link between a department *d* and an employee *e*, a new instance of *dWorksIn* is created. Even if, afterwards, we also delete the employee *e*, the instance of *dWorksIn* allows us to know that the department *d* needs to be considered when evaluating the constraint.

4.1.2 Instantiating the structural event types

In general, each structural event type will contain as many instances as events of that kind have been executed over the entity or relationship type. For instance, the structural event type *iDeparment* will contain an instance for each new department inserted during the transaction, *uEmployeeAge* an instance for each employee that has changed its age during the transaction, etc.

However, to improve the efficiency of these types we adapt the concept of *net effect* [2] and define two additional rules for insertions and deletions over structural event types:

- Before inserting an instance in an uETAttribute type we must check that the same instance does not appear previously in the types iET or uETAttribubte, as well. For instance, if we update three times the attribute *age* of the same employee during a single transaction, we only record this fact once.
- When deleting an entity or a relation, the corresponding instance is also deleted from the types iET (iRT), gET, sET and uETAttribute if existing. In addition, if the entity (relation) appears in iET (iRT) we do not need to record that it has been deleted. For instance, if we update the age of an employee and later on, during the same transaction, we delete the employee we do not need to worry about its age update. If the employee was inserted in the same transaction we neither record his/her deletion.

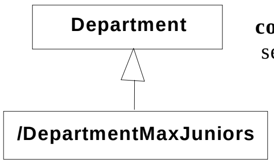
4.2 Constraint redefinition

As we said before, to evaluate an instance constraint only over the relevant instances of its CET we create a new derived entity type meant to contain the exact set of instances of CET that need to be verified.

This new entity type, called *ETConstraint* (i.e. the name of the entity type plus the name of the constraint) is defined as a derived subtype of CET. Then, we replace the original constraint with a new constraint with the same body but having as a context entity type the new type *ETConstraint*. This is possible because, as a subtype, *ETConstraint* contains all attributes and relationship types of its supertype. As an example, Fig. 4.3 includes the redefinition of the constraint *MaxJuniors* over the *Department* entity type.

Note that with this replacement we obtain an efficient evaluation of the constraint, since the population of *ETConstraint* is exactly the set of instances we need to check. In general, the cardinality of the *ETConstraint* type is, by far, lesser than the total number of instances of CET and it can never be greater. Moreover, this approach also avoids redundant checking. The population of an entity type is a set and this ensures that we check each instance only once even if the transaction includes several structural events that affect the same instance.

The last problem we need to address is the computation of the population of the *ETConstraint* entity type, i.e. how to automatically define its derivation rule using the set of structural events recorded in the structural event types explained in the previous section. In short, the population of *ETConstraint* is the union of instances of CET affected by each structural event appearing in the structural event types. Obviously, if the structural event is not a PSE for the constraint no instances of CET are affected.



context DepartmentMaxJuniors **inv** maxJuniors:
self.employee->select(e| e.age<25)->size()<self.maxJuniors

Fig. 4.3. Redefinition of the *MaxJuniors* constraint

4.2.1 Computing the instances of CET affected by a structural event

Intuitively, given an instance *i* of a structural event type *ev* over the entity type *ET*, we obtain the set of instances of CET affected by *i* by doing an inverse navigation from *i* to the instances of CET related with *i*. Roughly speaking, the inverse navigation involves four different steps:

- To select the subexpression of the constraint where the event *ev* is included in.
- To reverse the part of the subexpression affected by the event. We reverse the part of the subexpression that goes from the beginning of the subexpression up to the element affected by the event *ev*.
- To remove from the previous result all the elements except for the navigations over relationship types.
- For each navigation appearing in the reversed subexpression, to navigate through the same relationship type but in the opposite direction by means of using the opposite role.

As an example, consider the event UpdateAttribute(Age,Employee) over the constraint *MaxJuniors* (*context department inv: self.employee->select(e| e.age<25)->size()<self.maxJuniors*). This event affects the *select* operation of the constraint so we need to take into account the subexpression *self.employee->select(e|e.age<25)*. The subexpression only contains the navigation through the relationship type *WorksIn* using the *employee* role. Therefore, to obtain the affected departments after the age update, we just navigate from the updated employee to the departments related with him through the same *WorksIn* relationship type but using the opposite role (the *employer* role).

Formally, assume a constraint defined over a CET with a PSE attached to the node n_i of the tree representing the constraint and where n_i is included, at least, in an instance subexpression *sub* where $sub = [n_0, n_1, n_2, \ldots, n_{i-1}, n_i, n_{i+1}, \ldots n_n]$. n_0 is the initial node of the subexpression and n_n the last one. We obtain the set of instances of CET affected by an execution of the PSE by, first, reversing the sequence of nodes $[n_0, n_1, n_2, \ldots, n_{i-1}, n_i]$ to obtain the sequence $[n_i, n_{i-1}, \ldots, n_2, n_1, n_0]$. Note that the reversed sequence starts with the node responsible for the PSE.

Then, we delete from the sequence all nodes that do not represent a navigation through a relationship type (i.e. all nodes not representing an access to an association end). Finally, for each remaining node, we replace the node with another node representing an access to the opposite association end.

When the node n_i appears in an indirectly instance subexpression *sub* we need to concatenate the nodes of *sub* with those of its *parent* subexpression and repeat the process until we reach the node representing the initial *self* variable. The *parent* subexpression is the subexpression containing the iterator where *sub* is included. More concretely, if the parent subexpression is of the form $parent=[p_0, p_1, p_2, \ldots, p_{i-1}, it, p_{i+1}, \ldots, p_n]$ where *it* is the iterator where *sub* is included in, the result of the concatenation is $result=[p_0, p_1, p_2, \ldots, p_{i-1}, n_0, n_1, \ldots n_{i-1}, n_i]$, and after reversing the order of nodes $[n_i, n_{i-1}, \ldots, n_2, n_1, n_0, p_{i-1}, \ldots, p_2, p_1, p_0]$.

If the same PSE appears in different instance subexpressions or the node n_i is included in several ones we repeat the process for each of them.

As an example, we apply the formalization to obtain the set of departments we need to check in the *MaxJuniors* constraint after the event UpdateAttribute(Age,Employee). The subexpression *sub* where the event is included is *sub = [e, age]* (see Fig. 3.1). Since this is an indirectly instance subexpression we must concatenate it with its parent subexpresssion (*parent=[self, employee, select, size]*). The resulting subexpression is *[self, employee, e, age]*, and once reversed *[age, e, employee, self]*. We remove all the irrelevant nodes to obtain the sequence *[employee]* and, once replacing the node by the opposite role, we obtain the final result *[employer]*, where *employer* is the opposite association end of *employee*, the only node representing an access to an association end.

Therefore, to obtain the affected departments we need to apply the obtained subexpression (*[employer]*) over each updated employee (i.e. each instance of *uEmployeeAge*). For instance, if the *uEmployeeAge* type contains an instance *e1*, we access the updated employee using *e1.ref* (see section 4.1) and then we obtain the affected departments using the expression *e1.ref.employer*.

As a more complex example, consider a constraint stating that an employee cannot be older than the bosses of the departments where he/she works. This constraint could be expressed as: *context Employee inv: self.employer->forAll(d| d.boss.age> self.age)*. When computing the set of PSEs for the constraint we see that the event UpdateAttribute(Age,Employee) is included in both constraint subexpressions. Thus, to obtain the set of employees we need to check after an age update, we have to apply the previous process over both subexpressions and join the two sets of affected employees.

The first subexpression is *[self, employer, d, boss, age]*, and once reversed *[age,boss,d,employer,self]*. After removing the irrelevant nodes: *[boss,employer]* and the final result, once replacing the nodes with the opposite association ends, is *[managed,employee]*. Therefore, to obtain the affected employees we need to navigate from the updated employee to the department he/she manages (if any), and then, from the department to the employees of that department.

The second subexpression is *[self,age]*. It does not include any navigation, and thus, the final result will be an empty sequence of nodes. This means that given an updated employee, we only need to check that particular employee.

As a final result we obtain that after an age update we need to check the updated employee plus all the employees working in the department he/she manages, if any.

4.2.2 Derivation rule definition

The derivation rule for the *ETConstraint* entity type must ensure that the set of instances of the type be exactly the set of instances we need to check. It must include, for each instance of the structural event types corresponding to the PSEs for the constraint, the computation of the affected CET instances, as explained above. Using the work of [7] we define the population of a derived entity type by means of redefining its predefined *allInstances* operation (i.e. the population of the derived type will be the set of instances returned by the operation).

As an example, consider the previous *MaxJuniors* constraint. In this case, the derivation rule for the derived subtype *DepartmentMaxJuniors* must select, according to the PSEs for the constraint, all new inserted departments (departments recorded in the *iDepartment* structural event type), the departments that have updated its *maxJuniors* attribute (departments appearing in the *uDepartmentMaxJuniors* type) and the departments with new assigned employees (departments participating in a new relationship of the *WorksIn* relationship type, recorded in the *iWorksIn* type), and also, for each employee that has changed his/her age, all the departments where the employee was working in.

This last set of departments is obtained by applying the role *employer* over each updated employee (each instance of the *uEmployeeAge* type) as computed in the previous section.

Therefore, the derivation rule for *DepartmentMaxJuniors* is the following:

context *DepartmentMaxJuniors::allInstances() : Set(Department)*
body: *iDepartment.allInstances().ref->union(*
uDepartmentMaxJuniors.allInstances().ref->union(
iWorksIn.allInstances().refEmployer->union(
uEmployeeAge.allInstances().ref.employer))))->asSet()

Note that, we use the special relationship types between the structural event types and its corresponding entity types to access the modified instances (see section 4.1). For instance, *iDepartment.allInstances().ref*, returns the new departments by accessing the referenced departments from the *iDepartment* type.

In the appendix we show the results of the application of our method over the rest of instance constraints of our example.

5. Processing partial instance constraints

A constraint is classified as a partial instance constraint if it contains at least an instance subexpression and a class subexpression, both including nodes marked with PSEs for the constraint. These constraints can be checked efficiently when the transaction does not include any of the PSEs included in class subexpressions. Otherwise, we must check the constraint over all instances of the CET. For instance, the constraint *NumberEmployees* (*context Department inv: self.employee->size()< Employee.allInstances()->size()/2*) can be checked efficiently after assigning an employee to a department but not after the deletion of an employee.

To process this kind of constraints we split their set of PSEs into two different groups: the set of *instance* PSEs and the set of *class* PSEs, depending on the kind of subexpression where they are included. If a PSE is included in both kinds of subexpressions is considered a class PSE. With the set of instance PSEs we apply exactly the same process explained in section 4 with just a slight difference concerning the derivation rule of the *ETConstraint* entity type, as we explain below.

For the *class* set we also create the structural event types. In fact, we are not interested in knowing the exact instances affected by the class PSEs because we will need to check all instances. We only need to know whether any of the those events has been executed, and thus, we could think about creating a new set of singleton entity types enough to record the presence or absence of each event. However, since probably most structural event types will be already defined to deal with other constraints, we think it is worthwhile to reuse the same set of structural event types.

The only difference relies on the dET kind of entity types, which were not needed before. If there is a deleteET event event among the set of *class* PSEs, we need to create the corresponding dET type. As we have explained before, the instances of this entity type cannot reference the deleted instances since they no longer exist, but this does not suppose a problem since we are not interested in knowing those instances. We just create an empty instance in the dET type.

Afterwards, in a similar way as before, we create a new derived subtype, called *ETConstraint'*, under the context entity type, and change the context of the original constraint to *ETConstraint'*. Its population will be the same population of the context entity type if the transaction has executed any class PSE. Otherwise, its population will be empty. Therefore, the derivation rule for *ETConstraint'* is:

allInstances() = ***if*** *(ev_1.allInstances()->size() + ev_2.allInstances->size() + … ev_n.allInstances()->size() > 0)* ***then*** *CET.allInstances()*

where $ev_1..ev_n$ represent the structural event types corresponding to the *class* PSEs.

Moreover, we change the derivation rule *dr* for the *ETConstraint* type created for the instance PSEs. The new derivation rule will be: *allInstances() =* ***if*** *(ETConstraint'.allInstances()->size()=0)* ***then*** *dr.* This way we ensure that when the transaction includes a class PSE we check all instances of the original context entity type (*ETConstraint'* will contain the same instances as CET) and avoid redundant checkings (*ETConstraint* will be empty). Otherwise, we check the constraint efficiently (*ETConstraint* will contain the affected instances of CET whereas *ETConstraint'* will be empty).

In the appendix we process the partial instance constraint *NumberEmployees*.

6. Conclusions and further work

We have proposed a new method to improve efficiency of integrity constraint checking in CSs specified in UML with constraints written in OCL. As far as we know, ours is the first method to deal with this issue at the specification level.

The basic idea of our method is to record the set of structural events applied over the IB in order to compute the set of relevant instances for each constraint, and then, evaluate the constraint only over those instances, avoiding irrelevant verifications.

We believe this efficiency gain justifies the overhead of computing the set of relevant instances, since, in general, the number of relevant instances will be much lesser than the total number of instances. Moreover, the number of entity types added to the CS is limited since structural event types do not depend on the number of constraints in the CS and for each constraint at most two derived types are defined.

Our method uses only the standard elements of any conceptual schema (entity types, relationship types, derived elements and integrity constraints) to transform the original constraints into efficient ones. Therefore, the results of our method can benefit any implementation of the CS, regardless the technology used. In fact, any code-generation strategy able to generate code from a CS could be enhanced with our method to automatically generate efficient constraints, with only minor adaptations.

The expressive power of the OCL language permits to write the same semantic constraint in a variety of syntactic forms. Since our method relies on the syntactic definition of the OCL expressions, choosing the simplest representation of a constraint can entail a more efficient verification of the constraint. As further work, we plan to study how to automatically transform a constraint definition into its simplest representation to guarantee the best results when applying our method.

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. Cabot, J., Teniente, E.: Determining the Structural Events that May Violate an Integrity Constraint. In: Proc. of the 7th UML Conference (UML'04), LNCS, 3273, pp. 320-334
2. Ceri, S., Widom, J.: Deriving Production Rules for Constraint Maintenance. In: Proc. 16th VLDB Conference (VLDB'90), Morgan Kaufmann, pp. 566-577
3. Gogolla, M., Richters, M.: Expressing UML Class Diagrams Properties with OCL. In: A. Clark and J. Warmer, (eds.): Object Modeling with the OCL. Springer, 2002, pp. 85-114
4. Gupta, A., Mumick, I. S.: Maintenance of materialized views: problems, techniques, and applications. In: Materialized Views Techniques, Implementations, and Applications. The MIT Press, 1999, pp. 145-157
5. ISO/TC97/SC5/WG3: Concepts and Terminology for the Conceptual Schema and Information Base. ISO, 1982
6. Olivé, A.: Time and Change in Conceptual Modeling of Information Systems. In: S. Brinkkemper, E. Lindencrona, and A. Solvberg, (eds.): Information Systems Engineering. State of the Art and Research Themes. Springer, 2000, pp. 289-304
7. Olivé, A.: Derivation Rules in Object-Oriented Conceptual Modeling Languages. In: Proc. 15th Int. Conf. on Advanced Information Systems Engineering (CAiSE'03), LNCS, 2681, pp. 404-420
8. OMG: UML 2.0 OCL Specification. OMG Adopted Specification (ptc/03-10-14) ,2003
9. OMG: UML 2.0 Superstructure Specification. OMG Adopted Specification (ptc/03-08-02), 2003
10. Wieringa, R.: A survey of structured and object-oriented software specification methods and techniques. ACM Computing Surveys 30, 1998, pp. 459-527

Appendix A

After applying our method over the constraints defined in the conceptual schema of Fig. 1.1 we obtain the following results:

1. PSEs for each constraint:
    a. *MaxJuniors*: InsertET(Deparment), UpdateAttribute(MaxJuniors, Department), UpdateAttribute(Age,Employee), insertRT(WorksIn).
    b. *NotBossFreelance*: SpecializeET(Freelance), InsertRT(Manages).
    c. *ValidAssignment*: InsertET(Freelance), UpdateAttribute(Assignment).
    d. *NumberEmployees*: DeleteET(Employee), InsertRT(WorksIn)
    e. *MaxFreelances*: InsertET(Freelance), SpecializeET(Freelance)

2. Determining the incrementallity of each constraint:
    MaxJuniors, *NotBossFreelance* and *ValidAssignment* are instance constraints. *NumberEmployees* is a partial instance constraint.
    MaxFreelances is a class constraint.

3. Creating the structural event types:
    For each PSE of an instance or partial instance constraint we create the corresponding structural event type. We show them in Fig. 1, although in general they would be hidden from the designer's point of view.

4. Redefining instance constraints:
    To transform *NotBossFreelance* and *ValidAssignment*, we redefine the constraints over the new derived subtypes, *DepartmentNotBossFreelance* and *FreelanceValidAssignment*. They are shown in Fig. 2 along with their derivation rules. The transformation of *MaxJuniors* has been already explained in section 4.

5. Redefining partial instance constraints:
    We transform *NumberEmployees* by creating two new derived subtypes of *Department* (Fig. 3): *DepartmentNumberEmployees* for the instance PSEs and *DepartmentNumberEmployees'* for the class PSEs. Note that their derivation rules do not allow that both have instances at the same time.

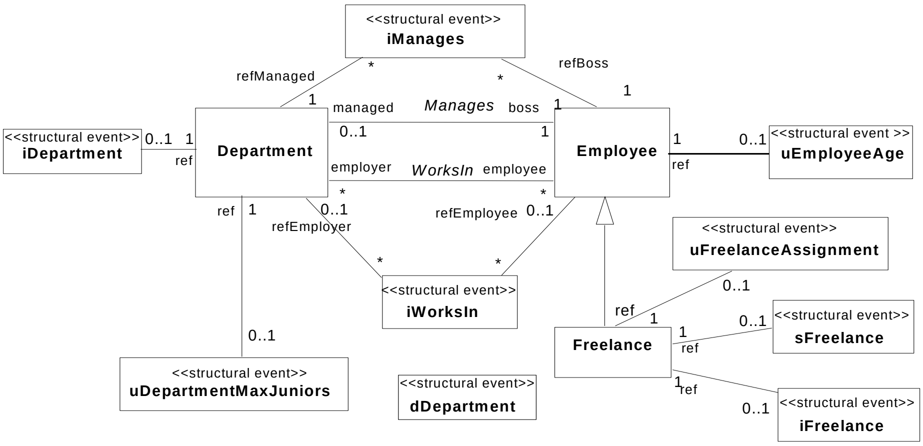
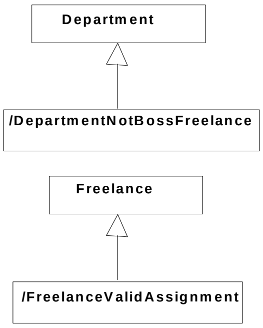


Fig. 1. Structural event types for the example CS



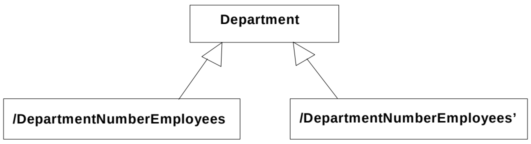
```
-- The redefined constraints
context DepartmentNotBossFreelance inv NotBossFreelance:
  not self.boss.oclIsTypeOf(Freelance)

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

-- The derivation rules
context DepartmentNotBossFreelance::allInstances() : Set(Department)
body: iManages.allInstances().refManaged->union(
        sFreelance.allInstances().ref.managed))->asSet()

context FreelanceValidAssignment::allInstances() : Set(Freelance)
body: iFreelance.allInstances().ref->union(
        uFreelanceAssignment.allInstances().ref)->asSet()
```

Fig. 2. Redefinition of the instance constraints



```
-- The redefined constraints
context DepartmentNumberEmployees inv NumberEmployees:
  self.employee->size()<= Employee.allInstances()->size() / 2

context DeparmentNumberEmployees’ inv NumberEmployees’:
  self.employee->size()<= Employee.allInstances()->size() / 2
```

```
-- The derivation rules
context DepartmentNumberEmployees::allInstances():Set(Department)
body: if DepartmentNumberEmployees’.allInstances->size() = 0 then
         iWorksIn.allInstances().refEmployer.asSet()

context DepartmentNumberEmployees’::allInstances():Set(Department)
body: if dEmployee.allInstances->size()>0 then
         Department.allInstances()
```

Fig. 3. Redefinition of the partial instance constraint