Incremental Evaluation of OCL Constraints

Jordi Cabot[1] 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 determining whether an operation execution violates a given integrity constraint. To perform this computation efficiently, several incremental methods have been developed. The main goal of these methods is to consider as few of the entities in an information base as possible, which is generally achieved by reasoning from the structural events that define the effect of the operations. In this paper, we propose a new method for dealing with the incremental evaluation of the OCL integrity constraints specified in UML conceptual schemas. Since our method works at a conceptual level, its results are useful in efficiently evaluating constraints regardless of the technology platform in which the conceptual schema is to be implemented.

1. Introduction

Integrity constraints (ICs) play a fundamental role in defining the conceptual schemas (CSs) of information systems (ISs) [8]. An IC defines a condition that must be satisfied in every state of an information base (IB). The state of an IB changes when the operations provided by the IS are executed. The effect of an operation on an IB may be specified by means of structural events [18]. A structural event is an elementary change in the population of an entity or relationship type, such as insert entity, delete entity, update attribute, insert relationship, etc.

The IS must guarantee that the IB state resulting from the execution of an operation is consistent with the ICs defined in the CS. This is achieved by ensuring that the structural events that define the operation's effect do not violate any ICs. This process, which is known as *integrity checking*, should be performed as efficiently as possible.

Efficiency is usually achieved by means of *incremental integrity checking*, i.e. by exploiting the information that is available on the structural events to avoid having to completely recalculate the ICs. Hence, the main goal of these methods is to consider as few of the entities in the IB as possible during the computation of IC violations.

For example, a *ValidShipDate* constraint in the CS in Fig. 1.1, which states that "all sales must be completely delivered no later than 30 days after the payment date," may be violated by the execution of the *AddSaleToShipment(s:Sale,sh:Shipment)* operation, which creates a new relationship between sale *s* and shipment *sh*, since *sh* may be planned for a date beyond the last acceptable date for *s*.

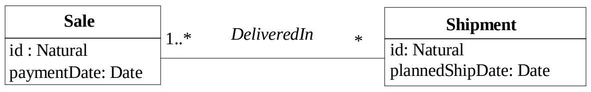


Fig. 1.1. A conceptual schema for sales and their shipments

To verify that *ValidShipDate* is not violated after the execution of the previous operation it is sufficient to consider sale *s* and shipment *sh* of the new relationship, as incremental methods do, rather than carrying out a naive evaluation which must check the previous constraint for all sales and shipments.

In this paper, we propose a new method for coping with the incremental evaluation of ICs at the conceptual level. We assume that CSs are specified in UML [12] and that ICs are defined as invariants written in OCL [11]. For each IC *ic* in the CS and for each structural event *ev* that may violate it, our method provides the most incremental expression that can be used instead of the original IC to check that the application of *ev* does not violate *ic*. By most incremental we mean the one that considers the smallest number of entities of the IB. Our method ensures the most incremental evaluation of the ICs regardless of their specific syntactic definition in the original CS.

If our method were applied to the previous example, it would return an expression whose computation would only verify that the value of the attribute *plannedShipDate* of *sh* does not exceed the value of the attribute *paymentDate* of *s* by more than 30 days.

Since our method works at the conceptual level, it is not technology-dependent. Therefore, the most incremental expressions obtained by our method can be used to efficiently verify the ICs regardless of the target technology platform chosen to implement the CS. Therefore, our results may be integrated into any code-generation method or any MDA-compliant tool to automatically generate an efficient evaluation.

To the best of our knowledge, ours is the first incremental method for OCL constraints. Several proposals have been made for an efficient evaluation of OCL constraints, but with limited results. Moreover, our method is no less efficient than previous methods for the incremental computation of integrity constraints in deductive or relational databases. A comparison with related research is provided in this paper.

The research described herein extends our previous research in [2], in which we proposed a method for computing the entities that might violate an integrity constraint; this method provides partial efficiency results in the evaluation of ICs. The main limitation of that research was that the results were totally dependent on the particular syntactic definition of the IC chosen by the designer, which involved, in the worst case, an almost complete recomputation of the IC after certain structural events. For instance, with the previous definition of *ValidShipDate*, after the *AddSaleToShipment* operation, [2] would verify that the planned date of all shipments of *s* is correct with regards to the payment date (instead of considering just *sh* and *s*, which is achieved using the method we propose here).

The paper is organized as follows. In the subsequent section we present several basic concepts. Section 3 describes our method for incremental integrity checking. Section 4 introduces an optimization for dealing with sets of structural events. An example of the method's application is shown in Section 5. Section 6 compares our approach to related research. Finally, Section 7 presents the conclusions and points out further work.

2. Basic concepts

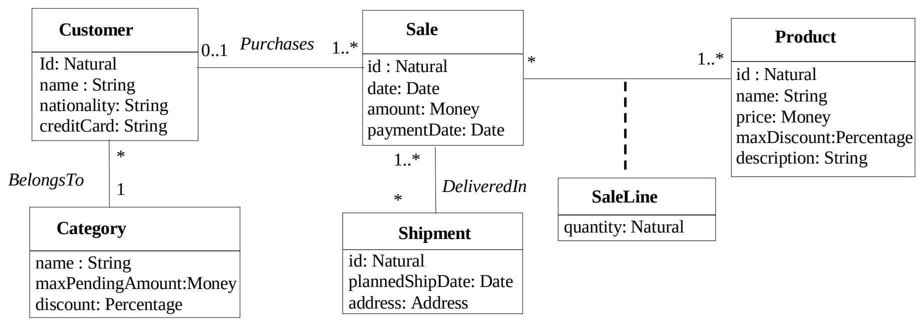
Our method assumes that CSs are specified in UML [12]. In UML, entity types and relationship types are represented as classes and associations respectively, while entities are called objects and relationships are referred to as links.

Additionally, the method assumes that textual ICs are defined as invariants written in OCL [11]. Graphical constraints supported by UML, such as cardinality or disjointness constraints, can be transformed into a textual OCL representation, as shown in [6]; therefore, they can also be handled by our method.

As an example, consider the CS in Fig. 2.1, which was designed to (partially) model a simple e-commerce application. The CS contains information on the sales and the products they contain. Sales can be delivered split up into several shipments and shipments can be reused to ship several sales. Finally, sales may be associated with registered customers who benefit from discounts depending on their category.

The CS includes three textual ICs. The first IC (*CorrectProduct*) verifies that all products have a price greater than zero and a max discount of 60% (the maximum discount permitted by the company). The second one is the previous *ValidShipDate* IC, stating that sales must be completely shipped within 30 days after the payment date (and that therefore all shipments of that sale must be planned before that date). Finally, *NotTooPendingSales* holds if customers do not have pending sales for an amount greater than the *maxPendingAmount* value in their category.

Note that an IC in OCL is defined in the context of a specific type[1] or *context type*, and its *body* (the Boolean OCL expression that states the IC condition) must be satisfied by all instances of that type. For example, in *ValidShipDate*, *Sale* is the context type, the variable *self* refers to an entity of *Sale* and the date condition (the body) must hold for all possible values of *self* (i.e. all entities of *Sale*).



context Product **inv** CorrectProduct: self.price>0 and self.maxDiscount<=60

context Sale **inv** ValidShipDate: self.shipment->forAll(s| s.plannedShipDate<=self.paymentDate+30)

context Category **inv** NotTooPendingSales:

self.customer->forAll(c| c.sale->select(paymentDate>now()).import->sum()<=self.maxPendingAmount)

Fig. 2.1. Our running example

As we mentioned above, ICs must be checked after structural events have been applied. In this paper, we consider the following kinds of structural event types:

[1] In UML 2.0, the context type may be either an entity type or a relationship type since both types are represented in the UML metamodel as subclasses of the *Classifier* metaclass.

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

3. Determining the incremental expressions of an OCL constraint

In this section, we describe the method we propose for obtaining the most incremental expressions that should be used instead of the original IC, to ensure that the IC is not violated when a structural event is applied to the IB. We start by providing an overview of the method in Section 3.1. Then, in Sections 3.2 to 3.4, we define the three main operators used in our method to obtain these incremental expressions. An implementation of the method is described in [4].

3.1 An overview of the method

A direct evaluation of the original OCL definition of an IC, i.e. the one specified in the CS, may be highly inefficient. For example, a direct evaluation of the constraint *ValidShipDate* (as stated in Fig. 2.1) after an event *InsertRT(DeliveredIn)*, which creates a new relationship *d* between sale *s* and shipment *sh*, would require taking into account all sales (because this is the context type) and, for each sale, all its shipments (because of the *forAll* operator), leading to a total cost proportional to P_sxN_{sh}, where P_s is the population of the *Sale* type and N_{sh} is the average number of shipments per sale. However, if we take the structural event into account we may conclude that the following expression:

$$exp \equiv d.shipment.plannedShipDate<=d.sale.paymentDate+30$$

suffices to verify *ValidShipDate* (since the IB satisfies *exp* iff *ValidShipDate* also holds). Evaluating *exp* only requires that two entities be taken into account: the shipment participating in *d* (*d.shipment*) and its sale (*d. sale*). Clearly, evaluating this expression is much more efficient than directly evaluating the original IC.

The main goal of our method is to translate an OCL constraint *ic* into the set of most incremental OCL expressions that allow an efficient evaluation of *ic* every time a structural event is applied over the IB. In general, there will be a different most incremental expression for each IC and each structural event that may violate it.

By incremental we mean that the evaluation of the expression does not need to take all entities of the context type of *ic* and all their relationships into account, since it can reason forward directly from the entities that have been updated by the structural event. The most incremental expression is the one that considers the lowest number of entities of the IB. Obviously, the more entities required to evaluate an expression the less efficient is its computation. We use $inc_{<ic,ev>}$ to denote the most incremental expression for a constraint *ic* after a structural event *ev* has been applied. In the previous example, *exp* is the most incremental expression for *ValidShipDate* after the event *InsertRT(DeliveredIn)* has been applied.

The events that may violate an IC are called *potentially violating structural events* (PSEs) for that IC and may be determined by the method proposed in [1]. Applied to our example, this method would state that only *InsertRT(DeliveredIn), UpdateAttibute(plannedShipDate, Shipment)* and *UpdateAttribute(paymentDate, Sale)* can violate *ValidShipDate*. Note that other events such as *DeleteET(Sale)* or *UpdateAttribute(address,Shipment)* may never violate that IC. The most incremental expressions of an IC must only be defined by events in the set of PSEs of the IC.

Determining the most incremental expressions depends on the given PSE and on the structure of the IC. Moreover, it generally requires changing the context type of the initial IC, since we cannot guarantee that the context chosen by the designer to specify the IC is the most appropriate one as far as efficiency is concerned.

Intuitively, our method works as follows. First, it selects from all possible context types for the constraint (those types referenced in the body of the IC) the most appropriate one with respect to the structural event (i.e. the one that will produce the most efficient expression at the end of the process for that event). Second, it redefines the body of the IC in terms of this new context type *ct'*. Third, it computes the instances of *ct'* that may have been affected by the event. Finally, the incremental expression is obtained by refining the body of the IC to be applied only over those relevant instances. This procedure is specified in the following algorithm.

Algorithm: *Obtaining the most incremental expressions*

Given an IC *ic,* which is defined in terms of a context type *ct* and an event *ev* (where *ev* is a PSE for *ic*), the following *IncrementalExpression* algorithm returns the $inc_{<ic,ev>}$ expression:

```
IncrementalExpression( ic: Constraint, ev: Event) : Expression
        Type bestContext := BestContext(ic,ev)
        Constraint ic':= Translate(ic,ev,bestContext);
        Expression rel := Relevant (ic', ev)
        return (Merge(rel, ic'))
```

where

1. *BestContext(ic:Constraint, ev:,Event)* returns the type that must be used as a context of *ic* to generate an incremental expression for i*c* after event *ev*.
2. *Translate(ic:Constraint, ev:Event, t:Type)* returns an IC *ic'*, which is defined using *t* as a context type, such that *ic'* is equivalent to *ic* regarding *ev*.
3. *Relevant(ic:Constraint, ev:Event)* returns an OCL expression whose evaluation returns the instances of *ct* (the context type of *ic*) affected by *ev*.
4. *Merge(exp:Expression, ic:Constraint)* creates the final $inc_{<ic,ev>}$ expression by applying *b* (the body of *ic)* to all entities reached in *exp* (the expression computing the relevant instances). If the evaluation of *exp* returns a single instance (i.e. all navigations included in *exp* have '1' as a maximum multiplicity), this operator just replaces all occurrences of *self* in *b* with *exp*. Otherwise, the final expression is *exp-> forAll(v|b)* where all occurrences of *self* in *b* are replaced with *v*.

Let us again consider the event *InsertRT(DeliveredIn)* and the constraint *ValidShipDate*. As we have seen their incremental expression is *exp*, which is obtained using our method in the following way:

1. BestContext(*ValidShipDate,InsertRT(DeliveredIn)*) = *DeliveredIn*
2. Translate(*ValidShipDate*, *InsertRT(DeliveredIn)*, *DeliveredIn*) =
   context *DeliveredIn* ***inv*** *newIC:*
   self.shipment.plannedShipDate<=self.sale.paymentDate+30
3. Relevant(*newIC,InsertRT(DeliveredIn)*) = *d*, the new relationship created by the *InserRT* event over *DeliveredIn*
4. Merge(*d,newIC*) (i.e. $\text{inc}_{<\text{ValidShipDate,InsertRT(DeliveredIn)}>}$) =
   d.shipment.plannedShipDate <= d.sale.paymentDate+30

We show in [3] that the expression generated by the previous algorithm is always the most incremental one.

In the rest of this section we formally define the *BestContext*, *Translate* and *Relevant* operators. To facilitate their definition, our method assumes a normalized representation of the ICs. The normalization reduces the number of different OCL operators appearing in their body (for instance, replacing the *implies* operator with a combination of the *not* and *or* operators or the *exists* operator with a combination of the *select* and *size* operators). This representation is automatically obtained from the initial IC and does not entail a loss of expressive power of the ICs we deal with.

All three operators work with the ICs represented as an instance of the OCL metamodel [11]. According to this representation, they can handle the OCL expression by forming the body of the IC as a binary tree, in which each node represents an atomic subset of the OCL expression (an operation, an access to an attribute or an association, etc.) and the root is the most external operation of the OCL expression. As an example, in Fig. 3.1 the constraint *ValidShipDate* is represented by means of the OCL metamodel. Each node is marked with the set of PSEs produced by that node [1] (i.e. the events that are PSEs of the IC because of that particular node).

3.2 BestContext(ic:Constraint, ev:,Event)

The best context to verify an IC *ic* after applying an event *ev* to the IB is automatically drawn from the node where *ev* is assigned in the tree representing IC. We use $node_{ev}$ to denote this node (when different $node_{ev}$ exist we repeat the process for each node). The *BestContext* operator always returns the same result regardless of the original syntactic definition of *ic*, since all possible syntactic definitions of *ic* must contain $node_{ev}$ (because all of them may be violated by *ev*).

To determine the context type, we must consider whether $node_{ev}$ participates (i.e. is included) in an *individual condition* or in a *collection condition*. Intuitively, individual conditions must be verified for each individual entity (for instance, each individual product must satisfy the *CorrectProduct* IC). In contrast, collection conditions must be verified by the set of entities affected by the condition as a whole (for instance, in *NotTooPendingSales*, the sum of all sales of a customer must satisfy the *maxPendingAmount* condition). Individual and collection conditions are formalized in Definitions 3.2.1 and 3.2.2.

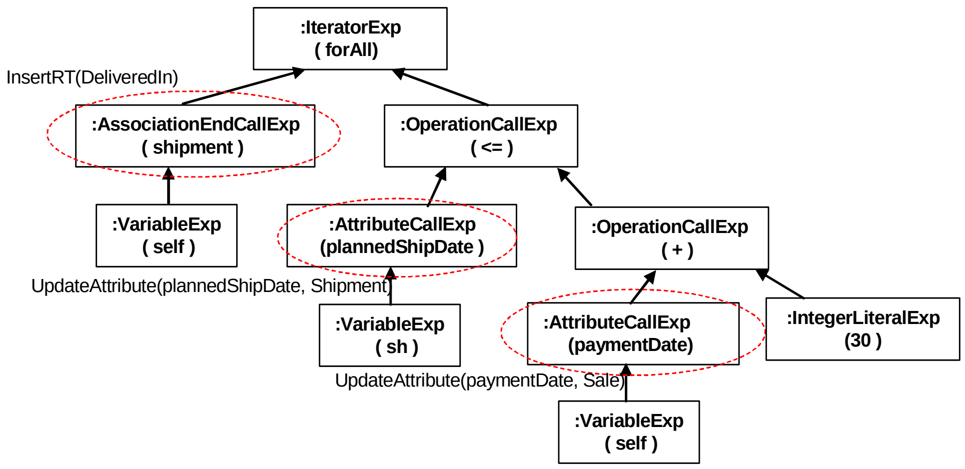


Fig. 3.1. The OCL metamodel of *ValidShipDate* and its set of PSEs

Definition 3.2.1 A node *n* participates in a *collection condition* when *n* is used to compute an aggregate operator. Formally, when *n* verifies that {∃n'| n'∈ *PathRoot(n)* and n'.oclIsTypeOf(OperationCallExp) and n'.referredOperation ∈ {*size*, *sum*, *count*}}, where *PathRoot(n)* is defined as the ordered sequence of nodes encountered between *n* (the first node in the sequence) and the root of the tree (the last one). *OclIsTypeOf* and *referredOperation* are elements defined in the OCL metamodel.

Definition 3.2.2. A node *n* participates in an *individual condition* if it does not participate in a collection condition.

Clearly, since individual conditions must hold for each individual entity restricted by the constraint, the most incremental expression will be the one that only takes into account the single entity updated by the event. The original IC must then be redefined in terms of the type of entity to obtain this expression.

Proposition 3.2.3 Let *ev* be an event over an entity *e* (resp. relationship *r*) of type *E* (resp. *R*). If $node_{ev}$ is included in an individual condition, *BestContext* returns the same type *E* (resp. *R*) as the best context.

In our example, the constraint *ValidShipDate* may be violated by three different structural events, all of them included in individual conditions: *InsertRT(DeliveredIn)*, *UpdateAttribute(plannedShipDate,Shipment)* and *UpdateAttribute(paymentDate,Sale)*, as shown in the tree in Fig. 3.1. Their best contexts are therefore *DeliveredIn*, *Shipment* and *Sale* respectively.

The same idea cannot be applied to events included in collection conditions since those conditions must be satisfied by the collection as a whole and not by each single instance. Thus, to consider the modified entity or relationship is not enough to verify it because, after every modification, the whole collection must be recomputed again and the other entities in the collection must also be taken into account. For this reason, in selecting the best context it must be ensured that, after each modification, only the exact set of entities involved in the condition is checked.

For instance, an *InsertRT(Purchases)* event (i.e. the assignment of a sale *s* to a customer *c*) may violate the constraint *NotTooPendingSales*. In this case, the *maxPendingAmount* condition must be satisfied by the set of sales of each customer; thus, after assigning a sale to a customer *c*, it is enough to verify the set of sales of *c*. In this way, the *Customer* type is the origin of the collection condition. Note that *Category* is not the origin since it is not the union of sales of all customers in a category who must satisfy the condition.

Therefore, if the event *ev* in the call to the *BestContext* operator is included in a collection condition, the type defined as the origin of the collection is the best context. This will be especially true when dealing with sets of events (see Section 4).

Definition 3.2.4 Given a node n, *PathVar(n)* is defined as the ordered sequence of nodes encountered between *n* (the first node) and the node representing the *self* variable (the last node) of the subtree to which *n* belongs. More precisely, *PathVar(n)* is computed as follows:

- The first node is *n*.
- For each node included in *PathVar* we also include its child (or the left child if the node has two children), if any.
- When a node *n* included in *PathVar* represents a variable other than *self* (i.e. variables used in *select* or *forAll* iterators), we add as a left child the node pointed to in *n.referredVariable.loopExpr* (i.e. the node representing the iterator; *referredVariable* and *loopExpr* are associations defined in the OCL metamodel).

Definition 3.2.5 Given an integrity constraint *ic*, an event *ev* and the sequence *PathVar(node$_{ev}$)*, *node$_{or}$*, the node origin of a collection condition is

- The left child of a node *n*$\in$ *PathVar(node$_{ev}$)*, representing a *forAll* iterator, when a *select* iterator is not encountered between the *self* variable and *n*.
- Otherwise, the last node in *PathVar(node$_{ev}$)* (i.e. the node representing the *self* variable). If following the *self* variable there is a set of nodes representing navigations $r_1...r_n$ where all r_i have a maximum multiplicity of 1, then the final *node$_{or}$* is the node at r_n.

Proposition 3.2.6 Let *ev* be an event included in a collection condition. The type of the entities at *node$_{or}$* is then returned as the best context.

In the *NotTooPendingSales* constraint, *Customer* is the origin of the condition since it is the type of the entities accessed in the node previous to the *forAll* iterator (the node navigating to customers from category). Therefore, *Customer* is the *BestContext* for all events included in the collection condition (updates of the *paymentDate* and *amount* attributes and inserts of *Purchases* relationships). The PSEs *UpdateAttribute(maxPendingAmount)* and *InsertRT(BelongsTo)* are included in individual conditions; thus, their best contexts are *Category* and *BelongsTo* respectively (as determined by Proposition 3.2.3).

3.3 Translate(ic:Constraint, ev:Event, t:Type)

Given an IC *ic* that has a context type *t* and an event *ev*, the *Translate* operator returns an IC *ic'* defined over a type *t'*, *t'*$\neq$ *t*, which is semantically equivalent to *ic* with respect to event *ev*. Having applied *ev* over the IB, *ic'* and *ic* are semantically equivalent when *ic'* is satisified iff *ic* is also satisfied in the new state of the IB.

The *Translate* operator extends the method we presented in [5] since the context changes required in the work reported here present two particularities that can be used in order to provide a more optimized redefinition than the one in the previous reference.

First, *t'* is the type returned by the *BestContext* operator (this implies, for instance, that *t'* is referenced in the body of *ic*). Second, *ic* and *ic'* need only be equivalent with regards to the particular event *ev*. Therefore, *ic'* need not worry about all the literals of *ic* that cannot be violated by *ev*.

For instance, given that the body of *ic* follows the pattern L_1 *and* L_2 (as *CorrectProduct* in Fig. 2.1) and that *ev* can only induce a change in the truth value of L_1, *ic'* does not need to include the verification of L_2. L_2 was true before *ev* was executed (since all the states of the IB must be consistent) and, since *ev* does not affect it, L_2 will still hold after its execution. When it does not hold it is because some other event, *ev'*, has been applied. The incremental expression for *ev'* will take care of this possible violation.

Translate is defined in two separate steps. First, the tree is pruned to remove the irrelevant conditions. Then the remaining tree is redefined over the context type *t'* to obtain the final body of the translated constraint *ic'*.

Definition 3.3.1 Let *ev* be an event attached to a node $node_{ev}$. A node n_{and} representing an *AND* condition may be pruned if $\{n_{and} \in PathRoot(node_{ev})$ *and* $\neg\exists n'|$ $n' \in PathRoot(n_{and})$ *and n'.oclIsTypeOf(IteratorExp) and n'.name="select"}*. N_{and} nodes are replaced with the child node $n_{child} \in PathRoot(node_{ev})$. Consequently, the other child of n_{and} (i.e. the other condition) is removed from the tree.

Definition 3.3.2 Given a pruned tree *tr* that represents a constraint *ic* defined using a context type *t*, an event *ev* and the new context type *t'*, the redefined tree *tr'* that represents an equivalent IC *ic'* defined over *t'* is obtained according to the following steps (see [5] for a more detailed explanation and examples):

- Determining the node $node_{t'}$. $Node_t$ is the node $\in$ $PathVar(node_{ev})$ whose evaluation returns entities of type *t'*. If *t'* is a relationship type, $node_{t'}$ is the node previous to the navigation through a role of *t'*.
- Replacing all subtrees that match the sequence $seq=PathVar(node_{t'})$ with a single node representing the *self* variable.
- Replacing all other nodes that represent *self* variables with the subtree corresponding to the expression $self.r_1 \ldots r_n$ (or $self.r_1 \ldots r_n$*->forAll(v|)* when the maximum multiplicity of some r_i is greater than 1), where $r_1..r_n$ are the roles needed to navigate from *t'* to *t* (the roles opposite to the ones used in the *ic* to navigate from *t* to *t'*). Formally, $r_1...r_n$ = $Inverse(PathVar(node_t))$ with *Inverse* defined as $\{\forall n \in PathVar(node_t) \mid n.oclIsTypeOf(AssociationEndCallExp) \rightarrow OppositeRole(n) \in Inverse(PathVar(node_t))\}$.
- Adding the subtree that corresponds to the expression $self.r_1 \ldots r_n$*->notEmpty() implies X* (where *X* is the tree resulting from the previous steps) to ensure that only those instances of *t'* related to a given instance of *t* are verified (otherwise, they were not involved in the original IC).

The resulting tree can be simplified [5] by, for instance, replacing the subtree *self.r₁...rₙ->notEmpty()* with *true* if all multiplicities of $r_1...r_n$ are at least 1 or by removing the *forAll* iterators over single entities.

For example, *Translate(NotTooPendingSales,UpdateAttr(paymentDate,Sale), Customer*) transforms the constraint *NotTooPendingSales*, as defined in Fig. 2.1, in the following *NotTooPendingSales'* constraint defined with the context *Customer*:

context *Customer* ***inv:*** *self.sale->select(paymentDate>now()).amount>sum()*
<=self.category.maxPendingAmount

where, after step one, *self.customer* has been replaced with *self*, the other *self* variable has been replaced with *self.category* (*category* is the role required to navigate from customer to category) and finally, *self.category->notEmpty()* has been simplified (all customers belong to a category) and the *forAll* has been removed.

3.4 Relevant(ic:Constraint, ev:Event)

After issuing a PSE *ev* for an IC *ic* whose context type is *t*, only the instances of *t* that may have been affected as a result of applying *ev* should be verified. The goal of the *Relevant* operator is to return an expression *exp* that returns this set of relevant instances when it is evaluated; *exp* can be automatically derived from the tree representing *ic* [2].

Intuitively, the relevant instances of *t* are the ones related to the instance modified by *ev*. Therefore, the basic idea is that *exp* will consist of the sequence of navigations required to navigate back from the modified instance to the instances of *t*. As in the previous operator, the navigations required are obtained by reversing the navigations used to navigate from the *self* variable to $node_{ev}$.

Definition 3.4.1 Let *ic* be an IC and *ev* a PSE that appears in nodes $node_{ev1}…node_{evn}$. Then, $Relevant(ic,ev) = Inverse(PathVar(node_{ev1})) \cup … \cup Inverse(PathVar(node_{evn}))$.

As an example, let us consider the *NotTooPendingSales'* IC (as redefined in the previous section). After the event *UpdateAttribute(amount, Sale)* that updates a sale *s*, the IC must be verified over customers returned by *Relevant(NotTooPendingSales', UpdateAttribute(amount, Sale))*. In this case, the operator returns the expression *s.customer*, which implies that we just need to verify the customer that the *sale* is assigned to (at most one, because of the maximum multiplicity specified in *Purchases*). In the expression, *customer* represents the opposite role of the *sale* role of the *Purchases* relationship type (the single role appearing in the *PathVar* sequence of nodes for the $node_{ev}$ of the update event).

4. Dealing with sets of events

Up to now we have provided a method that generates incremental expressions for the efficient verification of an IC after issuing a PSE *ev*. Obviously, if an operation consists of several PSEs for the IC, the consistency of the new state of the IB can be verified using the incremental expressions corresponding to each individual event. However, the efficiency can be improved by taking into account the relationship between the different events when computing the affected instances. This improvement is only relevant to events included in collection conditions (events in individual conditions must be individually verified by each entity).

By way of example, let us assume that the execution of an operation updates the amount of two sales (s_1 and s_2) and assigns a sale s_3 to a customer *c*. If one (or both) of the updated sales were also assigned to *c*, we must verify the *NotTooPendingSales* constraint over *c* several times (once because of the sale assignment and the other times because of the update of sales of *c*). However, if we first merge the customers affected by each single event and then verify them, we avoid having to verify the same customer several times.

Proposition 4.2 Let $set=\{ev_1,ev_2,\dots ev_n\}$ be a set of different events for an IC *ic* sharing the same IC definition *ic'* after the the *BestContext* and *Translate* operators, and included in the same operation (without loss of generality, we assume that each operation constitutes a single transaction). The *Relevant* operator is then redefined as

$$Relevant(ic',set) := Relevant(ic',ev_1) \cup \dots \cup Relevant(ic', ev_{ni})$$

Following the previous example, now the relevant customers (i.e. the ones that will be verified) are computed with the expression

c.union->(s_1.customer->union(s_2.customer))

Thus, each relevant customer will be verified only once.

5. Applying the method

We have applied our method to obtaining the most incremental expressions of all ICs in the CS in Fig. 2.1. The results are shown in Table 5.1. The first column indicates the IC. The second one specifies the structural events[2] that may violate each IC. Finally, the third column shows the most incremental expressions obtained for each IC due to each of the events. In this column, the initial variable represents the entity or relationship modified by the event (*d* represents the created *DeliveredIn* relationship, *sh* the updated *Shipment* and so forth).

For instance, Table 5.1 allows us to detect that the application of an event *UpdateAttribute(paymentDate,Sale)* over a sale *s* may violate the ICs: *ValidShipDate* and *NotTooPendingSales*. The most incremental expressions that allow us to verify that the new state of the IB does not violate any ICs are given by expressions 3 and 7.

As we said, using the most incremental expressions to verify the ICs in the original CS ensures the optimal efficiency of the integrity checking process as far the number of entities involved during the computation is concerned. To illustrate the importance of those results, Table 5.2 compares the cost of the most incremental expressions for *ValidShipDate* (as given by Table 5.1) with the cost of directly evaluating the original IC (see Fig. 2.1).

[2] To simplify, we use the notation *UpdateAttr(attr)* when the type is clear from the context.

Table 5.1 Results of applying our method over the example CS

IC	Event	Incremental expression
Valid Ship Date	InsertRT(DeliveredIn)	1. d.shipment.plannedShipDate<=d.sale.paymentDate+30
	UpdateAttr(plannedShip Date)	2. sh.sale->forAll(s| sh.plannedShipDate <= s.paymentDate+30)
	UpdateAttr(paymentDate)	3. s.shipment->forAll(sh| sh.plannedShipDate <= s.paymentDate+30)
NotToo Pend Sales	UpdateAttr(maxPending Amount)	4. c.customer->forAll(cu| cu.sale->select(paymentDate> now()).amount->sum()<=c.maxPendingAmount
	InsertRT(BelongsTo)	5. b.customer.sale ->select(paymentDate>now()).amount->sum()<=b.category.maxPendingAmount
	InsertRT(Purchases)	6. pur.customer.sale ->select(paymentDate>now()).amount->sum()<=pur.customer.category.maxPendingAmount
	UpdateAttr(paymentDate)	7. s.customer.sale->select(paymentDate>now()).amount->sum()<=s.customer.category.maxPendingAmount
	UpdateAttr(amount)	
Correct Prod	UpdateAttr(price)	8. p.price>0
	UpdateAttr(maxDiscount)	9. p.maxDiscount<=60
	InsertET(Product)	10. p.price>0 and p.maxDiscount<=60

In Table 5.2, P_s stands for the number of instances of *Sale*, N_{sh} for the average number of shipments per sale and N_s for the average number of sales per shipment. Cost comparisons for the evaluation of the other ICs are given in [3].

Table 5.2 Cost comparisons for *ValidShipDate*

Event	Cost(ValidShipDate)	Cost (Incremental Expression)
InsertRT(DeliveredIn)	P_s x N_{sh}	2
UpdateAttribute(paymentDate)	P_s x N_{sh}	1+1xN_{sh}
UpdateAttribute(plannedShipDate)	P_s x N_{sh}	1+1xN_s
Other events	P_s x N_{sh}	0

Designers may use the most incremental expressions to efficiently verify the ICs when they are implementing the CS in any final technology platform. For instance, during code generation for an object-oriented technology, adding expressions 3 and 7 to methods that include the *UpdateAttribute(paymentDate, Sale)* event is enough to ensure that the IB is not violated after the application of the event. Additionally, when we are using a relational database as an IB, we may create a set of triggers that verify both expressions before we apply the change to the *Sale* table data. For example, Fig. 5.1 shows a possible verification of expression 3 in both technologies.

6. Related work

Two kinds of related research are relevant here: methods devoted to the problem of integrity checking, of which there is a long tradition, especially in the database field (see Section 6.1), and tools that provide code-generation capabilities that may include facilities for improving the efficiency of integrity checking (see Section 6.2).

```
MethodX(Sale s,…)
{  . . .  s.paymentDate = value;    …
     //Verification of expression 3
    Iterator setsh = s.shipments.iterator();
    while ( setsh.hasNext() )
    {   Shipment sh = (Shipment) setsh.next();
        If  (sh.plannedShipDate>s.paymentDate+30)
            throw new Exception(“Invalid date”);
     }
}
```

```
create trigger uPaymentDate
before update of PaymentDate on Sale for each row
Declare v_Error NUMBER;
           EInvalidDate Exception;
Begin   --Verification of expression 3
    SELECT count(*) into v_Error
    FROM DeliveredIn d, Shipment sh
    WHERE d.sale = :new.id and d.shipment = sh.id
              and sh.plannedShipDate>:new.paymentDate+30;
     If (v_Error>0) then raise EInvalidDate; end if;
End;
```

Fig. 5.1. Examples of incremental expressions implemented in particular technologies

6.1 Integrity checking methods for deductive or relational databases

The most important results of related research of an incremental checking of integrity constraints are provided by methods proposed for integrity checking in deductive databases. In what follows we briefly show that the efficiency of our incremental expressions is equivalent to the incremental rules generated by the most representative proposals in this field (see [7] for a survey).

They define ICs as inconsistency predicates that will be true whenever the corresponding IC is violated. For example, they would represent *ValidShipDate* as (where *S* stands for *Sale*, *Sh* for *Shipment*, *D* for DeliveredIn, *pd* for *paymentDate* and *psh* for *plannedShipDate*)

$$Ic_{ValidShipDate} \leftarrow S(s,pd) \wedge D(s,sh) \wedge Sh(sh, psd) \wedge pd+30>psd$$

To incrementally check this constraint they would consider the following rules:

1. $Ic_{ValidShipDate} \leftarrow iS(s,pd) \wedge D(s,sh) \wedge Sh(sh, psd) \wedge pd+30>psd$
2. $Ic_{ValidShipDate} \leftarrow uS(s,pd') \wedge D(s,sh) \wedge Sh(sh, psd) \wedge pd'+30>psd$
3. $Ic_{ValidShipDate} \leftarrow S(s,pd) \wedge iD(s,sh) \wedge Sh(sh, psd) \wedge pd+30>psd$
4. $Ic_{ValidShipDate} \leftarrow S(s,pd) \wedge D(s,sh) \wedge iSh(sh, psd) \wedge pd+30>psd$
5. $Ic_{ValidShipDate} \leftarrow S(s,pd) \wedge D(s,sh) \wedge uSh(sh, psd') \wedge pd+30>psd'$

where *iX(y)* means that the entity *y* of type *X* has been inserted and *uX* means that it has been updated (those updates are only considered explicitly in [17]).

After applying our method to the same constraint, we obtain the following three incremental expressions (as shown in Table 5.1):

a. s.shipment->forAll(sh|s.paymentDate+30> sh.plannedShipDate)
b. sh.sale->forAll(s| s.paymentDate+30>sh.plannedShipDate)
c. d.sale.paymentDate+30>=d.shipment.plannedShipDate

where *s* is the updated sale, *sh* the updated shipment and *d* the new *DeliveredIn* relationship. The definitions we get are respectively equivalent to Rules 2, 5 and 3 in those methods. Note that the insertion of a shipment (Rule 4) cannot violate the constraint if it is not assigned to a sale, which is already controlled by our expression *c* (similarly, for the insertion of sales, Rule 1).

6.2 Tools with code-generation capabilities

Almost all current CASE tools offer code-generation capabilities. However, most of them do not allow the definition of OCL constraints or (more commonly) do no take them into account when they generate the code. This is the case of tools such as *Rational Rose*, *MagicDraw*, *ArcStyler*, *OptimalJ*, *Objecteering/UML* and many more.

All tools that are able to generate code for the verification of OCL constraints depart from the ICs exactly as defined by the designer; thus, their efficiency depends on the concrete syntactic representation of the IC. The differences between these tools lie in how they decide when the IC needs to be checked and the amount of entities they take into account every time the IC is checked.

Tools such as *Octopus* [10] or *OCLE* [16] transform the IC into a Java method; when the method is executed, an exception is raised if the IC does not hold. However, the decision of when to verify the IC is left to the designer. The *OO-Method* [13] verifies all ICs whose context type is *t* whenever a method of *t* is executed (even if the changes produced by the method cannot violate a given IC). *Dresden OCL* [15] verifies the ICs only after events that modify the elements appear in the IC body, but it does not consider whether that sort of change can really induce its violation. For instance, *Dresden OCL* would verify *ValidShipDate* after deletions of *DeliveredIn* relationships, although only the latter event can really violate the IC. *OCL2SQL* (included in [15]) transforms each IC into an SQL view so that the view returning data indicates that the IC does not hold. Nevertheless, the view is not incremental. Every time an entity is modified, the view verifies all the entities of the context type.

7. Conclusions and further work

We have presented a method that generates the most incremental expressions for OCL constraints defined in UML CSs. These expressions can be used instead of the original IC when the IB is verified after modifications caused by a set of structural events. The method has been implemented in [4].

The most incremental expressions use information on the structural events issued during the operation to optimize the integrity checking process by considering as few entities of the IB as possible. In this way, we ensure an optimal verification of the ICs regardless of the concrete syntactic definition originally chosen by the designer.

The main advantage of our approach is that it works at a conceptual level; therefore, it is not technology-dependent. In contrast with previous approaches, our results can be used regardless of the final technology platform selected to implement the CS. In fact, any code-generation strategy able to generate code from a CS, such as the ones presented in the previous section, could be enhanced with our method for the purpose of automatically generating an optimal integrity checking code for the ICs.

As further work, we could try to further improve the efficiency of the whole integrity checking process by considering, at the conceptual level, additional optimization techniques initially proposed for databases like [9] and [14]. Moreover, we would also like to adapt our method for the incremental maintenance of derived elements specified in a CS.

Acknowledgments

We would like to thank the people of the GMC group J. Conesa, D. Costal, X. de Palol, C. Gómez, A. Olivé, A.Queralt, R. Raventós and M. R. Sancho for their many useful comments in the preparation of this paper. This work has been partially supported by the Ministerio de Ciencia y Tecnologia under project TIN2005-06053.

References

1. Cabot, J., Teniente, E.: Determining the Structural Events that May Violate an Integrity Constraint. In: Proc. 7th Int. Conf. on the Unified Modeling Language (UML'04), LNCS, 3273 (2004) 173-187
2. Cabot, J., Teniente, E.: Computing the Relevant Instances that May Violate an OCL Constraint. In: Proc. 17th Int. Conf. on Advanced Information Systems Engineering (CAiSE'05), LNCS, 3520 (2005) 48-62
3. Cabot, J., Teniente, E.: Incremental Evaluation of OCL Constraints (extended version). UPC, LSI Research Report, LSI-05-12-R (2005)
4. Cabot, J., Teniente, E.: A Tool for the Incremental Evaluation of OCL Constraints. Available at www.lsi.upc.edu/~jcabot/research/tools/caise06 (2006)
5. Cabot, J., Teniente, E.: Transforming OCL Constraints: A Context Change Approach. In: Proc. 21st Annual ACM Symposium on Applied Computing (Model Transformation Track), (2006)
6. Gogolla, M., Richters, M.: Expressing UML Class Diagrams Properties with OCL. In: A. Clark and J. Warmer, (eds.): Object Modeling with the OCL. Springer-Verlag (2002) 85-114
7. Gupta, A., Mumick, I. S.: Maintenance of Materialized Views: Problems, Techniques, and Applications. In: Materialized Views Techniques, Implementations, and Applications. The MIT Press (1999) 145-157
8. ISO/TC97/SC5/WG3: Concepts and Terminology for the Conceptual Schema and Information Base. ISO, (1982)
9. Lee, S. Y., Ling, T. W.: Further Improvements on Integrity Constraint Checking for Stratifiable Deductive Databases. In: Proc. 22nd Int. Conf. on Very Large Data Bases. Morgan Kaufmann (1996) 495-505
10. Klasse Objecten.: Octopus: OCL Tool for Precise UML Specifications. (2005)
11. OMG: UML 2.0 OCL Specification. OMG Adopted Specification (ptc/03-10-14)
12. OMG: UML 2.0 Superstructure. OMG Adopted Specification (ptc/03-08-02)
13. Pastor, O., Gómez, J., Insfrán, E., Pelechano, V.: The OO-Method Approach for Information Systems Modeling: From Object-Oriented Conceptual Modeling to Automated Programming. Information Systems 26 (2001) 507-534
14. Ross, K. A., Srivastava, D., Sudarshan, S.: Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time. In: Proc. ACM SIGMOD international conference on Management of data, (1996) 447-458
15. Dresden University.: Dresden OCL Toolkit. (2005)
16. Babes-Bolyai University.: Object Constraint Language Environment 2.0.
17. Urpí, T., Olivé, A.: A Method for Change Computation in Deductive Databases. In: Proc. 18th Int. Conf. on Very Large Data Bases. Morgan Kaufmann (1992) 225-237
18. Wieringa, R.: A Survey of Structured and Object-Oriented Software Specification Methods and Techniques. ACM Computing Surveys 30 (1998) 459-527