Incremental Integrity Checking of UML/OCL Conceptual Schemas

Jordi Cabot a, b *, Ernest Teniente c

aUniversity of Toronto, Toronto, M5S 3G4, Ontario (Canada)
bOpen University of Catalonia, Barcelona, E08018, Catalonia (Spain)
cTechnical University of Catalonia, Barcelona, E08034, Catalonia (Spain)

Abstract

Integrity constraints play a key role in the specification and development of software systems since they state conditions that must always be satisfied by the system at runtime. Therefore, software systems must include some kind of integrity checking component that ensures that all constraints still hold after the execution of any operation that modifies the system state. Integrity checking must be as efficient as possible not to seriously slow down the system performance at runtime. In this sense, this paper proposes a set of techniques to facilitate the efficient integrity checking of UML-based software specifications, usually complemented with a set of integrity constraints defined in OCL (Object Constraint Language) to express all rules that cannot be graphically defined. In particular, our techniques are able to determine, at design-time, when and how each constraint must be checked at runtime to avoid irrelevant verifications. We refer to these techniques as incremental because they minimize the subset of the system state that needs to be checked after each change by assuming that the system was initially in a consistent state and just reevaluating the elements that may have been affected by that change. We also show how the techniques can be integrated in a model-driven development framework to automatically generate a final implementation that automatically checks all constraints in an incremental way.

Keywords: model-driven development, efficient constraint checking, incremental, runtime checking, UML, OCL

1. Introduction

A software system must include a formal representation of the knowledge of the domain and of the state of that domain to perform its functions. These representations are usually known as the Conceptual Schema (CS) and the Information Base (IB) of the software system, respectively [34].

A CS consists of a set of taxonomies of entity types (i.e. classes in the UML terminology) and relationship types (i.e. associations). At a given time \( t \), the IB contains the population of these entity and relationship types at \( t \). CSs include also the definition of the operations that may update the contents of the IB and the definition of the relevant integrity constraints [34] (i.e. invariants), which state conditions that the IB must always satisfy.

As a simple example consider the CS of Figure 1.1, a partial representation of a car rental system managing information about customers driving cars, the rentals they do, the types of cars rented and the branches where the cars are picked up and dropped off. The CS involves several entity types (such as Car, Rental or Branch) as well as some relationship types (such as Drives). At any time \( t \), the IB for this CS will consist of the set of instances of these types (e.g. the car and rental objects and the links between them) existing at \( t \).

The CS includes also several integrity constraints defined in OCL (Object Constraint Language, [48]). A precise and unambiguous definition of the integrity constraints using a formal language like the OCL provides several benefits [9]. Moreover, it is a mandatory requirement in current MDD (Model-Driven Development, [53]) and MDA (Model-Driven Architecture, [50]) approaches, where models are the primary artifact of the
-- BlackListed people cannot have active rentals
context BlackListed inv NoRentals:
  self.rentalsAsDriver->forAll(r| r.state<>'active')

-- The driving license must be valid throughout all the rental period
context Customer inv ValidLicense:
  self.rentalsAsDriver->forAll(r| r.agreedEnding>=self.licenseExpDate)

-- A car cannot be assigned to more than one rental at the same time
context Car inv OnlyOneAssignment:
  self.rentalsAsDriver->forAll(r| r.state='active')->size()=1

-- Branches must have available cars of all possible groups
context CarGroup inv QuotaForAllBranches:
  self.carGroupQuota->size()=Branch::allInstances()->size()

-- The company must have a fleet of at least 100 cars
context Car inv CarFleet:
  Car::allInstances()->size()>=100

Fig. 1.1. CS used as a running example

development process and, thus, they must contain all the relevant information, e.g. for code-generation purposes.

Within this automatic generation, software components must be developed to ensure that the IB always satisfies all integrity constraints at runtime. Efficiency of such components is a crucial issue since they must be executed every time that an operation or event changes the state of the IB. Unfortunately, current code-generation methods and tools present a rather limited support for integrity constraint checking since either they handle only a limited subset of constraint types or they generate an inefficient software component, resulting in a poor runtime performance.

To overcome these drawbacks we propose three techniques that help to achieve efficient constraint checking at runtime by providing helpful information at design-time regarding when and how each constraint should be checked depending on the kind of changes applied to the IB. Given a UML CS with OCL constraints, our techniques allow determining the events that may eventually violate a constraint, how to retrieve the set of instances that will need to be evaluated after those events and the best redefinitions of the original constraints’ contexts and expressions so that only those relevant instances are checked when evaluating the constraints at run-time.

For example, consider the constraint ValidLicense (Figure 1.1) which states that customers must have a valid license throughout all the rental period. A naïve approach to check ValidLicense would evaluate the constraint over all instances of Customer after any execution of any operation. This approach is largely inefficient since it involves several unnecessary verifications. For instance, ValidLicense may not be violated (and so it must not be evaluated at all) after the application of an operation UpdateName(c:Customer, n:String), aimed just at updating the name of a customer. Moreover, after the application of AddRentalToCustomer (r:Rental, c:Customer), which may certainly violate the constraint since it creates a new link between a customer c and a rental r, we do not need to check all customers but just c, which is the one modified by the operation. Moreover, among all rentals of c, only the new rental r may violate the condition.

The techniques we propose allow detecting that no checking is necessary in the first scenario and that, for the second one, only the expression c.licenseExpDate < r.agreedEnding should be checked at runtime to determine whether ValidLicense is violated after adding rental r to c.

Our techniques may be used separately but optimal results will only be obtained if they are integrated into a single method, as we also propose in this paper. The main goal of this integrated method is to get incremental checking of OCL constraints by considering as few instances of the IB as possible during the computation of constraint violations. Our method is incremental because it avoids evaluating the full IB population during the process of integrity checking.

To be able to incorporate the efficiency improvement provided by our techniques into current code-generation methods, we also propose how to automatically extend the initial CS so that an automatic implementation of the extended CS results directly in a set of efficient software elements with regards to the process of integrity checking.

Note that, since efficient integrity checking is embedded in the same elements of the CS (i.e. using standard UML elements) any CASE tool with code-generation capabilities can be used to (automatically) generate the implementation of the software system. This facilitates the development of the system while improving its quality and runtime performance. Moreover, this may be achieved regardless the final implementation language or platform where the system is going to be implemented.

It is worth to remark that we assume, as usual, that the initial CS is effective, clear and easy to use for the designer [56]. Including efficiency aspects into the extended CS is not contradictory with this definition since this extended CS is used only for internal purposes (i.e. the generation of efficient integrity checking components) and it is hidden to the designer since it is automatically obtained from the initial one. Formalizing efficient integrity checking in a UML/OCL terminology is analogous to what has been traditionally done with logic in deductive databases [31].

Although our techniques focus on OCL constraints, they also handle all kind of UML graphical constraints once transformed into a textual representation [30].

The work reported here extends our previous work in several directions. Firstly, the analysis techniques we propose in this paper are refined and extended versions of
the preliminary work reported in [13] [14] [15]. Secondly, we show how to use the techniques to automate the full software development process. Thirdly, we provide an exhaustive example that focuses on all relevant features of these techniques and we state the efficiency results we achieve while using them. Finally, a detailed comparison with related work is provided.

This paper is structured as follows. Section 2 introduces basic concepts. Section 3 defines the different techniques we propose for incremental integrity checking. Section 4 shows how to combine them into a single method while section 5 applies this method to our running example. Section 6 integrates our method in the context of MDD and Section 7 compares with related work. Finally, section 8 presents our conclusions and points out future work.

2. Basic Concepts

To achieve an efficient integrity checking, our techniques reason from the type of the events applied on the IB and from the internal structure of each integrity constraint (in terms of its representation as an instance of the OCL metamodel). The particular set of structural event types we consider in this work is described in Section 2.1. The representation of the constraints in terms the OCL metamodel is introduced in Section 2.2.

2.1. Event Types

A structural event is an elementary change in the population of an entity or relationship type. Each modeling language defines its own set of event types. Structural events are then defined as instances of these types.

The event types we consider in this paper are independent of any particular modeling language. Therefore, the techniques proposed in this paper can be applied to other sets of event types provided that the correspondence between those types and ours is provided. In particular, our event types do not match exactly with those of UML (see the Action packages [49]). However, it is not difficult to express our event types in terms of those of the UML.

The list of event types we consider is the following:

- **InsertET(ET):** it represents the insertion of a new object in the entity type ET. The new object may have its attributes initialized but it does not participate in any relationship.

- **UpdateAttribute(ET, Attr):** it updates the value of the attribute Attr of an instance of the entity type ET.

- **DeleteET(ET):** it deletes an object from entity type ET.

- **SpecializeET(ET):** it specializes an object of a supertype of an entity type ET to ET.

- **GeneralizeET(ET):** it generalizes an object of a subtype of an entity type ET to ET.

- **InsertRT(RT):** it inserts a new relationship (i.e., link) in the relationship type RT.

- **DeleteRT(RT):** it removes a link from a relationship type RT.

At runtime, structural events instantiate the previous structural event types to update the contents of the IB. To instantiate an event type we must replace the ET, RT and Attr parameters with the reference to the specific model element we want to modify. We also need to add all the additional parameters required to identify the particular object/link changed by the event (and its new value if necessary). For instance, the event InsertET(Car, ‘1234-BY’) would insert a new car in the IB with ‘1234-BY’ as a registration number; UpdateAttribute(Rental, bestPrice, r, 200) updates the best price of a rental agreement r to a value of 200; SpecializeET(BlackListed, c) turns a customer c into a blacklisted customer and InsertRT(Drives, r, c) creates a new relationship between customer c and rental r.

For taxonomy hierarchies, we assume InsertET events over a subtype s are preceded by InsertET events over all supertypes of s. DeleteET events over a type t are accompanied by DeleteET events over all supertypes and subtypes of t. Similarly, to specialize (generalize) an entity e of type t to a type t\_final which is not a direct subtype (supertype) we also require the corresponding specialize (generalize) events for all types t’ appearing in the path between t and t\_final (at least for one of the paths, if several different paths exist).

The creation of an instance of reified entity types (i.e., association classes) is defined by an InsertET (to create the entity facet) followed by an InsertRT event (to create the relationship facet). Likewise, to delete an entity from a reified entity type, DeleteET and DeleteRT events must be combined.

2.2. OCL Metamodel

OCL constraints are always defined in the context of a specific type, called the context type of the constraint. The actual OCL expression stating the constraint condition is called the body of the constraint. The body of a constraint is always a boolean expression that must be satisfied by all instances of the context type. This implies that the evaluation of the body expression over every instance of the context type must return a true value. As an example, in the ValidLicense constraint Customer is the context type, the variable self refers to an instance of Customer and the date comparison must hold for all possible values of self (i.e., all instances of Customer).

Instead of processing the constraints at the concrete syntax level, our techniques work with their abstract syntax representation, i.e., assuming that the constraint is expressed as an instance of the OCL metamodel. Intuitively, an instance of the OCL metamodel for an OCL expression is the equivalent of an annotated syntax tree for
the expression. In this section we review basic concepts related to the representation of OCL integrity constraints (invariants in the OCL terminology) as instances of the OCL metamodel.

The basic structure of the OCL metamodel (see Figure 2.1) consists of the metaclasses OCLExpression (abstract superclass of all possible OCL expressions), VariableExp (a reference to a variable, as, for example, the variable self), IfExp (an if-then-else expression), LiteralExp (constant literals like the integer ‘1’) and PropertyCallExp which is a supertype for the metaclasses ModelPropertyCallExp (expressions referring to model elements) and LoopExp (iterator expressions).

ModelPropertyCallExp (Figure 2.2) can be split into AttributeCallExp (a reference to an attribute), NavigationCallExp (a navigation through an association end or an association class) and OperationCallExp. This later class is of particular importance, because its instances are calls to operations defined in any class of the CS. This includes all the predefined operations of the types defined in the OCL Standard Library [48], such as the add operator (+) or the ‘and’ operator. These OperationCallExp expressions may include a list of arguments if the referred operation has parameters.

When expressing a constraint as an instance of the OCL metamodel, the body of the constraint can be regarded as a binary tree (sort of an abstract syntax tree) where each node represents an atomic subset of the OCL body expression (an instance of any metaclass of the OCL metamodel: an operation, an access to an attribute or an association …). 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 body of an iterator expression if the node represents one of the predefined iterators defined in the OCL standard (a forAll, select…) or the argument of the operation if the node represents a binary operation (such as ‘>’, union, ‘+’,…). In this latter case, the source can be regarded as the first operand of the operation.

We show in Figure 2.3 the constraint ValidLicense as an instance of the OCL metamodel. To facilitate its graphical presentation we will express the constraints using a simplified representation of the OCL metamodel, where we combine in the same tree node the kind of OCL subexpression and, in parentheses, the name of the element referenced by the node.

In this example, the forAll iterator (represented as an instance of the metaclass IteratorExp) is the root of the tree. The left child of the root is the source of the iterator (self.rentalsAsDriver). In its turn, this left child is represented as an instance of the AssociationEndCallExp metaclass corresponding to the navigation through the association end rentalsAsDriver. Its source is the access to the self variable. The right child of the root is the body of the iterator expression (r.agreedEnding <= self.licenseExpDate). The root of this right subtree is the operation ‘<=' represented as an instance of the
OperationCallExp metaclass having as referred operation the operation called ‘<=’. This node has two children. The first one is its source, an access to the attribute agreedEnding (with a last child representing the access to the r variable). The second one has the same structure, an access to the licenseExpDate attribute followed by an access to the self variable.

3. Techniques For Improving Integrity Constraint Checking

There are three different goals that must be achieved to ensure that constraint violations are efficiently checked at runtime after executing a certain operation. Firstly, a constraint must only be taken into account if the kind of changes performed by the operation may induce a violation of the constraint. Secondly, only the set of instances that may cause that violation should be considered. Thirdly, the constraint expression being evaluated needs to be appropriately defined.

In this section, we propose three different techniques that allow us to achieve each one of the previous goals. In this way, efficiency is obtained by identifying the constraints that may have been violated by the operation execution (section 3.1); by determining the minimal set of instances of the IB that must be evaluated when checking those constraints (section 3.2) and by expressing the constraints in an optimal way with respect to the changes applied by the operation (section 3.3).

Throughout the paper we consider, as usual, that the IB is initially in a consistent state (i.e. all constraints are satisfied by the system data before the execution of the operation). We also assume that the operations are specified in an imperative way [57]. In an imperative operation definition, designers explicitly define all events that must be applied on the IB when the operation is executed and thus no ambiguity problems (e.g. the frame problem) need to be considered. Alternatively, designers may prefer to first provide a declarative specification of the operations’ effect (by means of pre/postcondition contracts). These declarative specifications may be then transformed into imperative ones by hand or following an automatic transformation method as [12] before applying our techniques. Note that this declarative-to-imperative transformation is always necessary (regardless of the use of our method) before generating the implementation of the final system. In fact, some development methods only support imperative operation specifications, e.g. [42].

3.1. Computing the PSEs

Our first technique is aimed at computing the structural event types that may violate an integrity constraint. With this information, we may easily identify the subset of constraints that are not violated when applying a certain set of structural events over the IB. Clearly, those constraints can be skipped at runtime when checking the consistency of the new IB state after executing operations performing those kinds of changes, avoiding unnecessary verifications. For instance, as we have seen before, constraint ValidLicense is not affected by changes on the name of a customer and thus after executing an UpdateName operation, the software system does not need to check whether ValidLicense still holds improving this way the efficiency of the integrity checking process.

Given a constraint c and an event type ev, we have that ev is a potentially-violating structural event type (PSE) for c if the application of an event of type ev to a consistent state s of the IB may result in a state s’ that does not satisfy c. We assume that c is not satisfied in s’ if, for some instance of its context type, its body evaluates to false.

To compute the PSEs we analyze the tree representation of the constraints as an instance of the OCL metamodel. We first annotate each node in the tree to indicate whether the constraint may be violated either by increasing or decreasing the value or the number of elements returned by the evaluation of that node. This is decided depending on the values of the upper node/s in the tree. Then, annotations of the nodes and the kind of elements they reference allow us to determine the PSEs.

Some of the PSEs may be manually discarded by the designer if he/she decides that users will not be allowed to execute them on that particular system (e.g. updates of customer id attribute). This way, generation of integrity checking components for those events will be avoided in the next phases of the process.

3.1.1. Annotating the OCL Constraint Tree

Each node of the tree is annotated according to three different symbols:

- ‘+’ if the constraint can be violated by an increase on the value of the number of items returned by the node.
- ‘-’ if the constraint can be violated by a decrease.
- ‘–‘ if the constraint node itself does not provide new information for the computation of the PSEs (though it may influence the symbols of the children nodes).

![Fig. 3.1. Annotated tree for constraint ValidLicense.](image)
For instance, the result of marking the tree of the constraint ValidLicense, which states that the driving license of a customer must expire after the end of all his/her rentals, is shown in Figure 3.1. In the tree, ‘und’ symbols are shown as blank symbols.

In this figure, the navigation through the association end rentalsAsDriver is marked with ‘+’ since adding new rentals to the customer (that is, changing the IB to a state where the evaluation of this node returns more elements) may induce a violation of the constraint (adding new rentals to a customer may induce a constraint violation since the ending date of the new rental may go beyond the license expiration date of the customer). The access to the attribute licenseExpDate is also annotated with ‘+’ since extending the date of the rental (i.e., increasing the value of the attribute) may also violate the constraint. Instead, access to the attribute licenseExpDate is annotated with ‘-’ since now it is not an increase but a decrease in the value of the attribute what may induce a constraint violation.

More formally, to annotate the nodes of a tree we proceed as follows. First we initialize the root node with the und symbol. Then, we perform a preorder traversal of the tree. When visiting a node we annotate the children nodes depending on the type of the OCL subexpression represented by the node and its annotation (computed when visiting its parent node), as defined in Tables 3.1 to 3.3.

In each of these tables, rows and columns contain the possible values for annotations and expressions of the (parent) node while the cells indicate the marks that must be attached to the child node(s) (for nodes with two children nodes, symbols for each node are separated with a vertical line). Sometimes a node may propagate more than one symbol to its children. For parent nodes annotated with several symbols, the final value that the node propagates is obtained by applying the table information to each symbol. For the sake of clarity, propagation of und symbols is represented by blank cells. Blank cells can also indicate that there are no constraints including such combination.

Types of OCL expressions that may only appear in leaf nodes are omitted from the tables.

First, Table 3.1 shows the propagation of some basic OCL expressions like navigations through association ends or access to attribute values. For navigations we propagate the same symbol received. For instance, if the constraint may be violated when a navigation through an association end returns more objects (symbol ‘+’), an option to increase the number of objects returned by that navigation is to increase the size of the input collection of objects where the navigation is applied over (that justifies the propagation of the ‘+’ symbol to the child node). Likewise with the ‘-’ symbol. For attributes we always propagate a ‘+’ symbol, since a condition over an attribute may be violated if we create a new instance in the IB initialized with a wrong attribute value (it does not matter if the constraint may be violated by a decrease or an increase in the attribute value).

Then, Table 3.2 presents the treatment of predefined operations for primitive types. Among them, the ‘*’ (or ‘/’) operator is an example of a node with two children. In this case we propagate both symbols to each child (an equality comparison may be violated either by an increase or decrease in the values of any of the operands).

It is worth to remark the different behavior of arithmetic operators (as ‘*’ or ‘/’) depending on the type of the operands. For natural operands, it is clear that if a constraint can be violated when, for instance, the result of a multiplication increases, we can only get this increase by
growing the value of one (or both) operands (this justifies the propagation of the symbol ‘+’ to both children nodes). This is not true for real or integer operands because they may take negative values. In such a case, even a decrease in the value of an operand may result in a higher result after their multiplication (for instance, \(-2*1<-5*1\)). Then, when dealing with real or integer values we must propagate always both symbols (‘+’ and ‘−’). For and and or operations we propagate an und value to both children. For not operators we also propagate an und value (a special treatment for not operators is presented below).

Finally, Table 3.3 describes the propagation for iterator and collection expressions. A forAll can be violated if the number of elements that must verify the forAll body increases (that is why we propagate a ‘+’ symbol to the left child) or if some of the elements affected by the forAll changes its value in a way that does not satisfy its body. The events that may cause this change are the same events that we would obtain when dealing with the forAll body stand-alone, and thus, we just propagate the und symbol. We follow a similar reasoning with the select iterator. For the generic iterate operator, we propagate all symbols since this generic iterator is too expressive to be able to determine a more specific treatment. For collection operators (as union, intersection,…) the propagation depends on the semantics of each operator. For instance, the result of a union may increase if we increase the size of its operands. The asX operation represents the different conversion operations between the different collection types (operations asSet, asBag, …). As in the previous table, the behavior of the sum operator differs depending of the type of the objects where the sum is applied over. When they are of type integer or real, the sum operator propagates also the symbols in brackets.

Using the information on these tables we can automatically determine the annotations of all kinds of OCL constraints. For instance, the ‘=’ symbol in the node corresponding to the access to the attribute licenseExpDate (Figure 3.1) is deduced from Table 3.2 (symbol for the right child in the cell in the intersection from row ‘und’ and column ‘=’, which are the mark and type of its parent node). Predefined OCL operators not appearing in these tables can be reexpressed in terms of the ones shown here using the simplification rules presented in [16].

3.1.2. Drawing the PSEs from the Annotated Tree

Once the tree is annotated, we may determine the set of PSEs that may violate the integrity constraint. PSEs are drawn from the tree nodes, depending on its type and annotations, according to the information provided in Table 3.4. This table shows the events (cells) that when applied over an expression (columns) referenced in a node annotated with a certain symbol (rows) may induce a constraint violation. The union of the obtained events for each node forms the set of PSEs for the constraint.

For instance, the InsertRT event is a PSE for navigation expressions labeled with a ‘+’ symbol (indicating that the constraint may become violated if in the new state the navigation through that association end returns more objects than in the previous state) since this is the type of event that may cause such an increase in the number of elements returned by the navigation. In a similar way, if the navigation is labeled with a ‘−’, the critical event is the deletion of a link (event type DeleteRT) since, in this case, reducing the number of links of the association may induce the violation of the constraint.

The application of table 3.4 over the ValidLicense annotated tree gives as a result the preliminary list of PSEs shown in Figure 3.2.

Table 3.4
Computation of the PSEs

<table>
<thead>
<tr>
<th>Navigation</th>
<th>Access to</th>
<th>Access to</th>
<th>IsTypeOf</th>
<th>IsKindOf</th>
<th>allInstances</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>und</td>
<td>UpdAttr</td>
<td>GenerET</td>
<td>SpecET</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>InsertRT</td>
<td>InsertET</td>
<td>SpecET</td>
<td>InsertET</td>
<td></td>
</tr>
<tr>
<td>−</td>
<td>DeleteRT</td>
<td>InsertET</td>
<td>SpecET</td>
<td>DeleteET</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 3.2. Preliminary list of PSEs for constraint ValidLicense

In some cases, this initial result can be refined by considering additional rules that take into account a complete subtree instead of a single node. For instance, in the previous example, the PSE InsertET(Customer) could be removed since an insertion of a new customer by itself cannot violate the constraint. Only when this new customer event is followed by an InsertRT(Drives) event that links the new customer to a rental the constraint may be violated. Since this InsertRT(Drives) already appears as a PSE for the constraint, the InsertET(Customer) event is irrelevant. Therefore, the final list of PSEs for ValidLicense is the following:
- InsertRT(Drives)
- UpdateAttribute(Rental.agreedEnding)
- UpdateAttribute(Customer.licenceExpDate)
Moreover, the treatment of expressions that contain a *not* or *select* operator is more complex and it is not covered by Table 3.4.

A *not* expression may be violated by the *opposite* events of the PSEs attached to the nodes included in the child subtree of the operator. The basic idea is that, if a subexpression $s$ has $ev_1...ev_n$ as PSEs, the subexpression *not s* will have $\text{opposite}(ev_1)...\text{opposite}(ev_n)$ as PSEs. By *opposite* events we mean considering insertions instead of deletions, generalizations instead of specializations, and the other way around.

A similar thing happens with *select* expressions. When the constraint may be violated by an increase in the number of elements returned by the *select* expression we are interested in the events that favor an object to satisfy the *select* condition. As for the *not* operator, these events are the opposite of the events that favor the violation of the *select* condition.

### 3.2. Determining the Relevant Instances

Our second technique allows determining the subset of instances of the context type of a constraint that need to be considered during integrity checking after the application of an arbitrary set of events over the IB. In this way, efficiency of integrity checking is improved because the constraint can be evaluated only over these relevant instances instead of over the full type population. As a trivial example, assume that an operation changes the *licenseExpDate* of a customer $c$. After executing the operation it may happen that *ValidLicense* becomes violated. However, to check that, we just need to evaluate *ValidLicense* on $c$, it is unnecessary to evaluate all other customers as well since they have not been modified by the operation.

More formally, a constraint $c$ must be satisfied by all instances of its context type $t$ in each state of the IB. An event $ev$ applied over the IB will only affect in general a (minor) subset of the instances of $t$. Therefore, only those relevant instances must be considered during integrity checking since they are the only ones that may violate $c$. Clearly, if $ev$ is not a PSE for $c$, then no instances of $t$ should be evaluated.

Roughly speaking, the relevant instances of $c$ with regards to $ev$ is the set of instances of $t$ that are directly or indirectly related to the instance of the IB updated by $ev$. The relevant instances of $c$ after the application of a set of events $ev_1...ev_n$ are the set union of the relevant instances for each event. This implies that if the same instance is affected by more than one event, the instance is only evaluated once.

For instance, to verify *ValidLicense* after updating the *licenseExpDate* of a customer $c_1$ and adding a new rental (or adding $n$ rentals) to customer $c_2$, we just need to check that $c_1$ and $c_2$ still satisfy the date condition of the constraint. All other customers have not been changed, and thus, we are sure that they are already consistent with the constraint. However, updating the attribute *agreedEnding* of a rental $r$ requires evaluating all customers related to $r$ to check the constraint.

To compute the relevant instances of a constraint $c$ (i.e., the ones that need to be checked) after the application of an event of type $ev$ over the IB we must first distinguish whether $ev$ is classified as an instance event or a class event for $c$. Intuitively, changes due to instance events may be checked by considering only a subset of the population of their context type; while changes due to class events require always examining the whole population of the constraint context type. For instance, all PSEs for *ValidLicense* are instance events since we can always determine the exact subset of customers that must be checked after their execution. However, for *CarFleet* all events are class events, since we always need to access the whole car population after any change. We can also have constraints with both types of events, as in *QuotaForAllBranches*. For this constraint, deleting a *CarGroupQuota* link requires to reevaluate just the affected *CarGroup*. However, adding a new branch implies checking all car groups.

The previous classification may be drawn from an analysis of the OCL tree of the constraint. Roughly, instance events must be attached to a node of a subtree depending on a contextual instance (i.e., a *self* variable). Otherwise, they are class events.

Formally, we say that an event $ev$, attached to a *node* $node_{ev}$ in the constraint tree, is an instance event for an constraint $c$ when the last node of *PathVar*(node$_{ev}$) is a *VariableExp* node (see Figure 2.1) with *self* as a referred variable. The operator *PathVar*(node$_n$) returns the ordered sequence of nodes encountered in the OCL tree between $n$ and the node representing the initial *self* variable or the *allInstances* operation of the subtree to which $n$ belongs. In the absence of *union*, *intersection* and *difference* operations, that sequence is unique and computed as follows:

- The first node in the path is $n$.
- For each node $n$ in the path, its child node belongs also to the path. When the node has two children, the left child is the one included in the path.
- For each node $n$ in the path that represents a variable other than *self* (i.e., variables used in *select*, *collect* or *forAll iterators*), the path contains also the left child of the node pointed to in $n$.*referredVariable*.loopExpr (i.e., the node representing the iterator expression, *referredVariable* and *loopExpr* are navigations defined in the OCL metamodel, see Section 2).

As an example, Figure 3.3 shows the *PathVar* corresponding to the event *UpdateAttribute*(agreedEnding) in constraint *ValidLicense*. The numbers indicate the order in which the nodes are added to the *PathVar* sequence.
A collection operation does not always have a single origin since it may endow the union (or intersection or difference) of several other collections $col_1, \ldots, col_n$. In this case, PathVar returns a different sequence for each collection $col_i$. Each of those sequences is obtained by considering a different path every time we find a node representing one of such collection operations. Similarly, when the same event $ev$ appears in different nodes of the tree we must repeat the process for each node. In both situations, if at least one of the PathVars ends with an allInstance operation the event will be classified as a class event.

Once we have classified the events, we can proceed to compute the relevant instances. For class events, the set of relevant instances is the whole population of the context type. For instance events, the relevant instances are computed as follows. Given an integrity constraint $c$ defined over a context type $t$ and an event $ev$ that modifies an instance $i$, the set of relevant instances of $c$ with regards to $ev$ is determined by the path $p$ required to navigate back from $t$ to the related instances of $i$. These related instances are the only instances of $t$ that can be considered as affected by $ev$, and thus, the only instances that need to be verified.

The path $p$ is obtained through the function $\text{Inverse(PathVar}(node_{self}))$, which is defined as follows:

- The result contains only the nodes of PathVar that represent navigations through relationship types (i.e., instances of the NavigationCallExp metaclass).
- All these nodes are reversed in $\text{Inverse}$ by replacing the referred role with the opposite role in the relationship (or reified type) being navigated.
- The subexpression "select(o.ooclIsKindOf(t))" is added at the end of the navigation path if the result of reversing the last node is a supertype of $t$. This condition helps to ensure that only instances of $t$ are considered.

$\text{Inverse}$ returns the sequence of roles that must be considered to navigate from the instance modified by the event to the related instances of $t$. For instance, the application of $\text{Inverse}$ on the PathVar of Figure 3.3 would return the role $\text{driver}$ (opposite of rentalsAsDriver, the role referenced in the single node of PathVar representing a navigation), meaning that after updating the $\text{agreedEnding}$ attribute of a rental $r$ the set of relevant customers when checking ValidLicense is obtained by evaluating the expression $\text{r.driver}$, i.e., the constraint just needs to consider the set of customers related to the modified rental.

Note that if $\text{Inverse}$ returns an empty sequence of navigations it just means that the instance modified by the event is exactly the only relevant instance of $t$. That is what happens for instance, when updating the attribute licenseExpDate in the same ValidLicense constraint. In this case, PathVar consists of two nodes, the access to the attribute and the access to the self variable. Since no navigations are included, the result is an empty sequence, i.e., after updating the licenseExpDate of a customer $c$, only that customer needs to be evaluated.

When $ev$ generates different PathVar sequences, the relevant instances of $c$ with regards to $ev$ are computed by joining (set union) the relevant instances obtained by applying the $\text{Inverse}$ function to each PathVar.

3.3. Redefining a Constraint in Terms of the Best Context Type

In general, an integrity constraint may have several semantically-equivalent OCL representations, mainly depending on the context type used to express the constraint. The choice of the context type of a constraint is usually made for the sake of understandability or personal preferences of the designer. Then, it is likely to happen that a direct implementation of the selected representation is not the most appropriate for efficient integrity checking.

The third technique we propose aims at identifying the best context type of the constraint (as far as integrity checking is concerned) and replacing the original body expression of the constraint by a new, equivalent one, defined over this best context type. Designers may use these alternative representations to optimize integrity checking.

In general, there will be a different best context type (and a different redefine) for each event type that may violate the constraint. For instance, we have seen in the previous section that the relevant instances for ValidLicense after an UpdateAttribute(agreedEnding) event on a rental $r$ are all customers related to $r$. Given one such costumer $c_i$, the current representation of ValidLicense would verify that the agreedEnding of all rentals for $c_i$ happens after the expiration date of his/her license. It is not difficult to see that this is not an optimal result since it would suffice to compare the license expiration date of $c_i$ with the agreedEnding of $r$. Then, for this event, the optimal representation of ValidLicense would be:

```
context Rental inv ValidLicense :
self.driver->forall(d|self.agreedEnding <= d.licenseExpDate)
```

where now the computation of relevant instances would determine that $r$ is the only relevant instance and the
evaluation of $ValidLicense'$ over $r$ would just compare $r$ with its customers, the most efficient way of checking the consistency of the IB after this kind of update events.

Given a constraint $c$ and an event of type $ev$, the best context for $c$ with regards to $ev$ (i.e. the best context to express $c$ in order to optimize the verification of $c$ on the new IB state resulting from the application of an event of type $ev$) is automatically drawn from the node $node_{ev}$ where $ev$ is assigned to in the tree representing $c$. If there are more than one such nodes $node_{ev}$, we may end up with several different best context types and, consequently, with several redefinitions of the original constraint. In this case, all of them must be taken into account during integrity checking to ensure that $c$ is not violated after the application of $ev$.

The identification of the best context type depends on the kind of condition to which $node_{ev}$ belongs in the OCL tree. In this sense, we distinguish two different kinds of conditions: individual and collection ones. Individual conditions must be verified independently for each instance involved in the condition while collection conditions must be verified as a whole by the set of involved instances. Intuitively, nodes in collection conditions are those nodes that belong to a subtree that as a root node has a select iterator or an aggregate operation (as $sum$, $size$ or $count$).

Formally, we say that a node $n$ participates in a collection condition when $n$ is used to compute an aggregate operator or a select iterator, i.e., when $n$ verifies that:

\[
\exists n \mid n \in \text{PathRoot}(n) \text{ and } (n'.\text{oclIsTypeOf(OperationCallExp)} \text{ and } n'.\text{referredOperation} \in \{\text{size, sum, count}\}) \text{ or } (n'.\text{oclIsTypeOf(IteratorExp)} \text{ and } n'.\text{name} = \text{'select'})
\]

where $\text{PathRoot}(n)$ is defined as the ordered sequence of nodes encountered between $n$ and the root of the tree. A node participates in an individual condition if it does not participate in a collection condition.

For instance, all nodes in the constraint tree for $ValidLicense$ (see Figure 3.2) participate in individual conditions. Instead, all nodes in the constraint tree for $QuotaForAllBranches$ (see Figure 3.4) except for the equality comparison root node, participate in collection conditions since both children subtrees have a size operator as a root of the subtree.

Since individual conditions must hold for each individual entity restricted by the constraint, the best context type for an event in an individual condition is the type of the entity (or relationship) modified by the event. Then, when such an event is issued, the constraint may be checked just by applying the constraint body over the instance it modifies. Note that when the event modifies an instance involved in an $\text{oclIsKindOf}$ or $\text{oclIsTypeOf}$ expression, the best context type is the type of the elements at the child node of $node_{ev}$. This is necessary to ensure that the instance is evaluated at the appropriate level of the taxonomy, as required by the expression that defines the constraint.

Following this definition, we determine that the best contexts for the PSEs of $ValidLicense$ are the following:

- InsertRT(Drives). Best context: Drives
- UpdateAttribute(agreedEnding). Best context: Rental
- UpdateAttribute(licenceExpDate). Best context: Customer

The same idea cannot be applied to event types included in collection conditions since those conditions must be satisfied by the collection as a whole. Instead, we should use as context type the type of the entity used as the starting object to obtain the collection of entities that the whole collection condition must verify.

Given an integrity constraint $c$, an event $ev$ and the path returned by $\text{PathVar}(node_{ev})$, the node origin of a collection condition $node_{ev}$ is defined as:

- The left child of the first node $n \in \text{PathVar}(node_{ev})$ representing a $\text{forall}$ iterator, when a select iterator is not encountered between $n$ and the last node in $\text{PathVar}(node_{ev})$.
- Otherwise, the last node in $\text{PathVar}(node_{ev})$ (i.e., the node representing the $self$ variable or an $\text{allInstances}$ operation).

The best context type for an event $ev$ of a type included in a collection condition is the type of the entities at $node_{ev}$ if this node does not represent the $\text{allInstances}$ operation. Otherwise, the current context type of the constraint is returned as the best context.

For all events in $QuotaForAllBranches$ constraint the best context is already $\text{CarGroup}$ since all events are included in a collection condition where $node_{ev}$ is the last node in $\text{PathVar}$ (there are no $\text{forall}$ iterators in the constraint).

Once we know the best context type of a constraint with regards to an event, we can obtain the redefinition of the original integrity constraint in terms of the new best context by applying the transformations defined in [16].

Fig. 3.4. $QuotaForAllBranches$ constraint
4. Putting it All Together: Generating Incremental Expressions for the Constraints

The techniques we propose may be individually used to get a partial improvement of integrity checking efficiency. For instance, in a certain situation, we could be interested only to use the first technique and to discard the constraints that may not be violated by an operation without worrying about the other two aspects. Nevertheless, optimal results for efficient integrity checking may only be achieved if the advantages individually provided by each technique are integrated into a single method. This is in fact the main objective of this section: to define a method for incremental integrity checking of OCL constraints, which integrates the previous techniques.

The main goal of our method is to replace each integrity constraint \( c \) of the original CS with a set of equivalent boolean incremental expressions \( \text{set}_c \), such that each \( c' \in \text{set}_c \) leads to an incremental evaluation of \( c \) with regards to some of the event types that may violate it. In particular, for each constraint \( c \) and for each structural event type \( ev \) that may violate it, our method generates an incremental expression \( c' \) that can be used instead of \( c \) to check that the application of an event of type \( ev \) has not evolved the IB to an inconsistent state with respect to \( c \). If \( c' \) holds we may conclude that \( c \) is not violated in this new state. Since \( c' \) is created according to the previous efficient techniques, the cost of evaluating \( c' \) is much lesser than the cost of evaluating \( c \) (see the next section for a discussion on the efficiency gains). In this way, our method ensures an incremental evaluation of the constraints regardless of their specific syntactic definition in the original CS.

The definition of \( \text{set}_c \), for a constraint \( c \) works as follows (Figure 4.1). First, the PSEs for \( c \) are determined as explained in section 3.1.

As an example, consider again the `ValidLicense` constraint. We know from Section 3.1 that its PSEs are:

- `PSE_1`: `UpdateAttribute(Customer, licenceExpDate)`
- `PSE_2`: `UpdateAttribute(Rental, agreedEnding)`
- `PSE_3`: `InsertRT(Drives)`

Now, we compute at step 2 the most appropriate syntactic representations of \( c \) with regards to each PSE \( ev \). First, we identify the best context type \( t_c \) of \( c \) with respect to \( ev \) and then we redefine \( c \) in terms of \( t \) (as shown in section 3.3). As a result, we have now several alternative constraint representations \( c_1...c_n \), each one optimized to efficiently check violations of \( c \) after a subset of the PSEs for \( c \). Note that \( c_j \) may be an appropriate alternative for several PSEs and that the original \( c \) representation may already be in some cases the most appropriate one.

Regarding our example, we may use the technique proposed in section 3.3 to determine that the best context types for `PSE_1`, `PSE_2` and `PSE_3` are `Customer`, `Rental` and `Drives`, respectively. Then, using [16], we obtain in step 2 that the three expressions that redefine `ValidLicense` in terms of these best contexts are the following:

1. `ValidLicense'_1: c.rentalsAsDriver->forAll(r|r.agreedEnding<=c.licenseExpDate)`
2. `ValidLicense'_2,2: r.driver->forAll(d|r.agreedEnding<=d.licenseExpDate)`
3. `ValidLicense'_3,3: d.rentalsAsDriver.agreedEnding<=d.driver.licenseExpDate`

Finally step 3 creates, for each \( c_i \) and for each one of its PSEs \( ev_k \), an incremental expression \( c'_{i,k} \) that allows checking \( c_i \) violations by considering only the set of relevant instances regarding \( ev_k \).

More specifically, the final \( c'_{i,k} \) expression is generated by means of applying the body expression \( b \) of \( c_i \) to all instances reached in \( exp \), where \( exp \) is the expression to compute the relevant instances for \( PSE_{i,k} \) obtained as we have shown in section 3.2. If the evaluation of \( exp \) returns at most a single instance (i.e., all navigations included in \( exp \) have ‘1’ as a maximum multiplicity), we just need to replace all occurrences of \( self \) in \( b \) with \( exp \). Otherwise, the final expression is \( exp \rightarrow forAll(v|b) \) where all occurrences of \( self \) in \( b \) are replaced with \( v \).

In our example, we would obtain the following three expressions:

1. `ValidLicense'_1: c.rentalsAsDriver->forAll(r|r.agreedEnding<=c.licenseExpDate)`
2. `ValidLicense'_2,2: r.driver->forAll(d|r.agreedEnding<=d.licenseExpDate)`
3. `ValidLicense'_3,3: d.rentalsAsDriver.agreedEnding<=d.driver.licenseExpDate`

where \( c, r \) and \( d \) represent the modified customer and rental objects and drives link, respectively.

The obtained expressions can be used instead of the original `ValidLicense` constraint to efficiently check that this constraint is not violated when the IB is changed. For instance, every time we update the `agreedEnding` date of a rental we should check if `ValidLicense'_1 evaluates to true on that rental instead of checking the original constraint. With these expressions we ensure that a constraint is only evaluated when one of its PSEs has been applied to the IB.
and that we consider only relevant instances that may violate the constraint when performing such computation.

5. Applying our Method to the Running Example

The main goal of this section is to illustrate our method by applying it to the running example. In particular, Section 5.1 shows the expressions we obtain to incrementally check all constraints of the example according to the method defined in Section 4 while Section 5.2 illustrates the efficiency improvement we get with these new expressions.

5.1. Applying our Method

The first step is to draw the PSEs for each constraint:

1. ValidLicense. A customer license may become invalid when changing its expiration date, when linking the customer with a new rental that may not satisfy the date condition or when updating the agreed ending of some of the related rentals. List of PSEs:
   a. UpdateAttribute(Customer, licenseExpDate)
   b. InsertRT(Drives)
   c. UpdateAttribute(Rental, agreedEnding)

2. CarFleet. Removing a car may violate this constraint. List of PSEs:
   a. DeleteET(Car)

3. OnlyOneAssignment. The assignment of a new rental to a car or the reactivation of a rental may induce its violation. List of PSEs:
   a. InsertRT(AssignedCar)
   b. UpdateAttribute(Rental, state)

4. QuotaForAllBranches. Any change on the number of quotes for the car group or on the number of existing branches may induce its violation. List of PSEs:
   a. InsertRT(CarGroupQuota)
   b. DeleteRT(CarGroupQuota)
   c. InsertET(Branch)
   d. DeleteET(Branch)
   e. InsertET(CarGroup)

5. NoRentals. Adding rentals to a blacklisted customer, specializing a customer into a blacklisted one or changing the state of related rentals may violate the constraint. List of PSEs:
   a. InsertRT(Drives)
   b. UpdateAttribute(Rental, state)
   c. SpecializeET(BlackListed)

Once this is done, we determine the best context type for each pair <constraint, PSE> and we redefine the constraint in terms of this new context. This information is provided in Table 5.1.

With this information, we apply the relevant instances technique over the redefined constraints as an intermediate step to generate the final incremental expressions that will allow the efficient checking of constraints violations. The results are shown in Table 5.2. The first two columns in the table identify the constraint and the PSE to which the technique is applied, the third column shows the relevant instances of the constraint context type (where x stands for the object/link modified by the event) that must be taken into account after applying such PSE and the forth column specifies the incremental expression that could be used instead of the original constraint when checking the IB consistency.

As an example, QuotaForAllBranches, as a constraint with both, class and instance events, can be efficiently checked after some of its PSEs (insertions and deletions on CarGroupQuota and insertions on CarGroup) while for the others (insertions and deletions of branches) we must check all instances of CarGroup.

5.2. Efficiency Results

We show the benefits provided by our techniques by computing the efficiency gain they give when comparing the cost of checking our incremental expressions with the cost of checking the original constraints.
An empirical evaluation of some of our techniques has been done in an industrial environment by Altenhofen, Hettel and Kusterer [2]. They conclude that 1) the computation of PSEs reduces an 88% in average the number of objects that must be taken into account during its evaluation. Instead, here we propose a more theoretical framework and apply it to the running example to illustrate the general benefits of our approach. We define the cost of checking a constraint as the number of objects that must be taken into account during its evaluation.

It is worth noting that we cannot determine the exact cost at design time since that depends on the exact population of the entity and relationship types which are not known until runtime. We must then represent these values by means of abstract variables. However, the abstract formulas resulting from our cost analysis are clear enough to illustrate the efficiency improvement.

The cost of a direct checking of the original constraint (i.e., the cost of evaluating the body of the constraint over every instance of its context type) is always the same regardless the event applied over the IB. Instead, with our method there is a different cost formula for each pair <constraint, event type> based on the cost of evaluating the generated incremental expression for that event and constraint.

Table 5.3 defines and compares the cost formulas for the direct and efficient evaluation of the constraints. In the cells, $P_X$ stands for the population of the type $X$ (for instance, $P_{Customer}$ represents the number of instances of Customer). $N_{X,Y}$ stands for the (average) number of entities of $X$ related with an entity of $Y$ (for instance, $N_{customer-rental}$ represents the average number of rentals done by a customer) where $Y$ may be either the name of a role or the name of the destination type (when no ambiguity exists). For instance, a direct evaluation of ValidLicense requires just to check the newly linked customer and rental (and so, the cost is 2) and after the event UpdateAttribute(licenseExpDate) we only need to check the updated customer (cost 1) and all his/her rentals (cost $1 \times N_{customer-rental}$).

As a summary of our results we would like to highlight the following conclusions:

- In our approach, for all pairs <constraint, event> where...
event is not a PSE for constraint the cost is zero. In the original schema (where the determination of PSEs is not performed) the cost of non-PSE events is the same as the cost of PSEs.

- Considering just the PSEs, we improve the efficiency of integrity checking in a 79% of the cases (11/14).

- For these 79% of the events, in a 18% we reduce the checking cost to a constant value (that is, regardless of the population of the IB, the maximum cost value remains constant) and for the remaining <constraint, event type> pairs, the cost is at least independent of the population of the context type of the constraint in the schema, and thus, this variable is factored out of the cost formula.

We would like to note that our cost formulas assume that a single event is applied to the IB. When considering the application of sets of events, the average cost per event decreases because some redundant checkings are avoided with our techniques.

It is clear that the efficiency gain we achieve with our techniques depends on the CS considered and on the particular definition of the constraints. Nevertheless, we believe that the results we get in the running example are representative enough to illustrate their benefits.

6. Our Method in the Context of MDD

The advantages provided by our method may be easily integrated in the context of Model Driven Development (MDD) [53] approaches, aimed at the ultimate goal of automating software systems development using models as the primary artifacts throughout the development process.

In this sense, our goal is to automatically integrate our incremental integrity checking method within the development process, freeing the designer from performing this task. As usual, automating software development improves the quality of the resulting system while speeding up its development.

Given an initial conceptual schema CS, we perform this integration by automatically generating an extended schema CS’ that, when executed or directly implemented in a particular technology platform, is able to check at runtime all constraints incrementally. CASE tools can use CS’ (instead of CS) to generate an implementation of the system that will efficiently check the constraints in any technology platform P. See a schema of the process in Figure 6.1.

It is worth to note that, since the constraint checking behavior is already embedded in CS’, there is no need to develop specific runtime checking components for each platform. Indeed, our CS’ can be used to generate efficient system implementations in all kinds of platforms. Moreover, since all elements in CS’ are standard UML elements, we do not need to extend CASE tools to deal with CS’.

This section proposes two different techniques to generate the extended schema CS’. Both of them may be automatically applied, i.e. they do not require the designer’ intervention.

The first one is the most natural and it involves extending the operation contracts in the original CS with the incremental expressions required to efficiently check that the operation execution does not violate any constraint. To apply this technique, it is required that the only possible updates of the IB are precisely those defined by the operations specified in the behavioral part of the conceptual schema.

To be able to use our approach when this assumption is thought to be too restrictive, we propose a second technique that achieves incremental treatment of constraints by means of adding new derived elements to the CS. Therefore, this technique is more suited for software systems that are going to be implemented in a technology platform, such as relational databases which offers some mechanism to deal with derived elements. Otherwise its implementation would be too costly because the contents of such derived elements should be programmed into the information system itself.

We provide some implementation guidelines for both alternatives that show the feasibility of automatically obtaining from them an implementation of the software system that incrementally checks at runtime all possible constraint violations.

6.1. Moving Constraints inside Operation Definitions

The incremental expressions required to efficiently check the constraints may be automatically incorporated in the specification of the operations. In this way, the operations themselves will be responsible for ensuring that the IB state remains consistent after their execution.

This is achieved by:

1. Identifying all constraints that can be violated by the execution of the structural events included in an operation op.
2. Create the incremental expressions exp_c1, ..., exp_cn for each relevant constraint which suffice to check the constraint within the particular operation.
3. Including in the body of op, the expression: if not (exp_c1 or ... or exp_cn) then raise exception.

Regarding the first step, given an operation op whose effect is defined by the set of events set_ev, we have that a constraint c with cPSE as PSEs may be violated by op if set_ev ∩ cPSE ≠ ∅, that is, if some of the events executed by the operation appear in the list of PSEs for the constraint.
For instance, consider the operation UpdateName, aimed at changing the name of a customer, defined as:

```java
UpdateName(c: Customer, newName: String) {
    UpdateAttribute(Customer, name, c, newName);
}
```

We have that \( \text{set}_{\text{ev}} = \{ \text{UpdateAttribute(Customer, Name)} \} \). Since we know that \( \text{cPSE} \) for ValidLicense is \( \{ \text{UpdateAttribute(Rental, agreedEnding)}, \text{InsertRT(Drives)}, \text{UpdateAttribute(Customer, licenceExpDate)} \} \) we can conclude that UpdateName will never violate ValidLicense.

Applying the same procedure to the other constraints of the example, we may determine that this operation is safe in the sense that its execution will never violate any integrity constraint. Thus, UpdateName does not need to be extended at all.

Consider now the following operation:

```java
ExtendRentalEnding(r: Rental, newEnd: Date)
{
    UpdateAttribute(Rental, agreedEnding, r, newEnd);
}
```

In this case we have that ValidLicense is a relevant constraint for ExtendRentalEnding, since the event UpdateAttribute(Rental, agreedEnding) included in the operation is one of the PSEs for the constraint. Thus, the execution of this operation may bring the IB to a new state that violates the constraint.

Once we know that an operation \( \text{op} \) may violate a constraint \( c \), we must compute the incremental expression to be added to \( \text{op} \) in order to incrementally check that the execution of \( \text{op} \) does not violate \( c \).

More precisely, given the set of events in \( \text{set}_{\text{ev}} \cap \text{cPSE} \) the incremental expression \( \text{exp}_{\text{cPSE}} \) is defined as \( \text{exp}_{\text{cPSE}} = \text{exp}_{\text{cPSE}} \cup \cdots \cup \text{exp}_{\text{cPSE}} \) i.e., as the union of the incremental expressions \( c_i \) that allow checking that a particular event \( \text{ev}_{\text{cPSE}} \) included in \( \text{op} \) does not violate \( c \). When there are several relevant constraints for \( \text{op} \), the process must be repeated for each of them and \( \text{op} \) must be extended with all corresponding incremental expressions.

Let us again consider the operation ExtendRentalEnding. As we have seen before, the event UpdateAttribute(Rental, agreedEnding, r, newEnd) of this operation may induce a violation of the constraint ValidLicense. The incremental expression \( \text{exp}_{\text{ValidLicense}} = \text{UpdateAttribute(agreedEnding)} \) to ensure that this does not happen at runtime is \( r.\text{driver}->\text{forAll}(d | r.\text{agreedEnding} <= d.\text{licenseExpDate}) \) (as computed in Section 4).

Given this incremental expression and assuming that ExtendRentalEnding does not affect any other constraint, the final body of ExtendRentalEnding would be the following:

```java
ExtendRentalEnding(r: Rental, newEnd: Date)
{
    UpdateAttribute(Rental, agreedEnding, r, newEnd);
    If not(r.driver->forAll(d | r.agreedEnding <= d.licenseExpDate)) then raise exception
}
```

Note that this modified body suffices to efficiently check the IB state before committing the operation so that we can ensure that the changes applied by the operation have not violated the constraint and raise and exception otherwise. The exception handling mechanism will determine how to treat this exception (cancel all changes done by the operation, issue compensating actions,…) according to the designer’s preferences.

This alternative is similar to the one employed by typical design by contract languages where assertions are required to be true at the end of the operation execution and an assertion violation exception is raised otherwise (see Section 7 for a comparison with such related work).

Once we have the extended operation bodies OCL-to-Java tools (see [37] [8, 25] as examples) can be used to transform the types of the CS and their operations into a set of Java classes with a public method for each different operation. These tools are also able to transform the incremental expressions into an equivalent if-then-else Java class for the Rental type with the ExtendRentalEnding operation.

```java
public class Rental {
    String id;
    int bestPrice;
    String state;
    Date agreedEnding;
    Vector<Customer> driver;

    public void extendRentalEnding(Date newEnd) throws Exception {
        if(this.agreedEnding + newEnd)
            this.driver->forAll(d | this.agreedEnding <= d.licenseExpDate))
            throw new Exception("Invalid date");
    }
}
```

Fig. 6.2. Java class for the Rental type. Only the relationship type with the customer class has been included.

### 6.2. Specifying incremental integrity checking as derived entity types

This alternative approach is aimed at automatically incorporating in the structural part of the CS the information required to incrementally check the constraints. This is done by extending the original CS with additional entity and relationship types that allow explicitly recording 1 - the different PSEs that may violate some constraint of the schema (and their effect on the IB) and 2 - the set of relevant instances (taking into account the best context for each event) that must be evaluated to efficiently determine
possible constraint violations. This extension is obtained automatically and it does not require the designer participation. All elements added to the schema are standard UML elements, no new constructs are introduced.

The size of the extended CS increases because of the addition of the new entity and relationship types. However, the number of new model elements is at most linear regarding the number of integrity constraints (in all substeps of this technique the number of added elements is \( k*n \) where \( k \) is a constant value and \( n \) the number of constraint in the CS). Moreover, the designer does not have to be aware of this new information because he does not need to participate in its definition nor use.

One additional advantage of this technique is that the preferred checking time can be determined by the designer since with this approach the information about the consistency state of the IB is always available and can be consulted at any time. For long interactions, the designer may decide to postpone checking until the end of the interaction to reduce the number of integrity checks and benefit from an analysis of the interactions between the events provided by this technique to simplify the process.

6.2.1. Definition of Structural Event Types for the PSEs

Structural event types are a specific set of entity types required to explicitly record at runtime the events issued during the update of the IB state together with the information about the instances they modify.

A structural event type is added to the CS for each possible PSE of some constraint (at most one type for each PSE regardless the number of constraints that the PSE may violate). They are denoted by \( iET \) (insert entity type), \( dET \) (delete), \( gET \) (generalize), \( sET \) (specialize), \( uETAttribute \) (modify attribute), \( iRT \) (insert relationship type) and \( dRT \) (delete), in accordance to the different event types we have defined in Section 2.1.

All structural event types are labeled with the stereotype \(<\langle structural\ event\ >>\) to differentiate them from the other entity types of the CS. Since we assume that the IB is updated at runtime with the modifications produced by the PSEs, structural event types just need to contain a reference to the modified entity, but not the actual arguments of the event (the arguments of the event can be deduced, when necessary, from their effect on the state of the referenced entity).

For \( iET \) and \( uETAttribute \) event types, the multiplicity of the relationship type relating the event and the updated entity is ‘1’ in the role next to the entity type (since events always refer to exactly one entity) and ‘0..1’ in the role next to the structural event type (since most entities will not be modified during the operation execution). For the sake of simplicity, the role next to the entity type in all these relationship types is always named as \( ref \). An example is given in Figure 6.3. We treat \( gET \) and \( sET \) event types similarly. Structural event types recording \( dET \) events do not contain any relationship type since it is impossible to relate the event to the updated entity (which does not exist any more after the application of the event). Structural event types corresponding to \( iRT \) and \( dRT \) events contain as many relationship types as the number of participants in RT. The name of the roles next to the participant entity types is always \( ref \) concatenated to the name of the role of that participant type in RT (as usual, if the name of the role is not defined it is assumed to be the name of the participant).

For \( iRT \) types, the multiplicity on the participant type role is always ‘1’ since every new relationship must be related with an instance of the participant entity type. The multiplicity of the role of the structural event type is ‘*’ since an entity of a participant entity type can participate in general in several relationships of the relationship type. This multiplicity may be restricted to ‘0..x’ when it is known that the instances of the participant entity type cannot participate in more than \( x \) relationships. See Figure 6.4 for an example.

![Figure 6.3](#)

**Fig. 6.3.** Example of an \( uETAttribute \) structural event type. Each instance of the uLicenseExpDate attribute represents the execution of an update event to change the licenseExpDate attribute of a customer. The modified customer can be retrieved using the \( ref \) association end.

![Figure 6.4](#)

**Fig. 6.4.** Example of an \( iRT \) structural event type. Since rentals cannot have more than one assigned car the role next to is \( iAssignedCar \) has a 0..1 multiplicity.

For \( dRT \) types, the multiplicity on the participant type role becomes ‘0..1’, because, after deleting the relationship, it may happen that other events delete also some of the participants of the relationship.

\( iRT \) and \( dRT \) event types which refer to a reified relationship type (or to a relationship type that will be reified as part of the next step because it is determined as the best context for one of the constraints) are treated as \( iET \) and \( dET \) event types since there is a direct one-to-one correspondence between each relationship and its reified entity.

In general, each structural event type will contain as many instances as events of that kind have been executed over the corresponding entity or relationship type. For instance, the structural event type \( uLicenseExpDate \) will contain an instance for each customer that has updated its license expiration date during the operation execution.
To improve efficiency of integrity checking (achieved by minimizing the population of those types) we adapt the concept of net effect [18] and define two additional rules for insertions and deletions over structural event types:

- An instance will only be inserted in a \texttt{uETAttribute} type if the same instance does not already appear in the types \texttt{iET} or \texttt{uETAttribute}, as well. For instance, if we update three times the license date of the same customer, we only need to record this fact once.

- If we delete an entity or a relation, the matching instance is removed from the types \texttt{iET} (\texttt{iRT}), \texttt{gET}, \texttt{sET} and \texttt{uETAttribute}. In addition, if the entity (relation) appears in \texttt{iET} (\texttt{iRT}) we do not need to record that it has been deleted.

We would like to remark that the designer can disregard the creation of structural event types corresponding to PSEs that he/she thinks do not make sense in practice for that domain like, for instance, updating a costumer \texttt{id}, or that users will not be ever able to execute (due to restrictions imposed on the use of the system).

6.2.2. The Best Context as a Derived Entity Type

As a next step, we must redefine the constraints such that we can benefit from the information recorded in the structural event types. In particular we want that for each executed event (i.e., for each instance of a structural event type) the constraint is evaluated using its redefinition in terms of the best context type for that event and only over the relevant instances affected by the event.

For constraints having all its PSEs classified as instance events (see Section 3.2), this is achieved in our approach by creating, for each different best context type \( t \) (i.e., for each type that is determined to be the best context type for at least one of the PSEs for the constraint, see Section 3.3) a new derived type \( t_{\text{sub}} \), subtype of \( t \), aimed at containing the subset of instances of \( t \) that need to be checked after executing that kind of event. Then, the constraint is redefined in terms of the population of \( t_{\text{sub}} \). This way, we improve the integrity checking by avoiding evaluating the constraint over the whole population of \( t \) and we just evaluate it over the relevant instances found in \( t_{\text{sub}} \). As an example, to process \texttt{ValidLicense} we need to add to the CS three derived subtypes (one for each best context) and three constraints (the original one plus two new redefinitions).

The fewer instances \( t_{\text{sub}} \) contains the more efficient will be the integrity checking process. Therefore, the derivation rule for \( t_{\text{sub}} \) must ensure that, at any time, its population consists of just the instances of \( t \) affected by the execution of events of event types for which \( t \) is determined to be the best context. Derivation rules are specified by means of redefining its predefined \texttt{allInstances} operation (the population of the derived type will be the instances returned by this operation) [46].

Given that \( ev_1…ev_n \) is the set of event types having \( t \) as a best context, the derivation rule for \( t_{\text{sub}} \) is created as \( \text{relevant}(ev_1) \cup ... \cup \text{relevant}(ev_n) \). Relevant \( \text{relevant} \) is computed by first determining the expression that returns the set of relevant instances of \( t \) after executing an event of type \( ev_i \) (as explained in Section 3.2) and then applying this expression over all executed events of that type (recorded in the corresponding structural event type).

For instance, in Figure 6.5, the \texttt{Rental} type is the best context for \texttt{UpdateAttribute(agreeEnding)} events. Therefore, the subtype \texttt{RentalValidLicense}_3 is created and its derivation rule ensures that its population is just the set of rentals that have changed their \texttt{agreeEnding} date. Since \texttt{ValidLicense}_3 (specialized version of \texttt{ValidLicense} for this kind of events) is defined using \texttt{RentalValidLicense}_3 as a context type, checking this constraint only requires evaluating the modified rentals instead of evaluating the whole rental population. Similarly, constraints \texttt{ValidLicense} and \texttt{ValidLicense}_2 will only check customers that have changed his/her license and the new links between customers and rentals, respectively. Note that each PSE is checked using its most appropriate constraint representation. This way we get the same effect (and efficiency level) as in the previous alternative, where the incremental expressions were directly added to the operations’ bodies.

Fig. 6.5 Extended schema for \texttt{ValidLicense} constraint

For constraints with all PSEs classified as class events, it is useless to compute the affected instances of the context type since we must always check its whole population. Nevertheless, the CS can be easily extended to ensure that this kind of constraints is not verified during the integrity checking process unless one of the executed events is a PSE for the constraint. The only thing we must do is to modify the constraint body by adding an initial conditional statement establishing that the constraint must not be computed when all the structural event types corresponding to its PSEs are empty. See the schema modification for \texttt{CarFleet} (Fig. 6.6) as an example.
For constraints with both types of events, we follow a mixed approach. When the executed operation does not include any of the class PSEs we act as explained for the instance constraints. Otherwise we act following the guidelines for class constraints. As seen in Section 4, QuotaForAllBranches (Figure 6.7) is an example of this kind of constraints. It can be efficiently checked after some of its PSEs by evaluating the relevant instances are recorded in the derived type CarGroupQuotaForAllBranches while for the others (insertions and deletions of branches) we must check all instances of CarGroup (as recorded in CarGroupQuotaForAllBranches); empty if no branches have added or removed). If condition in the derivation rule for the CarGroupQuotaForAllBranches derived subtype ensures that no redundant checks are performed (i.e., if we need to check all CarGroup instances we should not check particular car groups as well).

This kind of extended schemas is suitable for being implemented using relational databases as a technology platform since relational databases incorporate predefined mechanisms for the treatment of derived elements. The implementation of the software system in a relational database from this extended schema may be obtained by means of the following steps:

1. Entity types must be translated into a set of relational tables (i.e., domain tables) while relationship types are translated into a set of tables or foreign keys (depending on their multiplicity), as explained in [27], among many others.
2. Each structural event type is translated into a temporary relational table (i.e., event table). Temporary tables are defined in the SQL:1999 standard [41]. Data of temporary tables is truncated at the end of each transaction.
3. For each event table, we must create a trigger in the corresponding domain table. This is required to reflect in the event tables the changes done by the user over the domain tables. The trigger will be in charge of inserting/removing the tuples of the event table automatically.
4. Each derived subtype is translated into a view over the data of the event tables corresponding to structural event types referenced in the derivation rule for the subtype. The view expression may be obtained automatically from the OCL derivation rule of the derived subtype using existing translation patterns ([24], [19]).
5. For each constraint, we generate a view that returns a non-empty result if and only if the constraint has been violated during the transaction. The SELECT clause of the view is generated from the constraint definition in a similar way as done for derived subtypes. To ensure an incremental verification of the constraint, the FROM clause retrieves the data from the view created for the derived subtype used as a context type of the constraint. This way, the constraint is only evaluated over the relevant data instead of querying the full domain table. When the constraint becomes violated, the rows returned by the view contain the information about which exact instances violate the constraint. This information can be given as a valuable feedback to the user.

As an example, Figure 6.8. shows a partial translation of the extended schema of Figure 6.5. In particular we focus on the database structures required to efficiently check ValidLicense after events of type UpdateAttribute(Customer, licenceExpDate). Note that the view corresponding to ValidLicense extracts its data from the event table generated in step 2 and not from the Customer type. Thus, the process of checking the constraint only access those customers modified during the transaction, exactly as determined by our analysis at the model level.

We would like to remark that, with the exception of step 3, the previous translation steps are not specific to the translation of our extended CS. In fact, any method generating a relational schema from a standard CS must deal with the same steps to be able to translate all the elements defined in the CS.
Regarding relational databases most methods are able to generate the skeleton of relational schemas from the CS. Some admit the definition of simple checks over attribute values [55] [8] [33, 45] [21] (defined using some tool-specific stereotypes; OCL constraints are either not supported or ignored during the code generation phase) , which are translated as check constraints over the corresponding table. [54] supports multiplicity constraints and translates them as a set of triggers. [25] (based on [24]) is able to handle all kinds of OCL constraints. Each constraint is translated into an equivalent SQL view (the view selects those tuples of the database not satisfying the constraint, and thus, a non-empty view indicates that the constraint has been violated). However, the views are not efficient since they examine the whole table population instead of considering only the tuples modified during the transaction as we do in our approach. Another relevant work is [1] whose main goal is to automatically provide a set of translation patterns that derive active relational database mechanisms, i.e. triggers, to efficiently check OCL integrity constraints, ensuring the termination of its execution. OCL constraints are not pre-processed before the trigger generation.

The methods that focus on generating a Java implementation of the CS present similar limitations. Apart from generating the skeleton of the Java classes corresponding to the entity and relationship types of the
CS, few tools generate code to check the constraints. [54] adds some code to control the multiplicity constraints. [42] supports some predefined types of constraints; more general ones must be manually specified using its Action Language. [37] creates a new method for each constraint (if the constraint does not hold, executing the method raises an exception) but since no hints are provided about when to call this method the designer is in charge of manually including the call wherever appropriate. [4], [36] and [58] offer a similar functionality. [8], [26] and [51] support constraints that restrict the attribute values of a single object and of related objects. However, all constraints are checked before and after every single method of the class acting as a context type for the constraint and not only after those methods that may induce a constraint violation. Moreover, for constraints involving linked objects, possible violations due to the execution of methods placed in related classes are not considered and thus we may reach an inconsistent state. [25] provides better efficiency since constraints are only checked after methods modifying some model element referred in the constraint body. However, as we have seen in Section 3, this is still inefficient since the type of the modification must be also considered when determining the violations.

An additional drawback common to all methods is that they always depart from the constraints exactly as defined by the designer. Thus, their efficiency depends on the concrete syntactic representation chosen (which may not be the optimal one for constraint verification purposes). Moreover, the constraint is not specialized to guarantee the best efficiency for each different event that may violate it, i.e. the same representation is used regardless the executed events.

7.2. Comparison with design by contract languages

Design by contract languages (as JML [39], Spec# [6] or Eiffel [43]) allow designers to detail the preconditions and postconditions of methods as well as the class invariants. Then, the execution of the methods is monitored to ensure that their behaviour at runtime respects the specified pre and postconditions and the invariants.

This monitoring usually consists in adding (by means of a precompiler) to the method code a set of if-then-else conditions that check that the program follows the expected behaviour and generate an exception otherwise. Clearly, part of these additional conditions must be in charge of guaranteeing that the constraints do not become violated. A largely inefficient approach (but generally used, like in the jmlc precompiler tool for the JML language) adds if-then conditions to check all constraints in each method even if the method cannot violate the constraint. Moreover, the invariant expression that defines the constraint is never optimized.

As we have seen, our techniques are able to address this kind of problems by generating an optimized set of incremental expressions for each operation. Therefore, adapting our techniques to the specific syntax and constructs of these languages could improve the efficiency of their runtime checking by helping them to generate more efficient versions of the if-then conditions they use for runtime checking. Although, describing how to port them is out of the scope of this paper we believe it is feasible because of the semantic similarities of the concepts defined in each language (for instance, JML and OCL present many similarities [32]).

7.3. Comparison with program verifiers

The goal of our techniques is to minimize the cost of checking, at runtime, that the constraints are not violated when a certain operation is executed. A different (but related) problem is considered by several formal specification languages and tools (e.g. [35], [40], [7], [52], [38]) aimed at proving at design time that the operations preserve the integrity constraints. An operation op preserves an integrity constraint c if no execution of op will ever violate c.

These methods and tools are intended to prove at design-time that op already contains all necessary conditions to prevent the violation of c but none of them is able to generate additional runtime checkings when they are not able to prove it. Therefore, designers are in charge of manually extending the operation definitions with new conditions until the operation is shown to prevent all integrity constraints. This situation is likely to happen often since the problem of proving the correctness of a program (i.e. an imperative operation definition in our case) with respect to a specification (i.e. the invariants) is not decidable in general. Typically, previous tools face this problem by either requiring user intervention during the prove process (which requires the designer to be knowledgeable in the use of the formal tool) or opt for bounded verification approaches (the absence of solution cannot be used as a proof).

Our approach may also be used to help solving this problem since we know that an operation op preserves an integrity constraint c if it does not include any of the PSEs for c. Moreover, when op contains PSEs for c we may generate incremental expressions that guarantee the consistency of the IB at runtime, freeing the designer from the burden of manually designing such conditions.

Moreover, we must also mention that, to our knowledge, proposals in this area do not provide specific support for CSs specified in UML/OCL, with the notable exception of [3] that provides a partial transformation from UML to Alloy.
8. Conclusion

The main contribution of this paper is the definition of a set of techniques (and a method that integrates them) for the efficient integrity checking of OCL integrity constraints defined in a UML conceptual schema.

Our techniques allow determining which constraints must be checked as a result of an operation execution; the set of instances that must be considered and the best redefinition of each constraint such that it may be incrementally checked. We have applied our technique to a sample conceptual schema and we have shown the significant efficiency improvement provided in this case.

We have also shown that the results provided by our techniques can be directly integrated within current MDD methods and tools (which still provide rather limited support for constraints) since they work at the model level. In fact, given a MDD method $M$ able to implement a conceptual schema $CS$ in a technology platform $P$, when $M$ uses the extended conceptual schema $CS'$ (resulting from applying our techniques over the original $CS$) instead of $CS$ to implement the specified system in $P$, the code it generates will automatically check all constraints efficiently. Note that $M$ does not need to be modified to benefit from $CS'$ and that the results of our work can be helpful in any platform $P_2$ provided that a MDD method $M_2$ for $P_2$ exists.

The techniques presented in this paper have been implemented in a prototype tool. Our tool [17] is implemented as a set of Java classes extended with the libraries of the Dresden OCL toolkit [25] (for parsing and loading the OCL constraints) and NetBeans MDR [44] (for the import/export of the UML models from XMI files).

From these results, we believe that the work presented here represents a step forward in the long-standing goal of automating information systems development since they allow automating efficient code-generation from integrity constraints, one of the open problems in the area [47].

Our techniques may also help to improve the efficiency of other OCL-intensive related problems. Among them, we find model consistency checking (where the model must satisfy the OCL well-formedness rules defined at the metamodel level [49], [22]; our techniques could be used to minimize the number of rules to be checked after evolving the model) and the treatment of derived elements (in UML derivation rules are expressed as an equality constraint between the value of the derived element and the value returned by the derivation rule). As further work, we would like to explore the applicability of our techniques to these related problems and the feasibility of (partially) porting our techniques to other similar modeling languages.

Moreover, we would like to study how to incorporate semantic query optimization results [23] to further improve efficiency of integrity checking in our approach and explore the use of tools for the formal analysis of OCL expressions (as [10] based on the Isabelle theorem prover and [20] based on Maude, a high-performance equational and rewriting logic language) to extend and analyze our set of transformation rules for OCL expressions.

References


56. Thalheim, B.: Achievements and Problems of Conceptual Modelling
   In: Proc. 1st Workshop on Active Conceptual Modeling of
   Learning (ACM-L'06), LNCS, 4512 (2006) 72-96
57. Wieringa, R.: A survey of structured and object-oriented software
   specification methods and techniques. ACM Computing
   Surveys 30 (4) (1998) 459-527