UNIVERSITAT POLITÈCNICA DE CATALUNYA

DEPARTAMENT DE LLENGUATGES I SISTEMES INFORMÀTICS

JORDI CABOT SAGRERA

INCREMENTAL INTEGRITY CHECKING IN UML/OCL CONCEPTUAL SCHEMAS

PhD. DISSERTATION

ADVISOR: Dr. ERNEST TENIENTE LÓPEZ

BARCELONA

2006

A dissertation presented by Jordi Cabot Sagrera in partial fulfillment of the requirements for the degree of *Doctor per la Universitat Politècnica de Catalunya*

A la Marta

A l'Enric, la Núria i la Neus

Acknowledgements

First and foremost, I would like to thank Ernest Teniente. With him I have learnt how to research and, most important, how to enjoy researching. Thanks Ernest for your scientific rigor, guidance, patience, complicity and friendship during our long discussions. You always let me find the right answer at my own pace, even when this took me some time.

I would also like to thank the examiners of the thesis board: Dr. Antoni Olivé, Dr. Piero Fraternali, Dr. Martin Gogolla, Dr. Óscar Pastor and Dra. Maria Ribera Sancho; for accepting to be members of this board and for the fruitful discussions on the thesis topics.

I am also indebted to many people who in some way have helped me to get here. Special thanks are due to Antoni Olivé. No matter his duties, he was always there whenever I needed him. I have benefited many times from his experience and advice. Thanks also to all my colleagues in the *Grup de Modelització Conceptual* (Anna, Cristina, Dolors, Jordi, Maria, Ruth and Xavier), in the *Secció de Sistemes d'Informació* and in the *Estudis d'Informàtica, Multimèdia i Telecomunicacions* for trusting me and giving me their best support during this thesis. I know that this has not always been an easy task. I also want to express my gratitude to Santi Ortego. He was my first mentor (I will never forget that) and, above all, he is a friend.

I am also very grateful to Piero Fraternali, who let me join the web engineering group at *Politecnico di Milano* for a five-month period. With him I learnt to take a different perspective on my own work. I had insightful discussions and exchanged ideas with many of the group members, especially with Marco Brambilla, Sara Comai, Giovanni Toffetti and Alessandro Bozzon.

Thanks also to Carol Cervelló and Raúl Solana. With your help, implementing the method presented in this thesis has been really hard. Without it, it would have been impossible.

Finally, I thank my family (Marta, Enric, Núria and Neus), who has always believed in me. Marta, your continuous support during all these years has been more important than you think. I have taken your strength, courage and determination whenever mine were missing.

This work has been partially supported by the *Ministerio de Ciencia y Tecnología* under project TIN2005-06053.

Abstract

Integrity constraints play a fundamental role in the definition of conceptual schemas (CSs) of information systems. An integrity constraint defines a condition that must be satisfied in each state of the information base (IB). Hence, the information system must guarantee that the state of the IB is always consistent with respect to the integrity constraints of the CS. This process is known as integrity checking. Unfortunately, current methods and tools do not provide adequate integrity checking mechanisms since most of them only admit some predefined types of constraints. Moreover, the few ones supporting a full expressivity in the constraint definition language present a lack of efficiency regarding the verification of the IB.

In this thesis, we propose a new method to deal with the incremental evaluation of the integrity constraints defined in a CS. We consider CSs specified in the UML with constraints defined as OCL invariants. We say that our method is incremental since it adapts some of the ideas of the well-known methods developed for incremental integrity checking in deductive and relational databases. The main goal of these incremental methods is to consider as few entities of the IB as possible during the evaluation of an integrity constraint. This is achieved in general by reasoning from the structural events that modify the contents of the IB. Our method is fully automatic and ensures an incremental evaluation of the integrity constraints regardless their concrete syntactic definition.

The main feature of our method is that it works at the conceptual level. That is, the result of our method is a standard CS. Thus, the method is not technology-dependent and, in contrast with previous approaches, our results can be used regardless the final technology platform selected to implement the CS. In fact, any code-generation method or tool able to generate code from a CS could be enhanced with our method to automatically generate incremental constraints, with only minor adaptations. Moreover, the efficiency of the generated constraints is comparable to the efficiency obtained by existing methods for relational and deductive databases.

Contents

1. Introduction

Since the very beginning of computer science, one of the main goals of software engineers has been to automate as much of the software development process as possible. In fact, the software engineering community envisages a future in which, of all the phases of software development, software engineers will only be strictly necessary during the specification of the information system while the remaining phases (mainly design, implementation and test) would be fully automated. The cost of software development could therefore be cut because these later phases easily involve well over half the total cost of a development or maintenance project.

The goal of automating information systems building was first stated in the late sixties [84] [85]. Additional proposals in this direction appeared in the 1970s [62] and the 1980s [94]. However, object-oriented methods truly bloomed in the late 1990s (see [92] for a detailed comparison and [79] for a particularly interesting example) when interest in the topic was revived. Recently, a number of new alternatives ([56], [74], [24], among many others) and standards [69] have emerged. Furthermore, code-generation capabilities of today's CASE tools (i.e. the ability of the tools to automate part of the design and implementation stages) are a key issue in their development and marketing strategy.

Obviously, we are closer to our goal now than we were forty years ago but several problems still remain to be addressed. In fact, this goal was recently classified as a grand challenge for information systems research [65]. [65] emphasized the central role of the conceptual schemas in the automatic development of information systems and presented a list of open problems that must be solved before this approach can be widely used in the development of industrial information systems. In conceptual modeling, a conceptual schema (CS) is the formal specification of functional requirements. CSs basically consist of a set of taxonomies of entity types and relationship types, also commonly referred to as classes and associations in object-oriented terminology. We refer to the representation of the state of the CS (the set of existing entities and relationships, also called objects and links in object-oriented terminology) in the information system as the information base (IB).

The list of open problems presented in [65] included the *enforcement of integrity constraints*. An integrity constraint states a condition that must be satisfied in each state of the IB. A complete CS must include the definition of all relevant integrity constraints [45].

Thus, most CSs contain a large number of constraints. The information system must enforce these constraints efficiently. This process is known as *integrity checking*.

In our context, this implies that a fully automatic method is required to generate, from all kinds of constraints present in the CS, the elements of the information system (for instance code fragments and/or data structures) resulting in such an efficient integrity checking. Currently, this method does not yet exist. As discussed in detail in Chapter 9, current proposals either only support a subset of all possible integrity constraints or (more commonly) the specified constraints are not taken into account during automatic code generation. Only a few proposals try to provide an efficient and automatic implementation of the constraints present in the CS ([90], [91], [74], [31]) but they achieve only partial results.

The main contribution of this thesis is to provide an automatic method for the incremental evaluation of integrity constraints in CSs, in particular, for CSs specified in UML (Unified Modeling Language [68]) with constraints specified in OCL (Object Constraint Language [67]). Since our method works with UML/OCL schemas, it is not tied to any particular technology. Moreover, this technology independence makes it possible to reuse its results when generating the system implementation in any technology platform.

We define our method for an automatic constraint generation from the CSs as an *incremental* method, since it adapts some ideas from incremental methods developed for deductive [41] and relational [23] databases to cope with the incremental checking of integrity constraints at the conceptual level. Given the basic assumption that the IB is in a consistent state prior to its modification, incremental methods exploit available information about the structural events applied during modifications of the IB to avoid a complete recomputation of the constraints (i.e. to avoid checking every time all entities restricted by the constraint). The modification of the IB may correspond to the execution of an information system operation or to the concept of transaction as in the database field.

A structural event can be defined as an elementary change in the population of an entity or relationship. The effect of each event depends on the type of the event. Each modeling language defines the possible event types. Examples of event types include: insert an entity in an entity type, delete an entity from an entity type, update an attribute, insert a relationship in a relationship type, etc.

Given this framework, the rest of the chapter is devoted to illustrate the problems that arise when dealing with an efficient checking of OCL constraints (Section 1.1), to provide an overall picture of the state of the art in integrity checking (Section 1.2), to present the goals and contributions of this thesis (Section 1.3) and to outline the organization of the chapters of this thesis (Section 1.4).

1.1 Problem description

Given a CS with a set of integrity constraints, the goal of our method is to ensure an incremental integrity checking of all constraints after the application of any arbitrary set of structural events over the IB. Moreover, we want the results of our method to be useful regardless the target platform used to implement the schema. Therefore, we need to provide such incremental checking at the conceptual schema level, i.e. in terms of a set of OCL expressions (constraints, derivation rules…) defined over the (processed) CS.

This section uses a running example to illustrate the main inefficiency problems resulting from a direct checking of the OCL constraints as defined by the OCL standard [67]. These problems are the ones our method overcomes when processing the original CS in order to provide an incremental checking of all integrity constraints.

1.1.1 Running example

As a running example throughout the thesis we will use the CS shown in Figure 1.1, meant to (partially) model a simple e-commerce application.

The CS contains information about the sales (*Sale* entity type) and the products (*Product* entity type) they contain. The reified relationship type *SaleLine* registers the number of products of the same type included in a given sale. Some of the products are classified as *RestrictedProducts* since they may not be available to every kind of customers (for instance, chemical products, pills…).

Sales can be split up into several shipments (*Shipment* entity type) and shipments can be reused to ship several sales. Finally, sales may be associated with registered customers (*Customer* entity type) who benefit from discounts depending on the category (*Category* entity type) they belong to.

Additionally, we define the following integrity constraints in the CS:

- *ValidShipDate*: All sales must be fully delivered within 30 days of the payment date.

- *CorrectProduct*: All products must have a *price* greater than zero and a *maxDiscount* of a 60% (which it is the maximum discount allowed in the company).

- *NotTooPendingSales:* The number of pending sales for a customer may not exceed the maximum pending amount permitted in his/her category.

- *AtLeastThreeCustomers:* At least three customers must be associated with each category.

- *NumberOfRestrictedProducts:* No more than 20 different types of restricted products may be on sale.

Some constraints can be expressed graphically in the CS, such as the constraint *AtLeastThreeCustomers*, which is expressed as a multiplicity constraint in the customer role of the *BelongsTo* relationship type. The others must be represented textually. In the next section we will express them as invariants written in OCL.

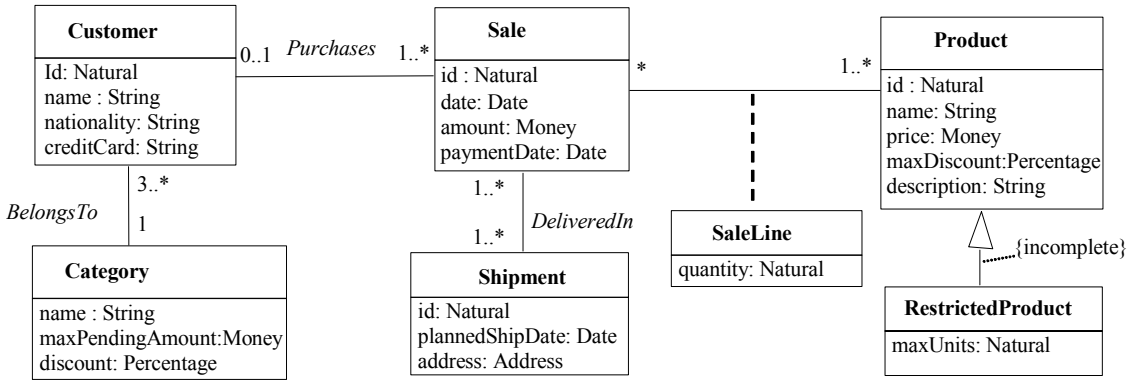


Figure 1.1. The CS of the e-commerce example

1.1.2 Integrity constraints in the OCL

Our method assumes that textual constraints are defined as invariants written in OCL. The use of a general-purpose (textual) sublanguage like OCL is required to be able to express all kinds of constraints since most constraints cannot be expressed using only the graphical constructs provided by the conceptual modeling language [33], UML in our case. At the time this thesis was written, the most recently adopted specification of the OCL standard was [67].

In OCL, invariants are defined in the context of a specific type (either an entity or relationship 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 (i.e. it evaluates to a boolean value) and 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.

Figure 1.2 shows the previous integrity constraints defined for the running example expressed in OCL. For instance, in *ValidShipDate*, *Sale* is the context type, the variable *self* refers to an entity of *Sale*, and the date comparison (the body) must hold for all possible values of *self* (i.e. all entities of *Sale*). Other constraints, such as *NumberOfRestrictedProducts,* require that the operator *allInstances* be used in their definition. *AllInstances* is a predefined feature on classes that returns the set of all instances of the type that exist at the specific time when the expression is evaluated.

```
context Sale inv ValidShipDate:
        self.shipment->forAll(sh| sh.plannedShipDate<=self.paymentDate+30)
context Product inv CorrectProduct:
        self.price>0 and self.maxDiscount<=60
context Category inv NotTooPendingSales:
        self.customer->forAll(c| c.sale->select(s| s.paymentDate>Time.now())->collect(sa|sa.amount)
        ->sum()<=self.maxPendingAmount)
context Category inv AtLeastThreeCustomers:
        self.customer->size()>=3
context RestrictedProduct inv NumberOfRestrictedProducts:
        RestrictedProduct.allInstances()->size()<=20
```

Figure 1.2. OCL constraints for the e-commerce example

Graphical constraints supported by the UML, like cardinality or disjointness constraints, can be transformed into a textual OCL representation, as shown in [39], and thus, they can also be handled by our method. As an example, Figure 1.2 includes the OCL representation of *AtLeastThreeCustomers*.

Since constraints must be satisfied in each state of the IB, a direct (naïve) checking of the constraints would involve: 1- checking all constraints after each modification of the IB (due to the application of a set of structural events over the IB), and 2 – for each constraint, evaluating the constraint body over all instances of the context type.

It is not difficult to see that this naïve strategy involves many irrelevant verifications. Therefore, an application generated using this default checking behavior would have a low run-time performance.

In particular, we have identified the following improvements to the default checking behavior of OCL integrity constraints:

1. Avoiding integrity checking of constraints that are irrelevant to the set of structural events applied over the IB. We define that an integrity constraint is irrelevant to a set of structural events if none of the changes brought about by the events may lead to the violation of the constraint.

2. Checking the constraints only over their relevant entities (those affected by the changes applied over the IB).

3. Generating a (possibly different) alternative representation of a constraint for each kind of event that may violate constraint. This new alternative will be specially suited to check the constraint after applying over the IB events of that kind. This is necessary since the high expressiveness of the OCL language permits to express the same constraint in a variety of forms, not all of them equally appropriate.

Next sections present these strategies rather informally. The subsequent chapters of this thesis contain a more formal description.

1.1.3 Determining the events that can violate an integrity constraint

It is not necessary to check all integrity constraints after each modification of the IB. For instance, the *CorrectProduct* constraint cannot be violated by events that insert a new sale, change the name of a customer, delete an existing product from the IB, etc. On the contrary, events updating the *price* or *maxDiscount* attributes of a product or inserting a new product can actually violate *CorrectProduct*. Therefore, *CorrectProduct* must only be verified after modifications of the IB that include events of one of these three types. Otherwise, we need not check it.

We call the structural event types that may violate a constraint the *potentially-violating structural events* (PSEs) for that constraint. At definition time, the PSEs for each constraint must be determined and compared at run-time with the particular events applied during the modification of the IB. When none of the applied events is an instance of a structural event type included in the set of PSEs for a constraint, that constraint can be discarded during the integrity checking process.

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

Although this strategy is better than a direct checking of the constraint, it is not yet precise enough since it would consider as PSEs events that will never violate the constraint. In other words, the set of structural event types provided by this naïve solution is a superset of the structural event types that may actually violate the constraint. For instance, using this approach we would determine that eight types of structural events may violate the constraint *ValidShipDate*: insert / delete / update of the entity type *Sale*, insert / delete of the relationship type *DeliveredIn* and insert / delete / update of the entity type *Shipment*. Nevertheless, only the modifications of sales (in particular, the update of the *paymentDate* attribute) and shipments (update of the *plannedShipDate* attribute) and the insertion of a new *DeliveredIn* relationship may actually violate the constraint. The other five events can never violate *ValidShipDate*. For instance, the insertion of a new *Sale* alone cannot possibly violate the constraint. Only when the sale is assigned to a shipment (insertion of a new *DeliveredIn* relationship) a violation may occur.

In order to precisely determine the PSEs of a constraint, we must consider both the set of elements referenced in the constraint and the context in which these elements are referenced (understood as the elements and OCL operations surrounding them). For instance, the PSEs that may violate *AtLeastThreeCustomers* constraint differ when, instead

of enforcing that all categories must have at least three customers, we want to enforce that all categories have less than three customers, even though in both cases we refer the same model elements (*Category*, *Customer* and the relationship type *BelongsTo* between them)

1.1.4 Evaluating an integrity constraint over the relevant instances

When checking an integrity constraint, it is extremely inefficient to evaluate the body of the constraint over all instances of the context type. Assuming, as usual, that the IB was in a consistent state prior to its modification, only the set of instances affected by the issued structural events can violate the constraint.

For instance, consider the following scenario. The initial state of a given IB is shown in the partial schema of Figure 1.3, in which we represent the OIDs (object identifiers) of an imaginary population for each entity type (i.e. s_1,s_2,… are instances of *Sale*). The population of the relationship type *DeliveredIn* is represented by the OIDs of the participant entity types (for example, the relationship $<s_1,sh_1>$ indicates that the sale s_1 is delivered in the shipment sh_1).

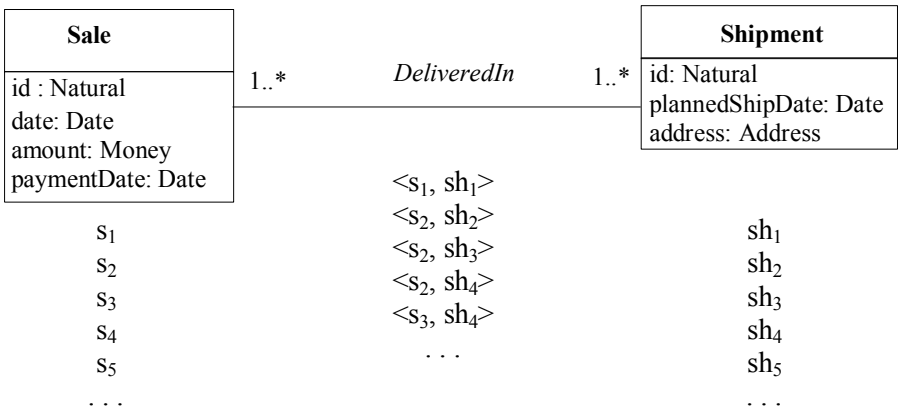


Figure 1.3. Initial state of the IB

Assume that over this initial state we apply the following structural events: 1 - update of the *paymentDate* of s_1, 2 - update of the *plannedShipDate* of sh_2 and 3 - insertion of a new relationship between s_3 and sh_5. After applying these events, *ValidShipDate* is one of the constraints that must be checked (all the issued events are PSEs for this constraint). However, given this set of events the only entities of *Sale* that must be checked (in order to ensure that *ValidShipDate* still holds) are s_1 (because of the attribute update), s_2 (due to its relationship with the modified shipment sh_2) and s_3 (due to its participation in the new relationship with sh_5). The constraint condition need not be evaluated over any other sale since none have changed during the modification of the IB.

Note that the computation of the relevant instances for a constraint must consider not only the entities of the context type directly modified by the structural events but also the entities related, directly or indirectly, to certain modified entities of other types.

1.1.5 Generating the best definition of a constraint with respect to a structural event

Due to the high expressiveness of the OCL, designers have different syntactic possibilities for defining each integrity constraint. For example, the constraint *ValidShipDate* could also be expressed using *Shipment* as a context type:

> ***context*** *Shipment* ***inv*** *ValidShipDate':*
> *self.sale->forAll(s| self.plannedshipDate <= s.paymentDate+30)*

Both representations (the initial *ValidShipDate* and *ValidShipDate'*) are correct and semantically equivalent. Of all the possible alternatives, the designer chooses one at definition time to represent the constraint (we do not assume that the designer uses any particular criterion). However, the selected representation may not be the most appropriate to check the constraint after the application of events instance of some (or all) of the structural event types that are PSEs for the constraint.

As an example, consider again the scenario introduced in the previous section where we drew that to check *ValidShipDate* only sales s_1, s_2 and s_3 had to be considered. Using the original definition of *ValidShipDate* (see Figure 1.2), the evaluation of the constraint body over those sales implies evaluating the following OCL expressions (where we instantiate the *self* variable with the three relevant sales):

$$exp \equiv s_1.shipment->forAll(sh|\ sh.plannedShipDate<=s_1.paymentDate+30)$$
$$exp_2 \equiv s_2.shipment->forAll(sh|\ sh.plannedShipDate<=s_2.paymentDate+30)$$
$$exp_3 \equiv s_3.shipment->forAll(sh|\ sh.plannedShipDate<=s_3.paymentDate+30)$$

If all three expressions return a true value, we may conclude that the IB is still consistent regarding *ValidShipDate*.

Nevertheless, the integrity checking of *ValidShipDate* using these three expressions cannot be considered incremental yet. Some of the verifications involved in the expressions are still irrelevant since we compare the payment date of each sale with all the planned dates of the related shipments. In fact, this is only required for s_1 where we must ensure that the new payment date is coherent with all related shipments. However, for s_2 we just need to compare its date with the new planned date of shipment sh_2, it is not necessary to compare the payment date of s_2 with the planned date of sh_3 or sh_4 since they have not been changed. Similarly, the payment date of s_3 only needs to be compared with the new assigned shipment sh_5.

These irrelevant checks are performed because *ValidShipDate* (as represented in Figure 1.2) it is not an appropriate syntactic representation for a direct verification of the state of the IB after the event types *updates of the Shipment entity type* or *inserts in the DeliveredIn relationship type*. Instead, after *Shipment* updates, we should use the alternative *ValidShipDate'* proposed above. Similarly, after insertions in *DeliveredIn* we should use a

third alternative, using *DeliveredIn* as the context type. By using the appropriate alternative after each applied event, we obtain an incremental checking of the original constraint. For some constraints, several appropriate definitions may exist with respect to some (or all) of their PSEs.

1.2 State of the art

The problem of efficient integrity checking has been widely addressed in the fields of deductive and relational databases as part of the solution to integrity checking or materialized view maintenance problems. These methods are predecessors to our method, which reuses some of their ideas. However, the differences between the expressivity of these methods and that of the UML/OCL make it impossible to directly apply them to the integrity checking problem in UML/OCL CSs.

Moreover, we have recently witnessed a growing number of methods that follow the MDD (Model-driven development [78]) and MDA (Model-driven architecture [69]) approaches. Both approaches give the CS a central role in the development process and promote the automatic generation of the system implementation based on its CS, either directly or by first transforming the CS into a new model adapted to the specific features and characteristics of the target platform. In the latter case, the initial CS is referred to as the PIM (platform-independent model) while the specific model is called the PSM (platform-specific model). Unfortunately, support for the generation of integrity constraints in existing MDD and MDA methods and tools is quite unsatisfactory [19]. In fact, these methods are unable to generate an implementation of the constraints that efficiently ensures a valid IB state and/or lack expressivity in the supported constraint definition language.

Therefore, no method can yet provide an incremental checking of the constraints (as methods for relational and deductive databases do) based on the constraints defined in the CS and specified using a highly expressive language like OCL.

In the following sections we introduce the most representative proposals from each field. We distinguish between methods for relational and deductive databases although the distinction is somewhat artificial since a relational database can be regarded as a type of deductive database.

Chapter 9 describes these methods in more detail and compares their results with those of our method.

1.2.1 Approaches for relational databases

[21] is, probably, the most prominent proposal in the field of relational databases. In this method, constraints are defined as predicates over the database state using the SQL

language. Predicates are defined such that if a predicate is true in a particular state then the constraint is violated. These kinds of predicates are known as *inconsistency predicates*.

For each constraint, this method creates a production rule (i.e. a trigger) to detect the constraint violation. The production rule is fired whenever a structural event that may violate the constraint is applied over the database. The firing of the rule starts the evaluation of the corresponding predicate. If the predicate is satisfied (meaning that the constraint has been violated by the event) the user may define which action the system should take. As an example, Figure 1.4 shows the production rule for the constraint *CorrectProduct.* When one of the PSEs for *CorrectProduct* is applied, the corresponding production rule checks that no exist a product with a wrong value in the *price* and/or *maxDiscount* attributes.

The constraint definition language is expressive enough (it admits negation, aggregate operators, bag semantics, etc) but this approach lacks of precision, since, when determining the events that can induce a constraint violation, the result is a superset of the set of events that can actually violate the constraint.

Furthermore, its production rules are unable to check all constraints incrementally. Depending on the constraint complexity, the rules may need to examine the whole table instead of just the modified tuples.

In a later work [23], the method was improved to incrementally check all constraints but, as a trade-off, the constraint definition language is restricted with respect to [21] (for instance, no aggregate operators can be used).

```
CREATE RULE CorrectProduct
WHEN inserted into product,
     updated product.price, updated product.maxDiscount
IF exists Product:( select *
                    From product: (inserted product
                              union new updated product.price
                              union new updated product.maxDiscount)
                    Where price<=0 or maxDiscount>60)
THEN <ACTION>
```

Figure 1.4. Production rule for *CorrectProduct*

1.2.2 Approaches for deductive databases

The deductive database field boasts a long tradition of methods devoted to the problem of integrity constraint checking ([42], [22], [88], [82] among others). Constraints are defined in first-order logic. Only few methods also provide constructs to specify more complex constraints (as aggregate operators or bag semantics). Support for all these constructs is required in our context since these constructs frequently appear in the definition of OCL constraints. See [41] for a survey of these deductive methods and a general discussion of their limitations.

Some of these approaches do not focus on the problem of integrity checking itself but rather on the related problem of materialized view maintenance. A materialized view is a view whose tuples are stored in the database instead of being recomputed every time the view is queried. Then, the materialized view problem deals with incrementally updating the view data in response to changes in the underlying tables in the view definition.

Integrity constraints may be expressed as inconsistency predicates in deductive databases. With this representation, constraints are expressed as views that must be empty (a non-empty view indicates that the corresponding integrity constraint has been violated). The query of the view corresponds to the constraint body in denial form (i.e. the view selects the tuples that do not satisfy the constraint body). Therefore, the integrity checking problem can be regarded as a subset of the view maintenance problem.

These approaches share a similar core mechanism. They all represent integrity constraints as inconsistency predicates. They then propose a set of rules to control the insertions over the predicate representing the constraint. Each rule identifies a situation that could possibly induce a constraint violation. Whenever one of the rules is found to be true, the constraint is considered violated. Chapter 9 discusses how the methods differ in terms of the precision and efficiency of the rules they propose.

As a simple example, the inconsistency predicate representing the *CorrectProduct* constraint is:

$$Ic_{CorrectProduct} \leftarrow Product(id, name, price, maxDiscount, description) \wedge (price<=0 \vee maxDiscount>60)$$

where $Ic_{CorrectProduct}$ contains those products with a *price* value lesser than one or a *maxDiscount* value greater than 60.

Given this predicate, the rules generated by the previous methods would be:

$$Ic_{CorrectProduct} \leftarrow iProduct(id, name, price, maxDiscount, description) \wedge (price<=0 \vee maxDiscount>60)$$

$$Ic_{CorrectProduct} \leftarrow uProduct(id, name, price, maxDiscount, description) \wedge (price<=0 \vee maxDiscount>60)$$

where the predicate *iProduct* registers products inserted during the transaction and *uProduct* products that have been modified (perhaps resulting in new values for the *price* or *maxDiscount* attributes that violate the constraint).

In fact, updates over attributes are only considered explicitly by [88]. The other approaches model updates as deletions followed by insertions, resulting in a loss of precision (and efficiency) when processing integrity constraints. We find a similar problem when dealing

with events to add (remove) types to an existing entity, neither supported by these methods.

1.2.3 Code-generation methods and tools

Of all the current tools and methods that use MDA and MDD approaches (see, for example, [56],[24], [74], [31], [12], [48]) and are devoted to the automatic code generation of an application from its CS, few consider the definition and/or generation of integrity constraints [19].

The differences between them lie in how they decide *when* a constraint needs to be checked and how many entities they take into account each time the constraint is checked. Some of them verify all constraint after each structural event. A few ones consider the type of the applied events when deciding whether to check a constraint. In particular, they determine that an event may induce the violation of a constraint if the event modifies one of the elements referenced in the constraint body, but without considering whether the kind of change can really induce its violation (i.e. some of the events they take into account are irrelevant to the constraint). After one of the relevant events is applied, most methods lack of precision when computing the set of entities of the context type that need to be evaluated (some methods compute a superset of the really affected instances while others compute a subset of them).

Moreover, all methods depart from the integrity constraints exactly as defined by the designer. Thus, their efficiency depends on the concrete syntactic representation of the constraint.

1.3 Objectives and contributions of this thesis

As discussed above, current code-generation methods fall short when dealing with the code generation of integrity constraints defined in a CS. This thesis proposes a new method to cope with the incremental integrity checking of OCL constraints specified in UML CSs. By dealing with this problem at a conceptual (i.e. platform-independent) level, our method is not bound to any technological assumption. Its results can be reused regardless of the final technology platform in which the CS is going to be implemented.

Given an initial *CS*, the result of our method is a standard conceptual schema *CS'* that, when executed or directly implemented in a particular technology platform, is able to check all constraints incrementally. Then, given a code-generation method *M* able to implement the conceptual schema *CS* in a technology platform *P*, when *M* uses the conceptual schema *CS'* instead of *CS*, the automatic generation of *CS'* in *P* results in an implementation of the schema that checks all constraints incrementally. Note that *M* does not need to be modified to benefit from *CS'*. The results of our work can be helpful in any platform P_2 provided that a code-generation method M_2 exists for P_2.

Additionally, our method meets the following subgoals:

1. It takes into account the different kinds of constraints that appear in the CS. Depending on their complexity, we can distinguish three levels of integrity constraints (adapted from [87]):

   1. Intra-entity constraints: Constraints that restrict the values of the attributes of a single entity.
   2. Inter-entity constraints: Constraints that restrict the relationships between an entity and other entities that are instances of different entity types
   3. Type constraints: Constraints restricting the relationship between a set of entities that are instances of the same entity type (for instance, a constraint stating that all entities of a given type must have a different value in a given attribute)

   For each kind of constraint, including constraints that combine more than one level, our method produces an efficient result (different techniques are required to handle the different kinds of constraints).

2. The efficiency of the incremental integrity checking obtained with our method is comparable to that of incremental methods for deductive and relational databases.

3. The method is feasible, in the sense that it can be integrated with other code-generation strategies to generate an implementation of the resulting *CS'* in the most popular technology platforms.

An implementation of this method is available at [16].

1.4 Thesis structure

This thesis is structured as follows. The next chapter presents an overview of our method. The various steps of the method are presented in Chapters 3 through 6. Chapter 7 then presents the architecture of a tool that implements the method. Chapter 8 presents the transformation of the resulting CS into a relational database to show the feasibility of our method. Chapter 9 discusses related work, including a comparison of the results of our method with those of incremental methods for databases and current code-generation tools. Finally, Chapter 10 presents some conclusions and related work.

Some of the results of this thesis have been already published in [14], [15], [17], [18] and [19].

2. Method Overview

This chapter provides an overall picture of our proposal for dealing with incremental integrity checking in conceptual schemas. Given an initial conceptual schema *CS*, the result of our method is another conceptual schema *CS'* that, when executed or directly implemented in a particular technology platform, is able to check all constraints incrementally.

All CASE tools can benefit from our method if, once the designer has defined the conceptual schema *CS*, the tool uses our method to obtain *CS'* and then departs from *CS'* to generate the application code and data structures (Figure 2.1).

Figure 2.1. Application scenario for our method

CS' is obtained by means of a sequence of transformation steps over the constraints appearing in *CS*. Some of these transformations also involve the addition of some new entity and relationship types to *CS*. The number of new entity and relationship types is linearly proportional to the number of constraints. All existing entity and relationship types of *CS* remain unchanged.

More precisely, the rationale of our method is to replace each integrity constraint *ic* defined in the *CS* with a set of equivalent integrity constraints $set_{ic'}$, in which each $ic' \in set_{ic'}$ leads to an incremental evaluation with regards to some of the structural events that may violate the original constraint. Our method assumes that the IB is updated at run-time with the modifications produced by the applied structural events. Therefore, when verifying the constraints, the state of the IB already reflects the changes induced by the events.

Our method can be formalized according to the steps depicted in Figure 2.2. It consists of three main steps (steps 1-3) plus a preliminary step (step 0). Each step tackles one of the efficiency problems described in the previous chapter.

Each step is briefly outlined below. The following chapters address each step in detail. As an example, we show how the CS in Figure 1.1 and the constraint *ValidShipDate* are processed in each step to obtain an incremental verification for *ValidShipDate* at the end of step 3.

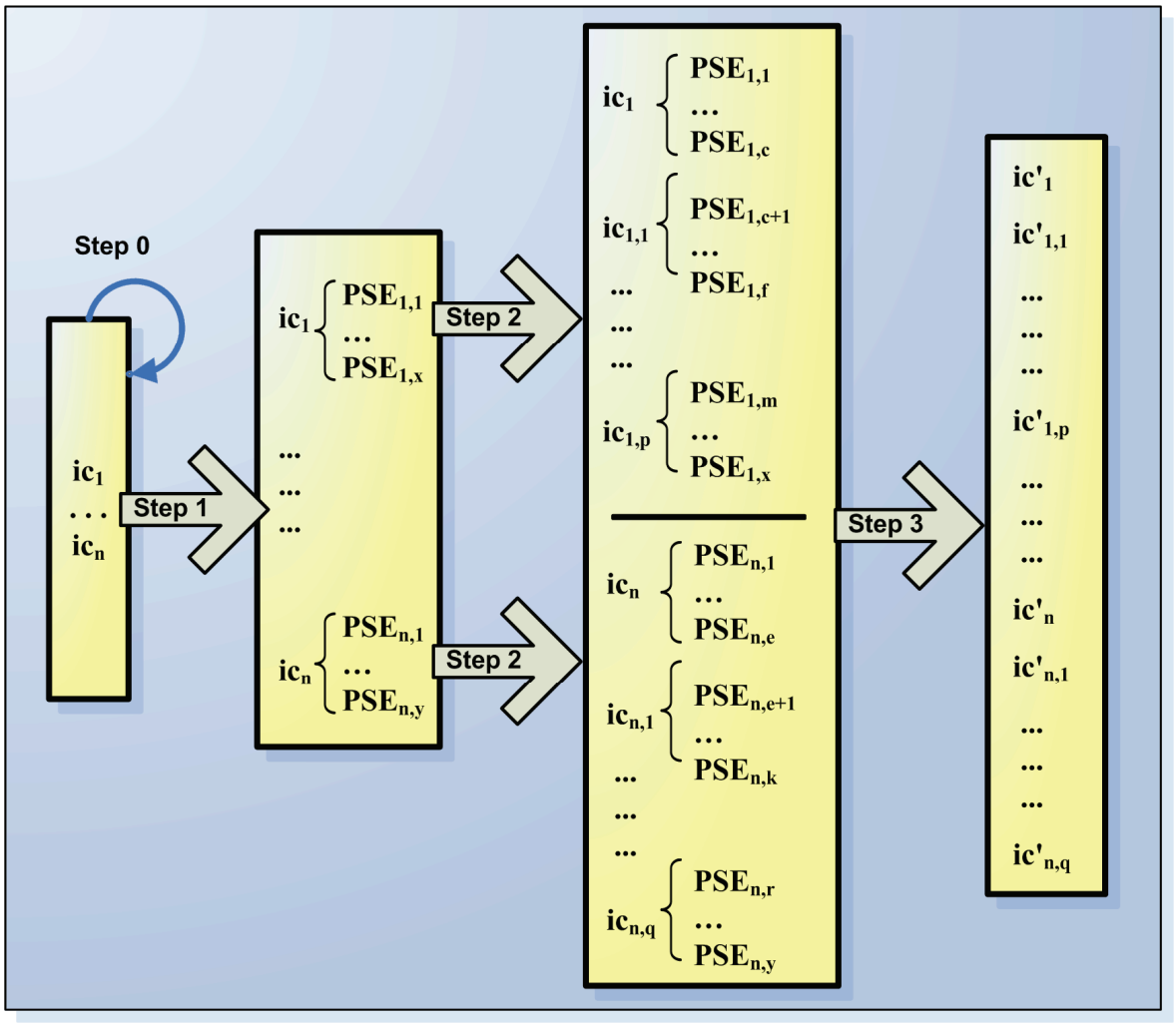


Figure 2.2. General schema of our method

Step 0: Simplification of the original constraints

To facilitate the definition of the different steps, our method assumes that the body of each integrity constraint is expressed in a simplified form. In this preliminary step (step 0), this simplified representation is automatically obtained from the original body expression. This simplification process does not entail a loss of expressive power in the constraints we may deal with.

Roughly, the simplification process reduces the expressivity of the OCL expressions that form the body of each constraint by means of applying several transformation rules that replace some of the OCL operators in the constraint body with equivalent (more basic) ones. As an example, the rule *X->reject(Y)* → *X->select(not Y)* removes the *reject* operator (left part of the rule) from OCL expressions and replaces it with the *select* operator (according to the right part of the rule).

The constraint *ValidShipDate*:

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

is already expressed in a simplified form. Thus, it is not modified in this step.

Step 1: Determining the potentially-violating structural events

In step 1, our method associates to each constraint of the CS a set of potentially-violating structural events. The PSEs are drawn from the syntactical definition of the constraint. In general, each constraint presents a different set of PSEs.

In particular, for *ValidShipDate* we would obtain the following PSEs:

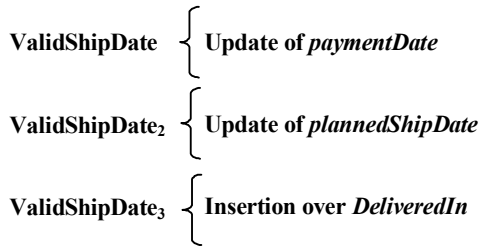
1. Update of the attribute *paymentDate* defined in the type *Sale*
2. Update of the attribute *plannedshipDate* defined in the type *Shipment*
3. Insertion of a new relationship in the relationship type *DeliveredIn*

Step 2: Obtaining an appropriate syntactic representation for each constraint

For each integrity constraint ic_i and event type *ev*, *ev* $\in$ set of PSEs of ic_i, this step determines an appropriate alternative syntactic representation $ic_{i,j}$ of ic_i with respect to *ev* (i.e. an alternative whose verification after the application of structural events of type *ev* yields to an incremental checking of ic_i). Note that $ic_{i,j}$ may be an appropriate alternative for several PSEs and that for some PSEs the original ic_i representation may already be the suited one.

Given the previous *ValidShipDate* constraint, at the end of this step we would obtain that the original *ValidShipDate* representation (the one defined over the context type *Sale*) is an appropriate representation to check the constraint after modifications of the *paymentDate* attribute. However, after modifications of the *plannedShipDate* attribute we need to generate an alternative representation ($ValidShipDate_2$) of *ValidShipDate* using *Shipment* as context type as well as another alternative ($ValidShipDate_3$) for assignments of shipments to sales using *DeliveredIn* as context type.

Figure 2.3 shows the different versions of *ValidShipDate* and the PSEs associated to each one. Each version will be used to verify its own set of PSEs.



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

context Shipment **inv** ValidShipDate$_2$: self.sale->forAll(s| self.plannedShipDate<=s.paymentDate+30)

context DeliveredIn **inv** ValidShipDate$_3$: self.shipment.plannedShipDate<=self.sale.paymentDate+30

Figure 2.3. Alternative representations for *ValidShipDate*

Step 3: Redefining the constraints to evaluate over the relevant instances

Finally, each constraint resulting from step 2 is redefined to be evaluated only over the instances of its context type affected by events instance of the event types included in its particular subset of PSEs (i.e. those event types for which the constraint is selected as an appropriate representation).

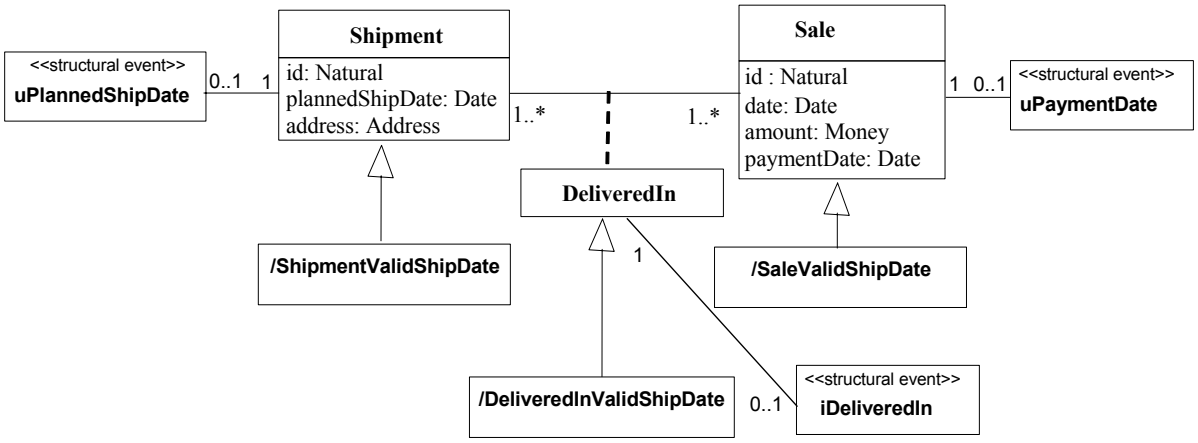
This means that, in the previous example, *ValidShipDate* should only be evaluated over the sales that have changed the value of their *paymentDate* attribute, *ValidShipDate$_2$* should only be evaluated over shipments where the value of *plannedShipDate* has been modified while *ValidShipDate$_3$* should only be evaluated only over new assignments between sales and shipments (i.e. over new relationships in the relationship type *DeliveredIn*).

This redefinition requires the addition of several new entity and relationship types to the original conceptual schema. As an example, Figure 2.4 shows the main aspects of the CS' resulting from processing the CS of Figure 1.1 to ensure incremental integrity checking of *ValidShipDate*. The basic idea is that each version of *ValidShipDate* is redefined using as new context type the new derived subtypes *SaleValidShipDate* (new context type for *ValidShipDate*), *ShipmentValidShipDate* (new context type for *ValidShipDate$_2$*) and *DeliveredInValidShipDate* (new context type for *ValidShipDate$_3$*). The definition of *DeliveredInValidShipDate* requires reifying the *DeliveredIn* relationship type.

Therefore, now the constraints are no longer verified over the whole population of *Sale*, *Shipment* and *DeliveredIn*, respectively, but rather over the population of the derived subtypes. Next, we must ensure that the population of these subtypes is exactly the set of entities that require verification, so that incremental checking of the constraint can be obtained.

The population of the derived subtypes is defined by means of the corresponding derivation rule (specified as a redefinition of the *allInstances* operation, according to the

method proposed in [64]). In the example, the derivation rule for *ShipmentValidShipDate* states that the population of the subtype is the set of shipments for which the value of the *plannedShipDate* attribute has changed (the information about updated shipments is recorded in the new *uPlannedShipDate* entity type). Since the context type of $ValidShipDate_2$ is now *ShipmentValidShipDate* rather than *Shipment*, the verification of $ValidShipDate_2$ becomes incremental because only the updated shipments are considered when checking the constraint. A similar reasoning is followed for the constraints *ValidShipDate* and $ValidShipDate_3$.



-- The derivation rules
context ShipmentValidShipDate::allInstances() : Set(Shipment)
body: uPlannedShipDate.allInstances().shipment

context SaleValidShipDate::allInstances() : Set(Sale)
body: uPaymentDate.allInstances().sale

context DeliveredInValidShipDate::allInstances() : Set(DeliveredIn)
body: iDeliveredIn.allInstances().deliveredIn

-- The redefined constraints
context SaleValidShipDate **inv** ValidShipDate:
self.shipment->forAll(sh| sh.plannedShipDate<=self.paymentDate+30)

context ShipmentValidShipDate **inv** $ValidShipDate_2$:
self.sale->forAll(s| self.plannedShipDate<=s.paymentDate+30)

context DeliveredInValidShipDate **inv** $ValidShipDate_3$:
self.shipment.plannedShipDate<=self.sale.paymentDate+30

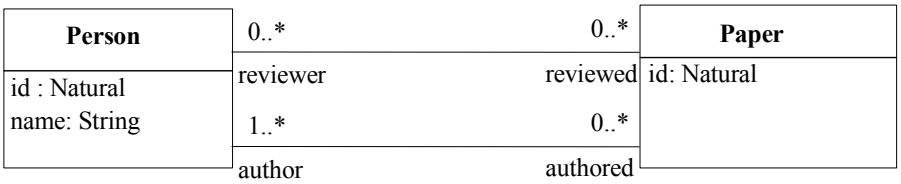
Figure 2.4. Final conceptual schema

We would like to remark that the quality of the results of our method does not depend on the particular representation of the original constraint. In the vast majority of cases, our method will always return the same incremental redefinition of the constraint regardless of its syntactic representation. For some constraints, the result of the redefinition process may differ slightly depending on the original representation. Nevertheless, the various

alternatives are equally suitable for verifying the constraint. Therefore, it does not matter which particular definition of the constraint is originally provided by the designer. In any case, we will obtain an incremental redefinition of the original constraint for each type of structural event that may violate it.

Two alternatives are equally suitable for checking a constraint after a structural event when they have the same level of complexity (i.e. the number of entities involved in their verification is similar). At the design stage, we do not know the population of the entity and relationship types of the schema. The level of complexity is therefore calculated by means of abstract expressions that represent the number of entities taken into account in each alternative. Thus, we may determine that an expression that requires to evaluate $X*Y$ entities is worse than another that requires accessing $1*Y$ entities. However, we may not know whether an expression e_1, involving X entities, is better than an expression e_2 involving Y entities. In this case, we assume that e_1 and e_2 present the same level of complexity.

As an example consider the CS of Figure 2.5 with two possible alternative representations for the constraint *AuthorNotReviewer* stating that a person cannot be an author and a reviewer of the same paper. Given the first alternative, our method would determine that, after the assignment of a reviewer *r* to a paper *p*, it is necessary to check that *p* is not one of the papers written by *r*. Given the second alternative, it would determine that we must check that *r* is not one of the authors of *p*. Therefore, the result is not exactly the same but for both alternatives our method gets the same efficiency level. At design time we cannot possibly know if, after this insertion event, it is better to check the papers of a reviewer or the authors of a paper, since this depends on the population of the IB at run-time.



context Person **inv** AuthorNotReviewer1:
self.reviewed->forAll(p:Paper| self.authored->excludes(p))
context Paper **inv** AuthorNotReviewer2:
self.reviewer->forAll(p:Person| self.author->excludes(p))

3. Simplifying the original OCL expressions

Due to the high expressiveness of the OCL, the designer has different syntactic possibilities for defining each integrity constraint. For instance, even the simple *CorrectProduct* constraint (Figure 1.2):

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

could be defined as complex as:

***context** Product **inv** CorrectProduct':*
not Product.allInstances()->exists(price<=0 or maxDiscount>60)

Although both representations have exactly the same meaning (both state that the price of all products must be greater than zero and the discount lower or equal to sixty), it is clearly easier to handle the first alternative than the second one since its definition does not require the *allInstances* operation nor the *exists* iterator. Nevertheless, designers are free to define the constraint as they desire so they could opt for defining *CorrectProduct* using the second alternative (or any other alternative).

The aim of this chapter is to provide a set of transformation rules to simplify the constraint definition. The constraint is simplified in the sense that the transformation rules reduce the number of different OCL operators (or their possible combinations) appearing in the constraint body. This does not necessarily imply that the resulting definition is shorter or easier to understand. We call the obtained representation the *simplified form* of the constraint. This simplified form is automatically obtained from the initial constraint definition as provided by the designer.

This simplification is helpful to facilitate the definition of the next steps of the method since they do not need to address the full expressivity of the OCL. In particular, they can avoid dealing with the kinds of OCL expressions simplified by the transformation rules. Therefore, the next steps always assume that constraints are expressed in their simplified form.

To generate the simplified form, we modify the body of the constraint but we do not consider the possibility of rewriting the constraint using a different type as context type. The reason is that, as we discussed in the previous chapters, there is not a single *best* context type for a constraint since depending on each event we may require a different

context type. The rules for the redefinition of a constraint over a different context type are addressed when coping with step 2 of the method (Chapter 5).

In this chapter, we first describe a set of general rules to simplify the number of different OCL operators (and their possible combinations) appearing in the constraint body (Section 3.1). Then, we focus on two especially useful sets of rules: rules to remove the *allInstances* operation (Section 3.2) and rules to transform an OCL expression into conjunctive normal form (Section 3.3). The application of these rules permits to obtain the original representation of *CorrectProduct* from the more complex representation *CorrectProduct'*. Except for rules of section 3.2, specific for integrity constraints, the rest may be applied to any OCL expression, including derivation rules and operation pre and postconditions.

Before applying the rules, each integrity constraint is unfolded. We say that a constraint is unfolded when all references to derived elements, query operations and variables resulting from let expressions are replaced with their definition. We restrict recursive derived elements to be unfolded just once. Additionally, all implicit variables are made explicit. This affects specially the *self* variable and the implicit variables used in iterator expressions. As a consequence, the previous *CorrectProduct'* is slightly modified by adding an explicit iterator variable *p* inside the *exists* iterator:

> ***context*** *Product* ***inv*** *CorrectProduct':* *not Product.allInstances()->exists(p| p.price<=0 or p.maxDiscount>60)*

3.1 Basic rules

Tables 3.1-3.3 present a list of basic simplification rules. Most of these rules are based on the equivalences defined in the OCL standard [67] itself. Some of the rules have also been proposed in [28],[38].

We group the equivalences by the type of expressions they affect (boolean, collection or iterator expressions). In the rules, the capital letters *X, Y* and *Z* represent arbitrary OCL expressions of the appropriate type (as required by the rule definition). The letter *o* represents an arbitrary object. The expression $r_1 \ldots r_n$ represents a (possibly empty) sequence of navigations.

Note that some rules reduce the number of different operations that can appear in an OCL expression (for instance, the rule *X->notEmpty()* $\rightarrow$ *X->size()>0* allows to avoid using the *notEmpty* operator) while others limit the possible combinations between different operators (for instance, *X->select(Y)->forAll(Z)* $\rightarrow$ *X-> forAll(Y implies Z)* simplifies the *select* iterator when placed before a *forAll*). The list does not pretend to be completely exhaustive but to include all rules that facilitate the processing of the integrity constraints in the next steps of the method.

Table 3.1 List of simplifications for boolean operators

X <> Y → not X = Y	X = true → X
X = false → not X	not false → true
not true → false	X and false → false
X and true → X	X or false → X
X or true → true	X>Y and X<=Y → false
X>Y or X<=Y → true	X>Y or X<Y → X<>Y
not X>=Y → X<Y	not X<Y → X>=Y
not X<=Y → X>Y	not X>Y → X<=Y
X=Y → (X and Y) or (not X and not Y) -- when X and Y are boolean expressions	not X=0 → X>0 -- when X evaluates to a natural type
X->size()<=0 or X->forAll(Y) → X->forAll(Y)	

Table 3.2 Simplifications for collection operators

X->includes(o) → X->count(o)>0	X->excludes(o) → X->count(o)=0
X->includesAll(Y) → Y->forAll(y1|X->includes(y1))	X->excludesAll(Y) → Y->forAll(y1| X->excludes(y1))
X->isEmpty() → X->size()=0	X->notEmpty() → X->size()>0
not X->isEmpty() → X->notEmpty()	not X->notEmpty() → X->isEmpty()
X->excluding(o) → X->–(Collection{o})	X->including(o) → X->union(Collection{o})
X->union(Y).r_1…r_n->forAll(Z) → X.r_1…r_n->forAll(Z) and Y.r_1…r_n ->forAll(Z)	X=Y → X->includesAll(Y) and Y->includesAll(X) -- when X and Y are collections of objects
X->last()→ X->at(X->size())	X->first()→ X->at(1)

Table 3.3 Simplifications for iterator expressions

X->exists(Y) → X->select(Y)->size()>0	not X->exists(Y) → X->forAll(not Y)
not X->forAll(Y) → X->exists(not Y)	X->reject(Y) → X->select(not Y)
X->select(Y)->size()=0 → X->forAll(not Y)	X->select(Y)->size()=X->size() → X->forAll(Y)
X->select(Y)->forAll(Z) → X->forAll(Y implies Z)	X->select(Y)->exists(Z) → X->exists(Y and Z)
X->one(Y) → X->select(Y)->size()=1	X->any(Y) → X->select(Y)->asSequence()->first()
$X.r_{1...}r_n.Y.attr.Z \rightarrow X.r_{1...}r_n.Y$->collect(attr).Z -- where *attr* represents an arbitrary attribute and at least a r_i has a multiplicity > 1	X->isUnique(Y) → X->forAll(x_1,x_2 | x_1<>x_2 implies x_1.Y <> x_2.Y)
X->forAll(Y) and X->forAll(Z) → X->forAll(Y and Z)	X->forAll(v| Y [and|or] X->forAll(v_2| Z)) → X->forAll(v,v_2| Y [and|or] Z)

With this set of rules we could partially simplify *CorrectProduct'* obtaining as a result the following alternative representation:

> ***context*** *Product* ***inv*** *CorrectProduct'': Product.allInstances()->forAll(p| not (p.price<=0 or p.maxDiscount>60))*

The next subsections complete the simplification of *CorrectProduct'* until we get its complete simplified form (the one already used in the original definition of *CorrectProduct*).

3.2 Removing the *allInstances* operation

AllInstances is a predefined feature on classes that gives as a result the set of all instances of the type that exists at the specific time when the expression is evaluated. For instance, the previous *CorrectProduct''* constraint states that all products (i.e. all instances of the *Product* entity type) must verify the constraint condition.

Nevertheless, some integrity constraints specified by means of the *allInstances* operator could also be specified using the variable *self* that represents an arbitrary instance of the context type. For instance, *CorrectProduct''* could also be specified as:

context *Product* ***inv*** *CorrectProduct''':*
not (self.price<=0 or self.maxDiscount>60)

Since constraints are assumed to be true for all instances of the context type (i.e. for all possible values of the *self* variable), both representations are equivalent. Moreover, *CorrectProduct'''* is clearly simpler than *CorrectProduct''*.

We propose two rules to remove the *allInstances* operation. They are applicable when the type over which *allInstances* is applied coincides with the context type of the constraint. They may not be applied if the constraint already contains any explicit or implicit reference to the *self* variable.

- *cet.allInstances()->forAll(v|Y)* $\leftrightarrow$ *Y'*, where *Y'* is obtained by replacing all occurrences of *v* (the iterator variable) in *Y* with *self*. As an example, see the previous *CorrectProduct'''* constraint.
- *cet.allInstances()->forAll(v_1,v_2..v_n| Y)* $\leftrightarrow$ *cet.allInstances()->forAll(v_2..v_n|Y')* where *Y'* is obtained by means of replacing all the occurrences of v_1 in *Y* with *self*. In this case the *allInstances* operation is not completely removed but, at least, the simplified expression is specified using the *self* variable which permits a more efficient treatment of the constraint in the next steps of the method.

  As an example, consider a constraint *UniqueName* (*context Product inv: Product.allInstances()->forAll(p_1,p_2 | p_1<>p_2 implies p_1.name<>p_2.name*), where we compare each pair of entities of *Product*. With this rule we replace p_1 with *self*, thus obtaining *Product.allInstances()->forAll(p_2|self<>p_2 implies self.name<>p_2.name)*.

3.3 Transforming an OCL expression into conjunctive normal form

A logical formula is in conjunctive normal form (CNF) if it is a conjunction (sequence of ANDs) consisting of one or more clauses, each of which is a disjunction (sequence of ORs) of one or more literals (or negated literals). Any logical formula can be translated into CNF by applying a well-known set of rules.

We propose to apply the same set of rules (with the addition of a new rule to deal with the *if-then-else* construct) to normalize the boolean expressions included in the body of the OCL constraints. Note that the body of constraint itself must be a boolean expression. Besides, boolean expressions appear frequently in OCL constraints, for instance, as parameters of the *forAll* and *select* iterators.

The rules are the following:

1. Eliminate the *if-then-else* construct and the *implies* and *xor* operators using the rules:
    a. X *implies* Y → *not* X *or* Y
    b. *if* X *then* Y *else* Z → (X *implies* Y) and (*not* X *implies* Z) → (*not* X *or* Y) *and* (X *or* Z)
    c. X *xor* Y → (X *or* Y) *and not* (X *and* Y) → (X *or* Y) *and* (*not* X *or not* Y)
2. Move *not* inwards until negations be immediately before literals by repeatedly using the laws:
    a. *not* (*not* X) → X
    b. DeMorgan's laws: *not* (X *or* Y) → *not* X *and not* Y

       not (X *and* Y) → *not* X *or not* Y
3. Repeatedly distribute *or* over *and* by means of:
    a. X *or* (Y *and* Z) → (X *or* Y) *and* (X *or* Z)

Once transformed into conjunctive normal form (and applying the rules to deal with *not X<=Y* and *not X>Y* expressions, see Section 3.1), *CorrectProduct'''* results in:

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

which is exactly the simplified form for this constraint.

3.4 Rule application

Given an expression *exp*, the simplified form of *exp* is obtained by applying repetitively the previous rules over *exp* until no rules can be applied.

The following simple algorithm can be used for this purpose:

Algorithm: *Obtaining the simplified form of a constraint*

```
SimplifiedForm(OCLExpression constraintBody) : OCLExpression
  OCLExpression result := constraintBody
  while ( isPossibleToApplyRules(result)
    rule r := selectRuleToApply(result);
    result := apply(result, r)
  fwhile
return result
```

where *isPossibleToApplyRules* examines the expression to determine if any simplification rule can be applied, *selectRuleToApply* choose one of the possible rules and *apply* modifies the expression according to that rule.

There are no cycles among the proposed set of simplification rules so termination of the process is guaranteed. There are a few rules that when applied over an expression X produce an output expression that can be used as an input for another rule. However, following any sequence of simplifications, X never returns to a state where we can apply the first rule over X again. What it may happen is that some of the applied rules can be used again to simplify a subset of X not previously targeted (i.e. there may exist a subexpression $x \subset X'$ targeted by those rules).

4. Determining the Potentially-Violating Structural Events

The aim of this chapter is to determine which kinds of events, when applied over the IB, may induce the violation of a given integrity constraint. This knowledge helps to improve the efficiency of the integrity checking process by discarding the verification of those constraints not possibly violated by the set of structural events applied during the modification of the IB.

Given a constraint *c* and an event type *ev*, we define that *ev* is a potentially-violating structural event (PSE) for *c* if the application of an structural event of type *ev* over a consistent state of the IB may result in a new state of the IB that does not satisfy *c*. We consider that a constraint *c* with a body *b* and defined over a context type *ct* is not satisfied in a state *s* of an IB when there is an entity *e* of *ct* that evaluates *b* to false in *s*.

We call these event types *potentially-violating events* (*PSEs*) since defining that the event type *ev* is a PSE for a constraint *c* does not necessarily imply that *c* is violated every time an event of type *ev* is applied over the IB (it depends on the exact state of the IB and on the parameters of the particular structural event at run-time).

When computing the set of PSEs, we assume that integrity constraints are represented as instances of the OCL metamodel. According to this metamodel, each constraint can be regarded as a binary tree, where each node represents an atomic subset of the OCL expression (an operation, an access to an attribute or an association …). The tree is a binary tree since all OCL predefined operators have at most two parameters, and user-defined operations have been already unfolded as part of the simplification process presented in the previous chapter.

Given the tree that represents the body of an integrity constraint as an instance of the OCL metamodel, our method performs two different steps to determine the PSEs that may violate the constraint:

1. Marking the tree. Each node (i.e. each atomic subset of the OCL expression) is marked with information about the kinds of modifications over the model elements referenced in the node (an increase in its value, a decrease…) that may induce the violation of the constraint. For instance a node representing an access to an attribute may be marked with a plus sign to indicate that the constraint could be violated if

the attribute value is increased during the IB update. The computation of this information depends on the relationships between the different nodes of the tree.

2. Drawing the PSEs. The method determines the PSEs by taking the mark and the subexpression corresponding to each node into account. In short, the method identifies which event type may produce the kind of change required by the mark. Following the previous example, the method would determine that an update event over the attribute referenced in the node may increase its value, and thus, it may violate the constraint.

The rest of the chapter is structured as follows. Section 4.1 presents the different event types our method deals with. Section 4.2 introduces some basic concepts about the OCL metamodel. Then, in Section 4.3 and Section 4.4 we explain the two previous steps. Finally, section 4.5 summarizes the obtained results.

4.1 List of event types

To determine the PSEs that may violate an OCL integrity constraint our method reasons about the following set of event types. Each one of these event types is instantiated at run-time to generate the different kinds of modifications over the IB.

- *InsertET(ET)*: it represents the insertion of a new entity in the entity type *ET*. The new instance may have its attributes initialized but it does not participate in any relationship. Example: events of type *InsertET(Sale)* insert a new sale in the IB. For instance, the event *InsertET(Sale, 1, '26/06/2006', 1000, '26/07/2006')* would insert a new sale in the IB, initialized with the values *id=1*, d*ate='26/06/2006', amount =* 1000 and *paymentDate='26/07/2006'*.

- *UpdateAttribute(Attr,ET)*: it updates the value of the attribute *attr* of an entity of the entity type *ET*. When *ET* is clear from the context (i.e. the name of the attribute is not ambiguous) we also refer to this event type as just *UpdateAttribute(Attr)*. Example: events of type *UpdateAttribute(price, Product)* change the price of a product.

- *DeleteET(ET)*: it deletes an entity from an entity type *ET*. Example: events of type *DeleteET(Shipment)* imply the deletion of a shipment from the IB.

- *SpecializeET(ET)*: it specializes an entity of a supertype of an entity type *ET* to *ET*. Example: A *SpecializeET(RestrictedProduct)* event specializes a product into a restricted product.

- *GeneralizeET(ET)*: it generalizes an entity of a subtype of an entity type *ET* to *ET*. Example: A *GeneralizeET(Product)* event transforms restricted products into just "common" products.

- *InsertRT(RT)*: it inserts a new relationship in the relationship type *RT*. Example: events of type *InsertRT(DeliveredIn)* create new relationships between pairs of sales and shipments.

- *DeleteRT(RT)*: it removes a relationship from a relationship type *RT*. Example: events of type *DeleteRT(DeliveredIn)* remove a sale from a shipment.

Reified entity types (i.e. association classes) have two different facets: the entity type facet and the relationship type facet. Since we do not have specific events for dealing with the insertion (deletion) of entities of reified entity types, to insert an entity in a reified entity type we must combine an *InsertET* (to create the entity facet) and an *InsertRT* event (to create the relationship facet). Likewise, to delete an entity from a reified entity type we combine the *DeleteET* and *DeleteRT* events.

To deal with 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 need to generate the corresponding specialize (generalize) event 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).

We would like to remark that our set of events does not exactly correspond to the kind of events defined in the UML language (as defined in the *Action* packages [68]). The event types we use are more basic than those of the UML. Moreover, our independence of a particular modeling language allows us to incorporate our results to different predefined sets of structural event types providing that we define the correspondence between our event types and those different sets.

Besides, the result of our method in terms of our set of internal event types can be easily expressed in terms of the UML event types. For the sake of completeness we provide in the next subsection the correspondence between both sets.

4.1.1 Correspondence with the event types defined in the UML language

The set of event types allowed in UML behavior specifications [68], is the following:

- *AddSructuralFeatureValueAction(StructuralFeature s)*: It adds a new value for the structural feature *s* to the object indicated at run-time. In UML 2.0, a structural feature is either an attribute or an association end (i.e. a role in a relationship type).

- *CreateLinkAction(Association a)*: It creates a new link (i.e. relationship) for the association (i.e. relationship type) *a*.

- *CreateLinkObjectAction(Association a)*: It creates a new link object for the association class (i.e. reified entity type) *a*.

- *CreateObjectAction(Classifier c)*: It creates a new instance of the classifier (i.e. entity type) *c*.

- *DestroyLinkAction(Association a)*: It removes a link from *a*.

- *DestroyObjectAction(Classifier c)*: It removes an object from *c*.

- *ReclassifyObjectAction(Classifier[] newClassifiers, Classifier[] oldClassifiers)*: It adds to the object indicated at run-time the list of classifiers specified in *newClassifiers* and removes those in *oldClassifiers*.

- *RemoveStructuralFeatureValueAction(StructuralFeature s):* It removes the value of *s* in the object indicated at run-time.

Given this set of event types table 4.1 shows the correspondence between our internal set of events (first column) and those of the UML (second column).

Table 4.1. Correspondence between our events and the UML event types

Internal event types	UML event types
InsertET(ET)	- *CreateObjectAction* over *ET* - *CreateLinkObject* over *ET* (if *ET* is an association class) - Any of the previous events over a subtype of *ET* (which induces an insertion over *ET*)
UpdateAttribute(Attr,ET)	- *AddStrucuturalFeature* over *Attr* (possibly preceded by a *RemovalStructuralFeatureAction* to remove the previous attribute value)
DeleteET(ET)	- *DestroyObjectAction* over *ET* - *DestroyLinkAction* over *ET* (if *ET* is an association class) - Any of the previous events over a supertype or subtype of *ET* (both induce the deletion of the instance over *ET*)
SpecializeET(ET)	- *ReclassifyObjectAction* with *ET* in the set of new classifiers.
GeneralizeET(ET)	- A *ReclassifyObjectAction* with a direct subtype of *ET* in the set of old classifiers.
InsertRT(RT)	- *CreateLinkAction* over *RT* - *CreateLinkObject* over *RT* (if *RT* is an association class)
DeleteRT(RT)	- *DestroyLinkAction* over *RT*

4.2 The OCL metamodel

The representation of the constraints as instances of the OCL metamodel facilitates their treatment during the different steps of the method. At the moment of writing this thesis, the last adopted specification of the OCL metamodel is [67].

The basic structure of the OCL metamodel (see Figure 4.1) 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 4.2) 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 [67 ch. 11], such as the add operator ('+') or the 'and' operator. These *OperationCallExp* expressions may include a list of arguments if the referred operation has parameters. Note that, as can be seen from Figure 4.2, the current version of the OCL standard is not yet completely aligned with the last version of the UML 2.0 standard since some of the OCL metaclasses (as *AssociationEndCallExp*) still reference obsolete UML metaclasses as the *AssociationEnd* metaclass, replaced by the metaclass *Property* in the UML 2.0.

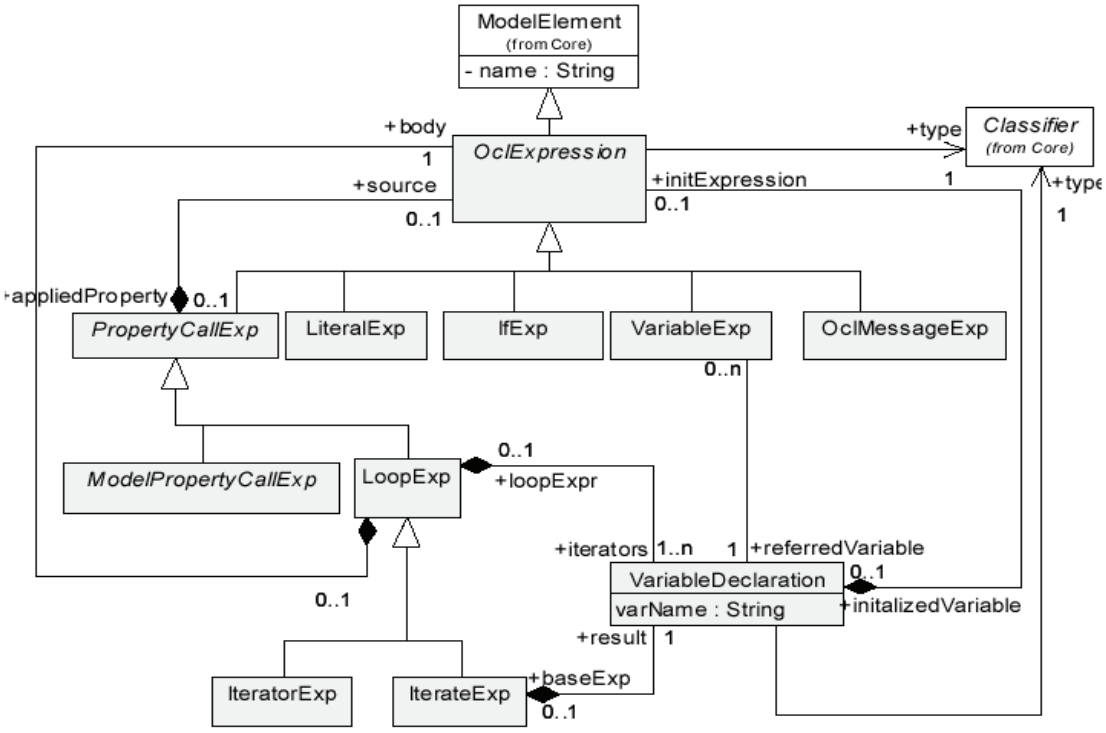


Figure 4.1. Basic structure of the OCL metamodel

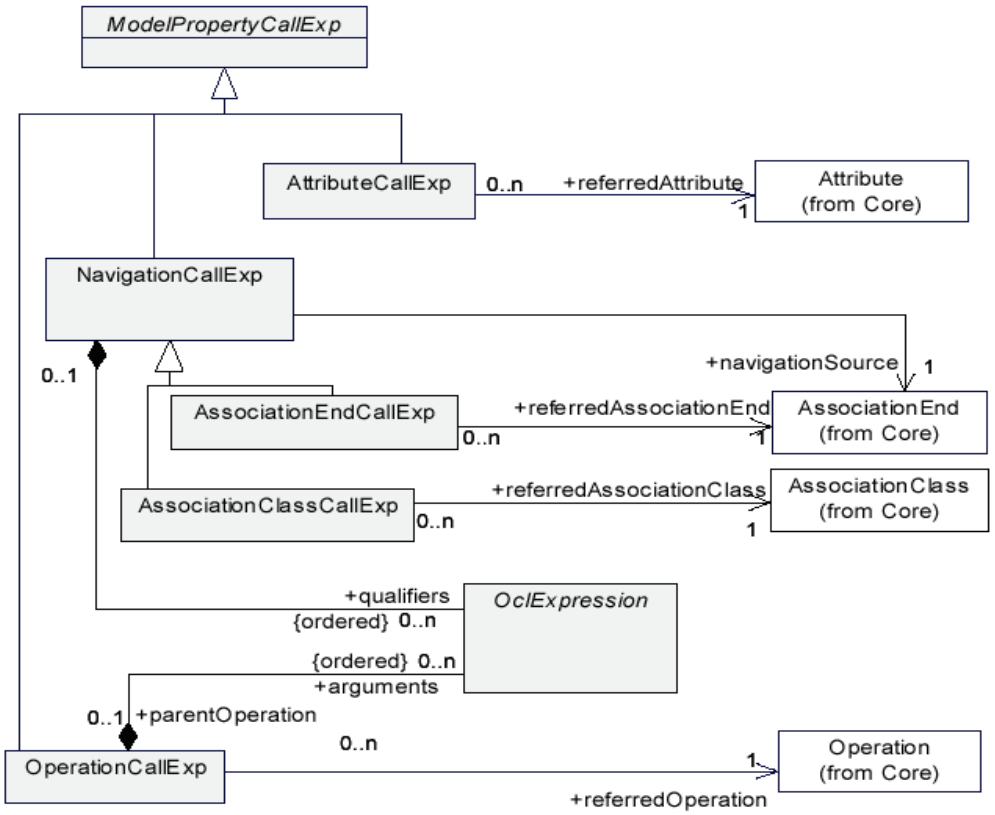


Figure 4.2. OCL Metamodel fragment for *ModelPropertyCallExp*

When expressing a constraint as an instance of the OCL metamodel, the body of the constraint can be regarded as a binary tree where each node represents an atomic subset of the OCL expression defining the constraint body (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 4.3 the constraint *ValidShipDate* (*self.shipment->forAll(sh| sh.plannedShipDate<=self.paymentDate+30)*) as an instance of the OCL metamodel. 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.shipment*). In its turn, this left child is represented as an instance of the *AssocationEndCallExp* metaclass corresponding to the navigation through the association end *shipment*. Its source is the access to the *self* variable. The right child of the root is the body of the iterator expression (*sh.plannedShipDate <= self.paymentDate+30)*. 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 *plannedShipDate* (with a last child representing the access to the *sh* variable). The second one is the operation '+' between the *paymentDate* attribute (left child) and the integer 30 (right child).

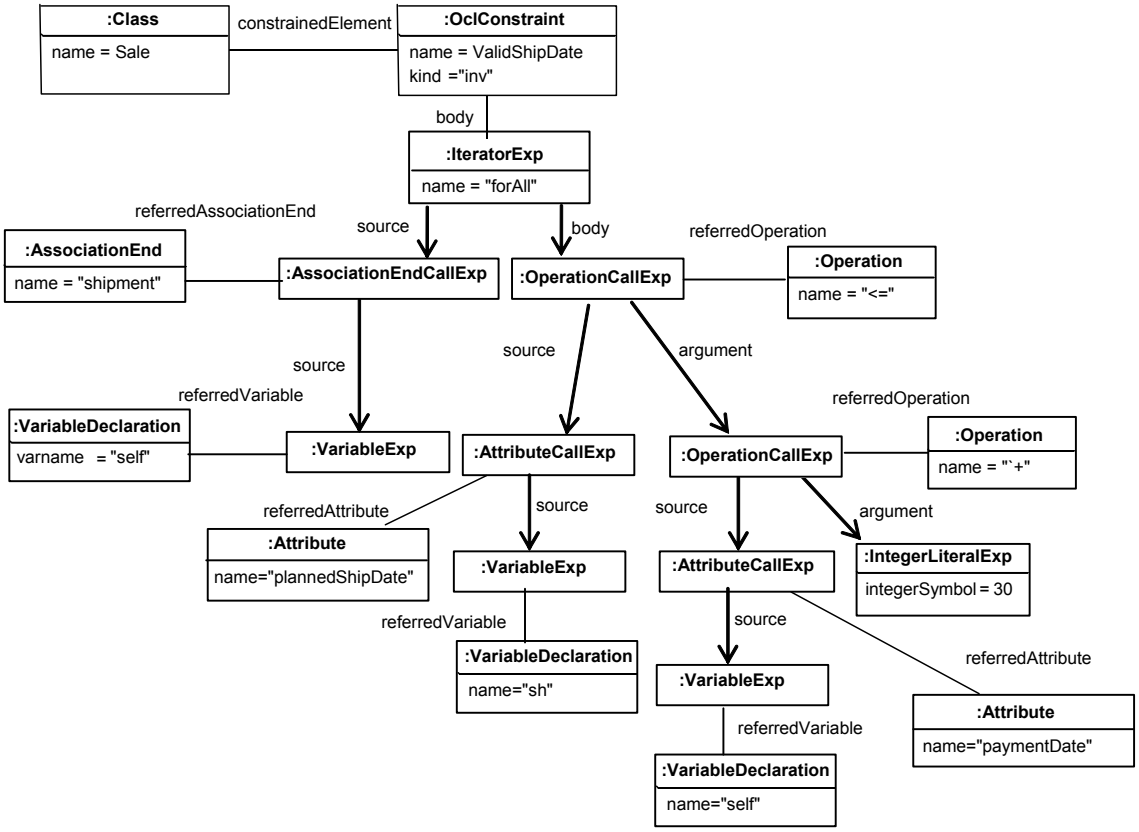


Figure 4.3. Constraint ValidShipDate as an instance of the OCL metamodel

As can be seen from the previous figure, a complete representation of the constraint as an instance of the OCL metamodel is quite cumbersome. Therefore, from now on we will express the constraints using a simplified version of the previous representation. Figure 4.4 shows *ValidShipDate* represented in this simplified form, where we combine in the same tree node the kind of OCL subexpression and the name of the model element referenced by the node.

Figures 4.5 – 4.8 show the other constraints of the running example (Figure 1.2) expressed as instances of the OCL metamodel.

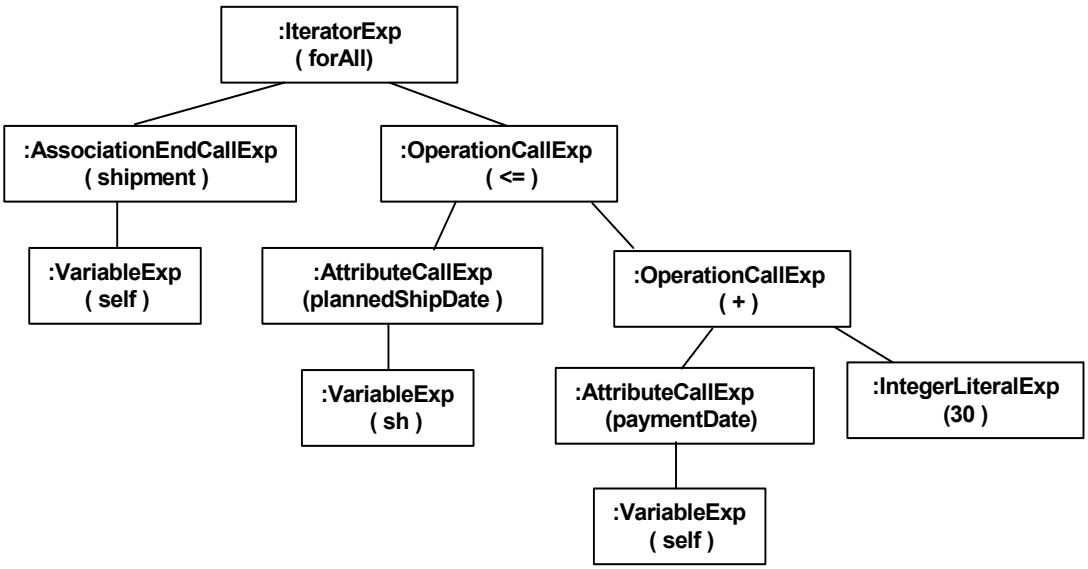


Figure 4.4. Simplified representation of *ValidShipDate*

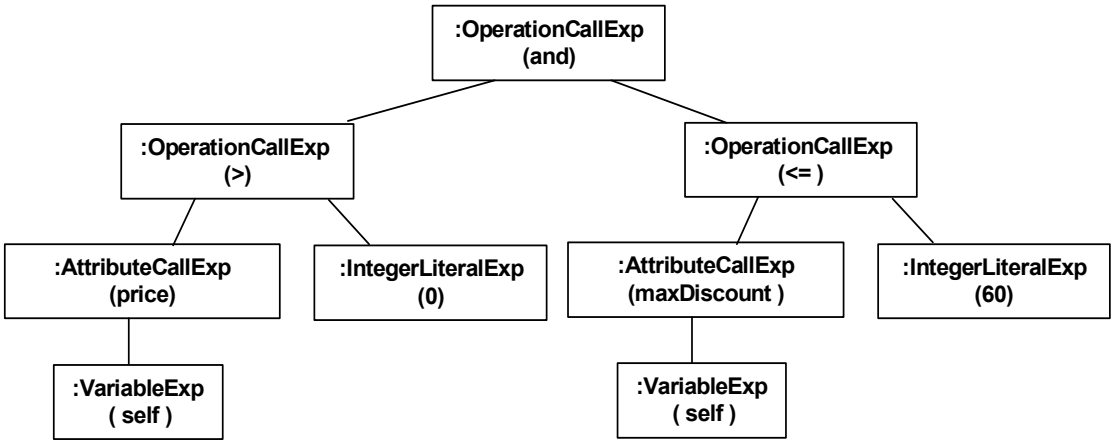


Figure 4.5. Constraint *CorrectProduct* as an instance of the OCL metamodel

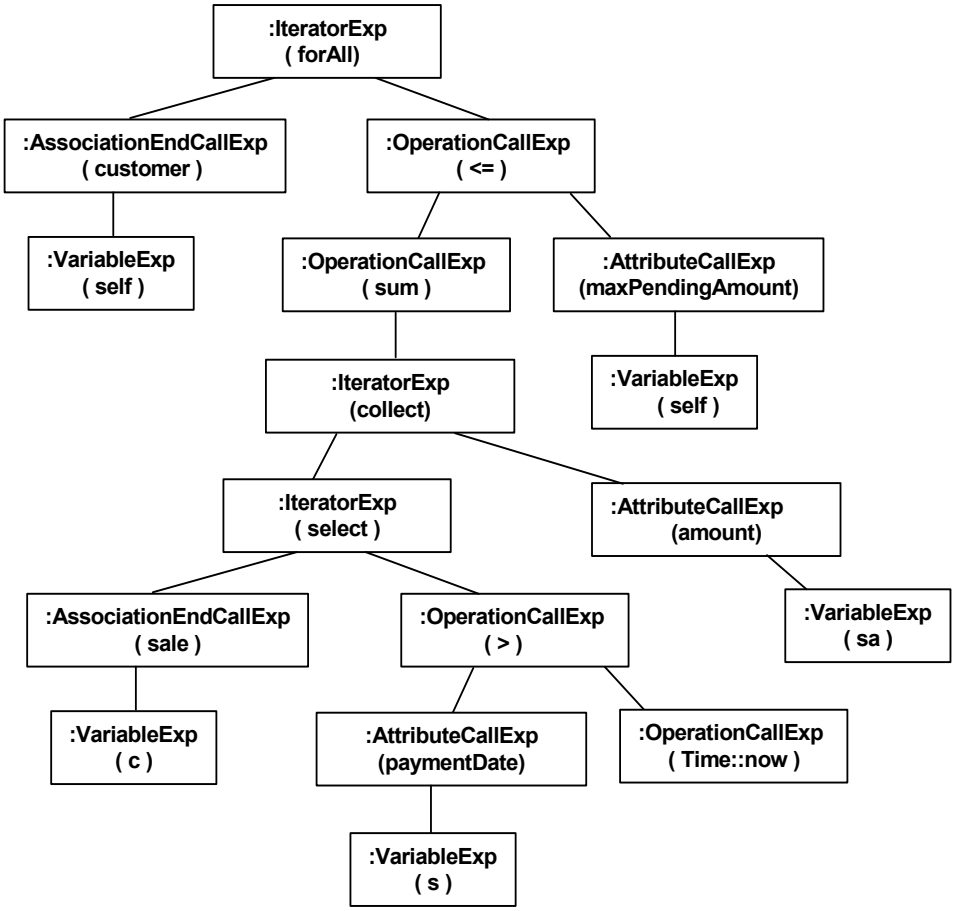


Figure 4.6. Constraint *NotTooPendingSales* as an instance of the OCL metamodel

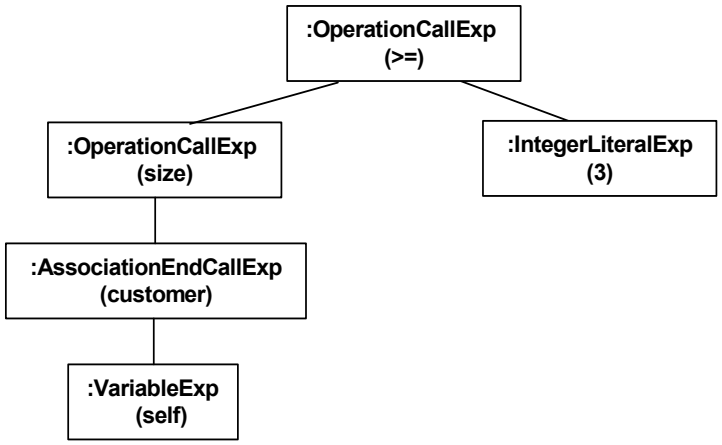


Figure 4.7. Constraint *AtLeastThreeCustomers* as an instance of the OCL metamodel

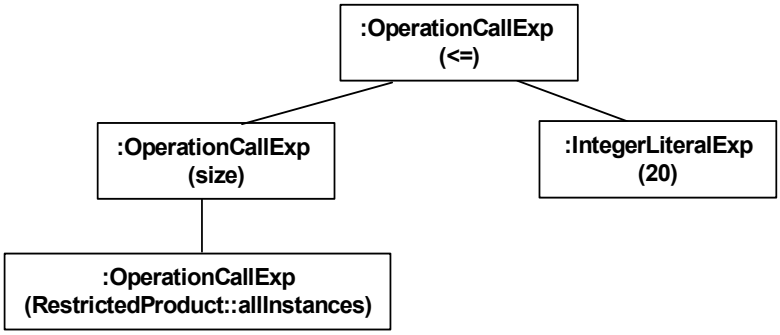


Figure 4.8. Constraint *NumberOfRestrictedProducts* as an instance of the OCL metamodel

4.3 Marking the constraint tree

To compute the PSEs it is not enough with examining each part of the OCL expression separately. For instance, to determine whether the constraint *AtLeastThreeCustomers* may be violated by a customer assignment to a category or by a customer removal from a category we can not merely take into account the subexpression *self.customer->size()*. In fact, both events change the value resulting from the evaluation of this subexpression. However, since the *size* operation is involved in a '>=' comparison operator, only removing a customer from a category may induce a violation of this constraint. On the contrary, if we had used '<=' instead, the expression could be violated only when assigning a new customer to the category.

Therefore, and prior the computation of the PSEs, we need to analyze the relationship between the different nodes of the OCL constraint tree. For each node we must determine which kind of modifications over the elements referenced by the node may induce a constraint violation. The kind of changes depends on the node type and on the information propagated by the parent node. Then, we propagate this information to the children node to repeat the process, following a preorder traversal of the tree. In a preorder traversal, we process all nodes of the tree by first processing the root and then, recursively, processing in preorder the children subtrees. To start the process the method assumes that the root node is marked with the *und* symbol (see below).

There are three different symbols to propagate:

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

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

The symbol 'und' is used by operations like *'and'* or *'or'*. It denotes that the node does not influence its children expressions at all. The events that can violate '*A and B'* are the same that violate *A* plus those of *B* stand-alone. A *forAll* also propagates an *und* to its *body* since the events that can violate the body expression of the iterator are the same events that can violate the expression stand-alone.

Tables 4.2-4.5 show the symbol propagation for all combinations of each kind of node (columns) and symbol (rows) that may be received from a parent node. Then, cells indicate which symbols must be propagated to the children nodes when that type of node receives that symbol from its parent.

Sometimes a node may propagate more than one symbol to its children. When a node receives several symbols from its parent node, the final value that the node propagates is obtained by applying the table information to each received symbol. When a node has two children the cell states the symbol (or symbols) for each child. For the sake of clarity, propagation of *und* symbols is represented by blank cells. Blank cells can also indicate that there is no constraint including such combination.

First, Table 4.2 shows the propagation of some basic OCL expressions like navigations through roles (i.e. association ends) or reified entity types (i.e. association classes) and access to attribute values, variables and constants. For variables and constants, the node always is a leaf of the tree, and thus, no symbol can be propagated. 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 the navigation is to increase the size of the collection of objects where the navigation is applied over (that justifies the propagation of the '+' symbol). 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).

Table 4.2. Basic OCL expressions

	1. Navigation AssocitationEnd	2. Navigation AssociationClass	3. Access attribute	5. Access to variable	6. Constant expression
1. und			+		
2. +	+	+	+		
3. –	–	–	+		

Then, Table 4.3 deals with the generic operations of all OCL types, which in the OCL standard are defined as operations of the *OclAny* predefined type (all other types are subtypes of the *OclAny* supertype). Among those operations, the '=' operator is an example of a node with two children. In this case we propagate both symbols to each child (an equal comparison may be violated either by an increase or decrease in the values of any of the operands). This default behavior can only be optimized when one of the operands evaluates to a natural type and the other is the integer zero. Then, it is enough to propagate a '+' symbol to the natural operand (since it evaluates to a natural value, the operand cannot possibly take a value < 0, which means that the constraint cannot be violated by a decrease in the value of that operand). For *oclIsUndefined*, *oclIsTypeOf* and *oclIsKindOf* operations the constraint can be propagated if these operations are applied over a new object (that may not satisfy the type condition), thus, we propagate a '+' symbol. The *allInstances* operation is a leaf of the tree so no symbol is propagated.

Table 4.3. OclAny operations

	1. =		2. oclIsUndefined	3. oclAsType	4. oclIsTypeOf	5. oclIsKindOf	6.allInstances
1. und	+ –	+ –	+		+	+	
2. +				+			
3. –				–			

Table 4.4 presents the treatment of operations defined for primitive types. Here, 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 '+'). 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 '–'). As commented before, 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 in the next section).

Table 4.4. Primitive types operations

	1. +,–,*,/, div, mod (real,int)		2. +,* (nat)		3. –,/ (nat)		4. <, <=		5. >, >=		6. max min		7. floor round	8. abs	9. –	10. and Or		11. not	11.size, toInt toReal (string)	12. concat	
1. und							+	–	–	+											
2. +	+ –	+ –	+	+	+	–					+	+	+	+ –	+ –				+	+	+
3. –	+ –	+ –	–	–	–	+					–	–	–	+ –	+ –				–	–	–

Finally, Table 4.5 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 the *forAll* 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 propagate the *und* symbol. We follow a similar reasoning with the *select* iterator (see also the special treatment for *select* expressions when obtaining the PSEs in the next section). For the generic *iterate* operator, we propagate all symbols since this generic iterator 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* (column 4) operator propagates also the symbols in brackets. The sum of an attribute *at* of a set of objects *s* may increase either when adding to *s* an object with a positive value in *at* or when removing from *s* an object with a negative value in *at*.

Table 4.5. Iterator and collection expressions

	1. forAll		2. select		3. iterate		4. sum	5. count collect		6. U,∩, product		7. –		8. sym Differ		11. size asX, Flatten	12.at, indexOf, subseq	
1. und	+				+ –	+ –											+ –	+ –
2. +			+		+ –	+ –	+ (–)	+	+	+	+	+	–	+ –	+ –	+	+ –	+ –
3. –			–		+ –	+ –	– (+)	–	–	–	–	–	+	+ –	+ –	–	+ –	+ –

The application of these tables over the constraint *ValidShipDate* is shown in Figure 4.9. We start with processing the *forAll* iterator. Since it is the root of the tree it does not receive any initial information. To mark its children we use cell 4.5:1.1 (Table 4.5 row 1,

column 1) which states that the left child must be marked with the '+' symbol while the right must be 'und' (shown as a blank symbol in the tree). The left child (the navigation through the association end *shipment*) propagates a '+' to its single child (cell 4.2:2.1). The root of the subtree corresponding to the body of the *forAll* is the node representing the operation '<='. This node propagates a '+' to the left child and a '–' to the right one (cell 4.4:1.4). In its turn, the left child (the access to the attribute *plannedShipDate*) propagates a '+' to its child (cell 4.2:2.3). The right child (the *plus* operator) propagates a '–' to both children (cell 4.4:3.2).

Figures 4.10-4.13 show the results of marking the trees corresponding to the rest of constraints.

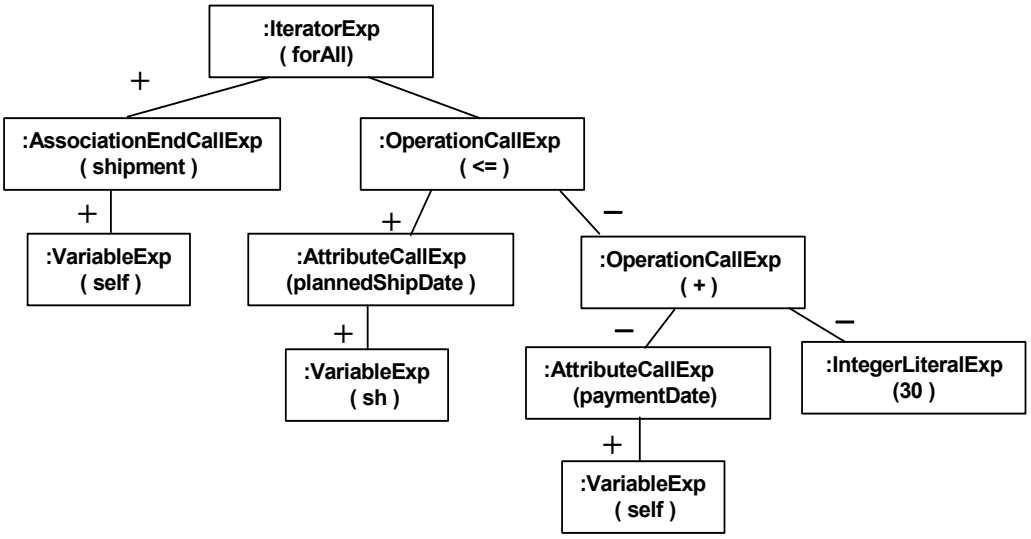


Figure 4.9. Marking the tree corresponding to *ValidShipDate*

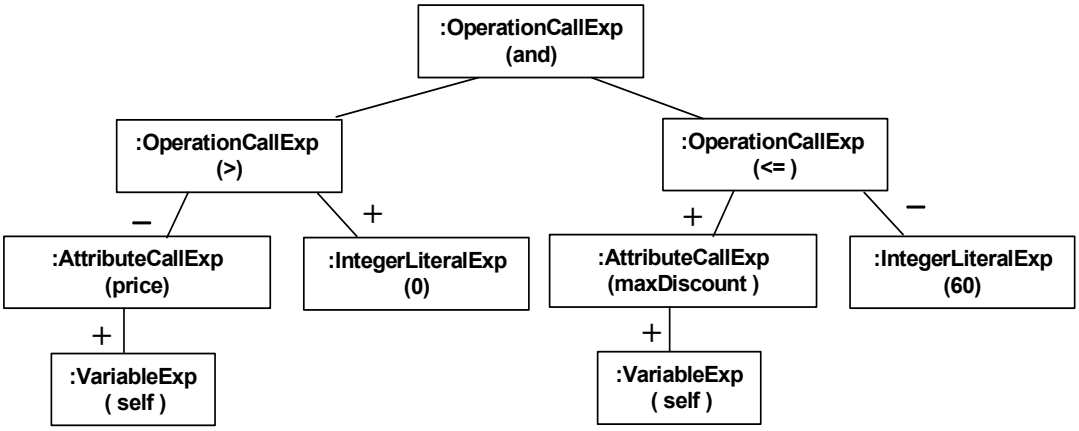


Figure 4.10. Result of marking *CorrectProduct*

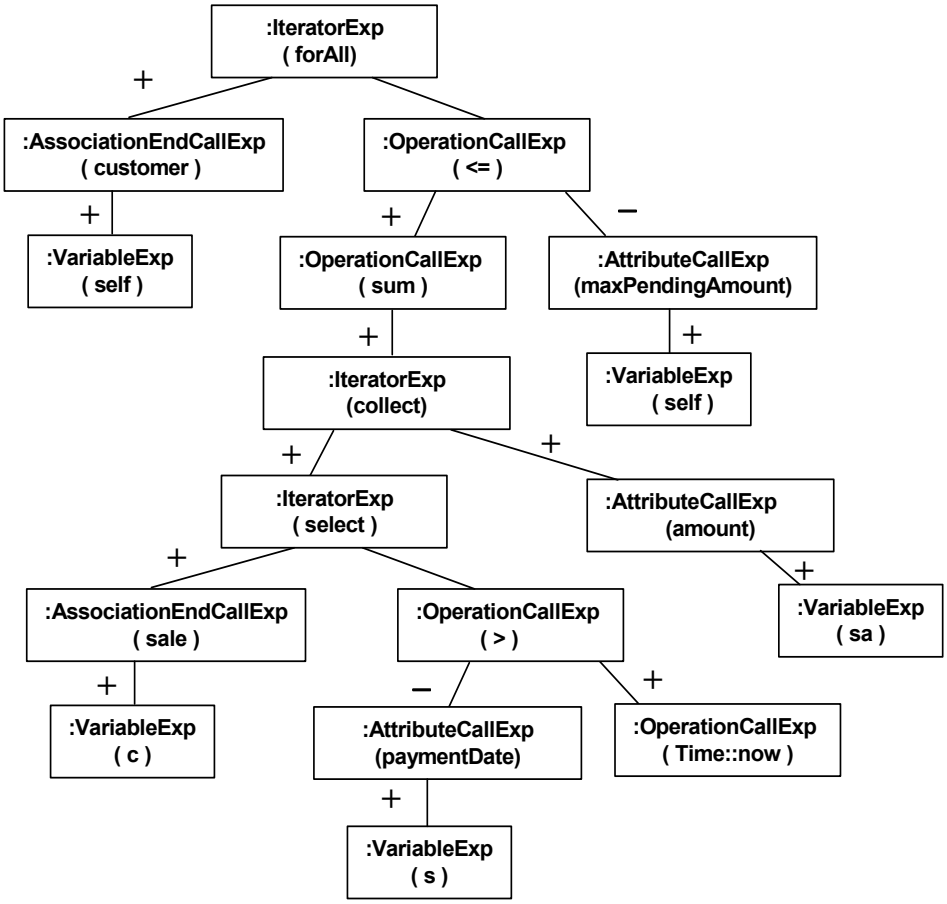


Figure 4.11. Result of marking *NotTooPendingSales*

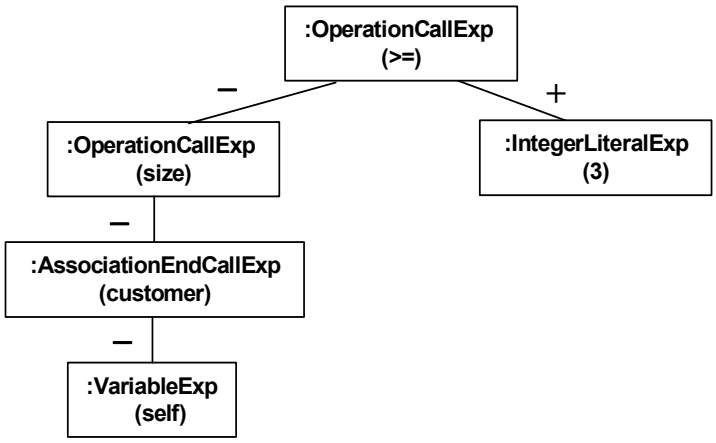


Figure 4.12. Result of marking *AtLeastThreeCustomers*

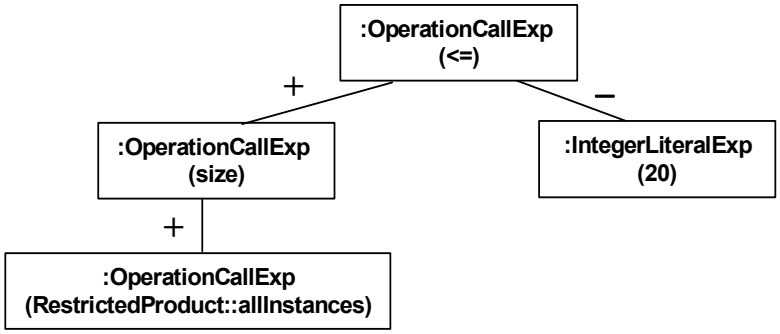


Figure 4.13. Result of marking *NumberOfRestrictedProducts*

4.4 Drawing the potentially-violating structural events

Once the tree is marked as explained in the previous section, our method is able to determine the PSEs that may violate the integrity constraint. Now, the tree is traversed in postorder (we process all nodes of the tree by first recursively processing in postorder the children subtrees and then the root). During the traversal we attach to each node of the tree the information about the PSEs generated because of that particular node.

Table 4.6 describes the set of PSEs we determine for each node in terms of the node type (columns) and its marks (rows). Among all columns used in the tables of the previous section we only include in Table 4.6 those columns with a direct effect on the generation of the PSEs.

When a PSE appears between brackets implies that more sophisticated conditions and patterns must be evaluated before considering that event type as a PSE for a constraint including a node of that type with that mark. These conditions as well as the meaning of the *Opp* keyword in the last two columns will be explained later on. In what follows we discuss in detail each column.

Table 4.6. Computation of the PSEs

	1. Navig AssEnd	2. Navig AssClass	3.Access Attribute	4. Access to var	5. IsTypeOf	6. IsKindOf	7. allInst	8. not	9. select
1. und			UpdAttr		GenET SpeET	GenET		Opp_1	
2. +	InsertRT (DeleteRT)	InsertRT (DeleteRT)	UpdAttr	(InsertET) (SpeET)			InsertET (SpeET)		Opp_2
3. –	DeleteRT	DeleteRT	UpdAttr	(InsertET) (SpeET)			DeleteET (GenET)		

First we discuss the possible options for nodes representing a navigation through an association end (column 1). When the association end is labeled with a '+' symbol, the

constraint may be violated by an insertion over the association (event type *InsertRT*) where the association end belongs. Remember that '+' pointed out that the constraint could be violated due to an increase of the number of elements resulting from the navigation through that association end, and thus, the events that can cause the violation are those events that increase such number. That is precisely what events of type *InsertRT* do. In a similar way, if it is labeled with a '–', the critical event is the deletion of a link of the association (event type *DeleteRT*) since we are interested in reducing the number of links of the association.

The *DeleteRT* event type that appears in brackets is only relevant for navigations included in the body expression of a *select* iterator. A *select* returns only those objects that evaluate the select condition to true. On the contrary, those evaluating the condition to false or undefined will not be selected. Since events of type *DeleteRT(RT)* (where *RT* is referenced in the select condition) may cause the subexpression including the navigation through RT to return an undefined value, this event type must also be considered as a PSE for constraints that may be violated when the amount of elements returned by a *select* decrease. Note that this PSE is not generated when at the end of the sequence of navigations where RT is included in, we find an iterator or a collection operation since then, according to the OCL standard, the result is never undefined (even if the iterator or the collection operation are applied over an empty collection).

The same reasoning serves to explain the processing of navigations towards an association class. The only difference is that, in this case, the relationship type appearing as a parameter of the *InsertRT* or *DeleteRT* event type is a reified relationship type instead of a simple relationship type.

For attributes, we have that all kinds of conditions over the attributes may be violated if we change the value of the attribute (*UpdateAttribute* event type).

When accessing a variable, we need to consider the event types *InsertET(ET)* and *SpecializeET*(*ET*) as PSEs for the constraint (where *ET* represents the type of the variable) when the following additional conditions apply:

- The referenced variable must be the *self* variable
- The parent node must be either an *AttributeCallExp* or a *NavigationCallExp*
- The node must not be included inside the body of an iterator expression
- When the parent node is an *AttributeCallExp* the condition regarding the attribute must compare the attribute value with a constant (as in *self.attr>5*) or with another attribute of the same *self* instance (as in *self.attr1>self.attr2*). Conditions comparing the value of the attribute with the values of different instances (as in *self.role$_1$->forAll(i| i.attr>self.attr)*) does not generate any PSEs. In such a case, the constraint

can only be violated if the navigation $self.role_1$ is not empty which is already controlled when processing the part of the subtree dealing with this navigation expression.

- When the parent node is a *NavigationCallExp* the variable node must be marked with a '–' symbol. This case indicates that all new entities of the entity type must have a minimum set of relationships with other entities, and thus, if the new created entity does not participates in this minimum set, the constraint will be violated. For instance, all categories must have at least three customers, and thus, when creating a category we must also assign to the category an initial set of customers to avoid violating the constraint.

- The *SpecializeET(ET)* event is only generated when *ET* has at least a direct supertype, which implies that new instances of *ET* can be created either by inserting a completely new entity or by specializing to *ET* an existing instance of a supertype of *ET*. Moreover, the constraint body must reference at least an attribute or relationship type defined in the supertype. Otherwise the specialized entity would not present values in any of the elements referenced in the constraint and thus, it will be surely consistent with it. As an example, consider a constraint *context RestrictedProduct inv: self.maxUnits >0*. When specializing an existing product *p* into the restricted product category, *p* cannot violate the constraint until we issue an *UpdateAttribute(maxUnits, RestrictedProduct)* event type to initialize its *maxUnits* value. Therefore, the specialization of *p* itself does not violate this constraint. Instead, if we restrict *RestrictedProduct* entities to have a price over 10000, specializing a product into a restricted product may induce a constraint violation since the product may already present a price over 10000.

For *oclIsTypeOf(Type)* and *oclIsKindOf(Type)* operations the constraint may be violated when changing the type of the affected entities. In the first case, when the entity is either generalized to a supertype or specialized to a subtype of *Type* (*object.oclIsTypeOf(Type)* returns true iff the type of *object* is exactly *Type*). In the latter, only when it is generalized (*object.oclIsKindOf(Type)* returns true if *object* is either of type *Type* or of one of the subtypes of *Type*). Note that, we must generate a PSE for each direct supertype (and subtype, for *oclIsTypeOf* operations) of *Type*.

The *Type*::*allInstances* operation generates an *InsertET(Type)* event type as a PSE (and a *SpecializeET(Type)* when *Type* has at least a supertype) when the corresponding node is marked with a '+'. When issuing these events the number of instances returned by the *allInstances* operation is increased, and thus, their application over the IB may violate constraint containing an *allInstances* node marked with the '+' symbol. Similarly, when the symbol is a '–', it generates a *DeleteET(Type)* and a *GeneralizeET(supertype)* for each direct supertype of *Type*.

The processing of expressions affected by *select* and *not* operators is more complex. The effect of the *not* operator is defined by means of the operation O*pp*$_1$*()*. This operation replaces the PSEs attached to the child node of the *not* operator with their opposites (i.e. the events violating a *not X* expression are the opposite events to those PSEs that violate *X*). The exception is when the *not* operator appears in front of an *oclIsUndefined* operation, where we must change the PSEs of the whole child subtree with their opposites, since even changes in one of earlier nodes of the subtree may turn the whole expression to (not) undefined. Note that when the *not* operator appears in front of an equal comparison the *Opp*$_1$*()* does not have any effect (since the particular node corresponding to the equal comparison never holds a PSE). This is perfectly correct since the events that may violate the equal comparison are the same as the ones that may violated a not-equal comparison. In both cases, any change on one of the operands may cause the comparison to return a *false* value.

The basic idea for the *Opp*$_1$ operation is that the opposite of an insertion (specialization) is a deletion (generalization) and the opposite of a deletion (generalization) is an insertion (specialization). The opposite of an update event is the event itself. The opposite of the opposite of an event must return the original event. Therefore, the opposites for each event are:

- Opposite(InsertET(ET))=DeleteET(ET)
- Opposite(DeleteET(ET))=InsertET(ET)
- Opposite(UpdateAttribute(Attr,ET))=UpdateAttribute(Attr,ET)
- Opposite(InsertRT(RT))=DeleteRT(RT)
- Opposite(DeleteRT(RT))=InsertRT(RT)
- Opposite(SpecializeET(ET))=GeneralizeET(ET_{sup}) where ET_{sup} is a supertype of *ET*. In particular, ET_{sup} must be the supertype appearing as a parameter of the operation *oclIsTypeOf* operation that has produced the *SpecializeET* PSE. Note that, even though other cells generate also *SpecializeET* events, among them only the *oclIsTypeOf* operation may appear next to the *not* operator.
- Opposite(GeneralizeET(ET))=SpecializeET(ET_{sub}) where ET_{sub} is a subtype of *ET*. In particular, ET_{sub} must be the subtype appearing as a parameter of the operation *oclIsKindOf* or *oclIsTypeOf* operations that have produced the *GeneralizeET* PSE. As before, other cells generating *GeneralizeET* events cannot appear next to the *not* operator.

A similar thing happens with a *select* expression. When the constraint may be violated by an increase in the number of elements returned by a *select* expression we are interested in the events that favor an object to satisfy the select expression. These events are the opposite of the events that favor the violation of the select condition. Therefore, we must apply the operation *Opp*$_2$ over the *select* body to transform all the events initially computed

(i.e. those that violate the select condition). This operation simply applies the previous opposite function Opp_1 to all the PSEs attached to nodes included in the select body (and not only to the PSEs of the immediate child as in the *not* operator).

Figure 4.14 applies Table 4.6 to the *ValidShipDate* constraint. Since the traversal is in postorder we start by processing the leaves of the tree. First, we process the access to the variable *self* marked with the symbol '+'. The variable does not generate any PSE (it does not verify the conditions stated above). After this step, we consider the navigation through the *shipment* association end. According to the table, an association end marked with the '+' symbol generates an *InsertRT* event type, in this case, an *InsertRT(DeliveredIn)*. Then, we proceed with the right child of the *forAll* iterator. The access to the *sh* variable does not affect the computation of the PSEs but the access to the *plannedShipDate* attribute produces the *UpdateAttribute(plannedShipDate)* event type. Finally, the access to the attribute *paymentDate* increases the set of PSEs for the constraint with the *UpdateAttribute(paymentDate)* event type. Note that its child (the access to the *self* variable) neither generates an *InsertET* event type since the condition over the *paymentDate* attribute involves comparing its value with the value of an attribute of a different object.

Figures 4.15 – 4.18 apply Table 4.6 to the rest of the constraints. For the sake of clarity, in the figures we indicate events of type *UpdateAttribute(Attr,ET)* as just *UpdateAttribute(Attr)* when *ET* is clear from the context (i.e. the name of the attribute is not ambiguous).

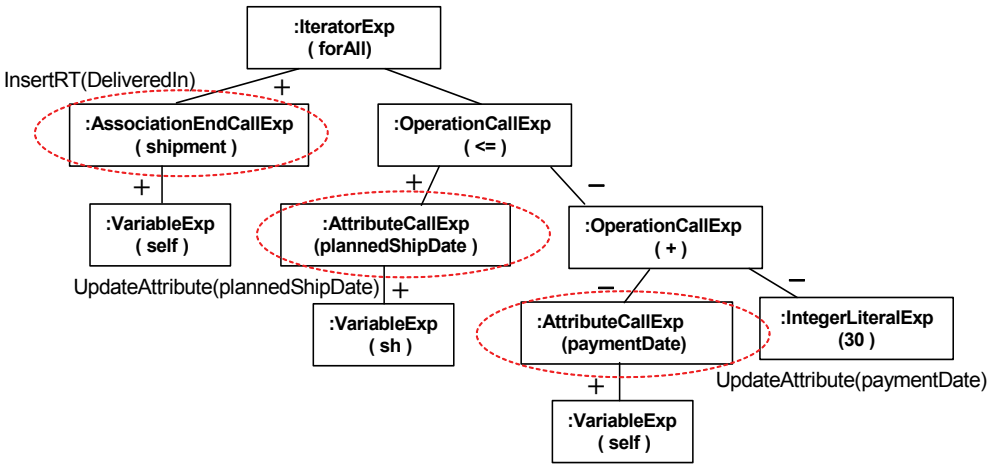


Figure 4.14. PSEs for *ValidShipDate*

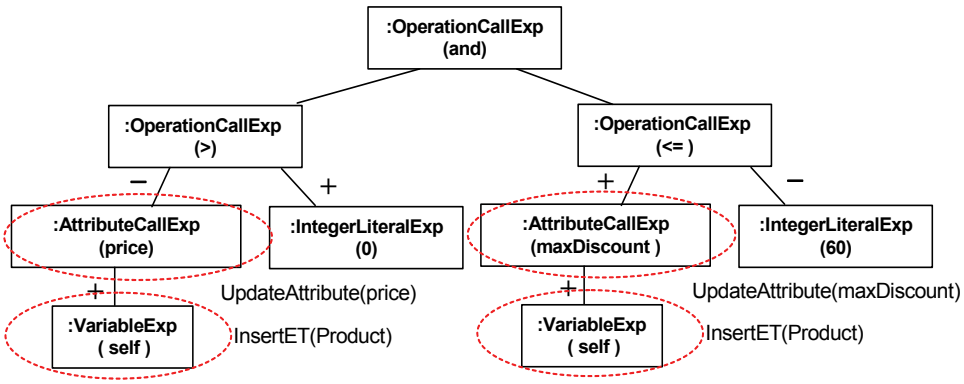


Figure 4.15. PSEs for constraint *CorrectProduct*

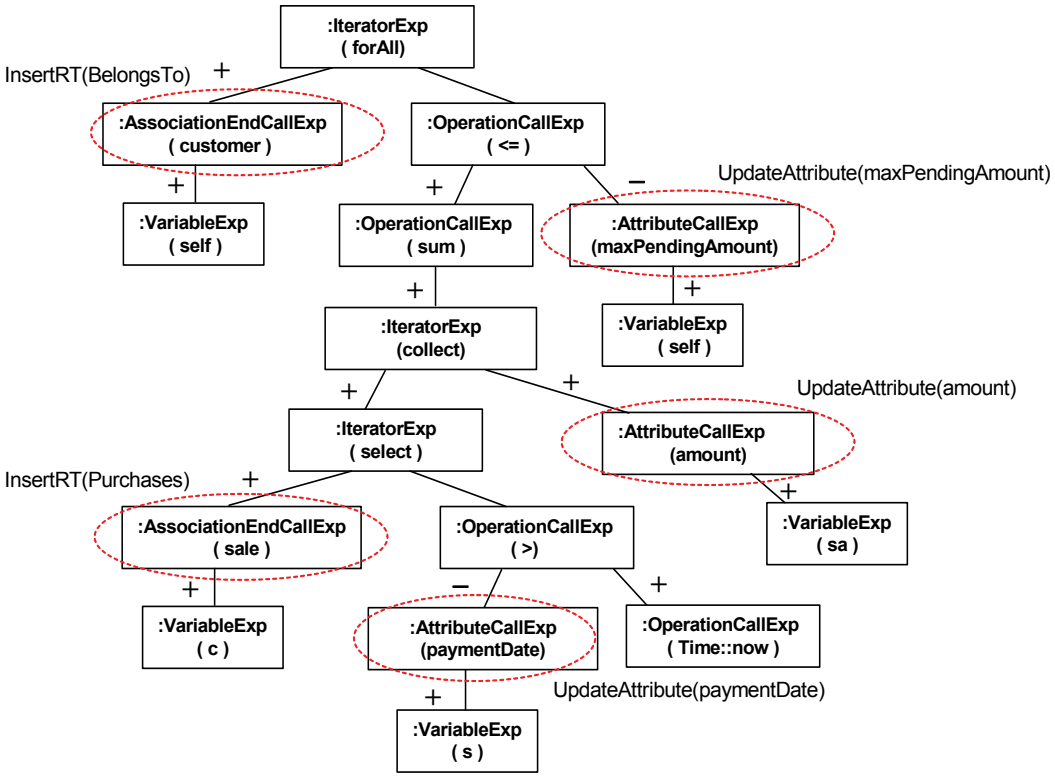


Figure 4.16. PSEs for constraint *NotTooPendingSales*

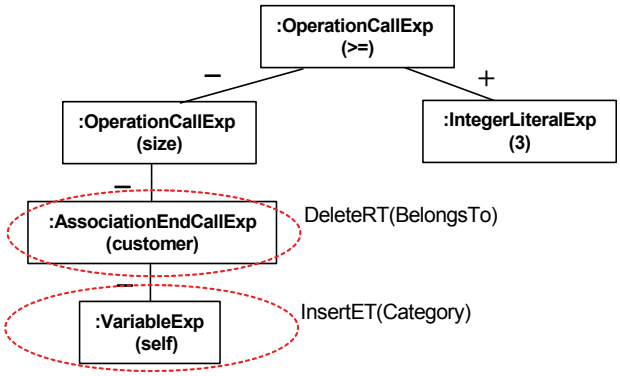


Figure 4.17. PSEs for constraint *AtLeastThreeCustomers*

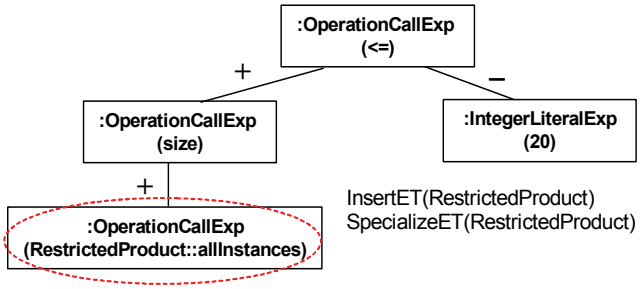


Figure 4.18. PSEs for constraint *NumberOfRestrictedProducts*

4.5 Summary

Table 4.7 summarizes the results of processing all example constraints to obtain their set of PSEs.

We would like to remark that the main benefit of determining the PSEs of a constraint is that it avoids a lot of unnecessary verifications when checking the state of the IB. With our results we may reduce the number of constraints to be considered during the integrity checking process since the events that can really violate the constraints are a small subset of all events affecting the elements referenced in the constraint body (which is a common strategy to compute the PSEs in other methods, see Chapter 9).

As an example, neither insertions nor deletions of sales and shipments may violate *ValidShipDate*. New sales (shipments) can only induce a constraint violation when we assign them to an existing shipment (sale), which implies the issue of an *InsertRT(DeliveredIn)* event type, already a PSE for the constraint. Similarly, the *DeleteET(Customer)* event type is not a PSE for the constraint *AtLeastThreeCustomers*. Deletion of customers not related with a category (though this is a situation not permitted in our particular running example) does not decrease the number of costumers per category, and thus, it may not induce the violation of the constraint). It is the deletion of the link (*DeleteRT(BelongsTo)* event type) between the customer and the category

(deletion that can be a preliminary before deleting the customer itself) what can violate the constraint.

Another relevant example is the constraint *NotTooPendingSales*. Even though this constraint involves three different entity types of the CS, only five event types are PSEs for the constraint.

Table 4.7. Summary of the PSEs for each constraint of the running example

Constraint	**PSE**
ValidShipDate	UpdateAttribute(paymentDate, Sale) UpdateAttribute(plannedShipDate, Shipment) InsertRT(DeliveredIn)
CorrectProduct	InsertET(Product) UpdateAttribute(price,Product) UpdateAttribute(maxDiscount,Product)
NotTooPendingSales	UpdateAttribute(maxPendingAmount,Category) InsertRT(BelongsTo) InsertRT(Purchases) UpdateAttribute(amount, Sale) UpdateAttribute(paymentDate,Sale)
AtleastThreeCustomers	DeleteRT(BelongsTo) InsertET(Category)
NumberOfRestrictedProducts	InsertET(RestrictedProduct) SpecializeET(RestrictedProduct)

5. Obtaining an appropriate representation for a constraint regarding a structural event

Due to the high expressiveness of the OCL language, the designer has different syntactic possibilities to express each integrity constraint, all of them semantically-equivalent. Two constraints c_1 and c_2 are semantically-equivalent when the IB satisfies c_1 iff also satisfies c_2.

Among all possible alternatives, the designer chooses one at definition time to represent the constraint (we do not assume the designer uses any particular criterion). However, it may happen that the selected representation is not the most adequate to efficiently check the constraint after an IB update (see the example in Section 1.1.5).

The aim of this chapter is to generate, for each PSE *ev* of a constraint *c*, an appropriate alternative representation of *c* regarding *ev*. We say that an alternative representation *ar* of a constraint *c* is an appropriate one with respect to a PSE *ev* when checking *c* expressed as *ar* after events of type *ev* requires taking into account less entities of the IB than the entities required to check *c* using any other alternative representation we may generate. The computation of the number of instances involved in the verification of a constraint after a given event is addressed in the next chapter. Obviously, for event types that are not PSEs for *c*, we do not need to generate an alternative representation of *c* since *c* does need to be checked after such events. Moreover, for some event types we could have several alternative representations all of them equally appropriate.

There exist two different ways to generate an alternative representation for a given integrity constraint: 1 - we can either replace the body of the constraint with an equivalent one (as it happens with the simplification rules of Chapter 3) or 2 – we may rewrite the constraint by using a different entity type as a context type for the constraint (for instance, using *Shipment* instead of *Sale* as a context type for the *ValidShipDate* constraint). We have already addressed the first possibility when simplifying the constraint according to the rules of Chapter 3. Therefore, when looking for alternative representations we just need to care about the possible alternatives due to context changes.

To obtain an appropriate alternative for a given PSE, we first select from the set of possible context types for the constraint the one that it is most suited with respect to that event type (i.e. the one that will result in an efficient incremental checking when verifying the constraint after the issue of events of that kind). Then, we generate a new version of the

constraint using as a context type, the selected type. The same type may be the best context type for several PSEs of the constraint.

At the end of the process, an integrity constraint *ic* with a set of PSEs set_{PSE} is split up into a set of alternative constraints $ic_1 ... ic_n$ (where n>=1), each one with its corresponding set of PSEs set_{PSEi} (where $set_{PSEi} \subset set_{PSE}$ and $\cup\ set_{PSEi} = set_{PSE}$). As an example, Figure 5.1 shows the generation of alternative constraints for the integrity constraint *ic*. The original constraint is already an appropriate alternative for the PSEs $PSE_1 ... PSE_f$, the alternative version ic_1 is the best representation for $PSE_{f+1} ... PSE_i$ and so forth.

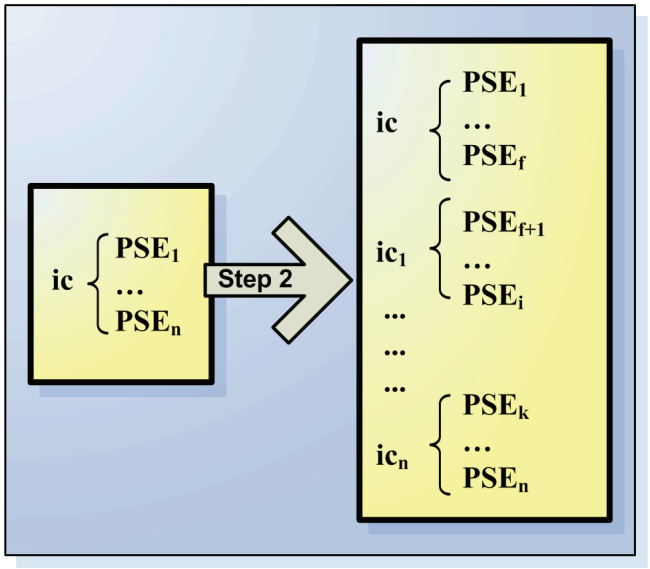


Figure 5.1 Generation of several alternatives for the constraint *ic*.

5.1 Deciding the best context type for a constraint with respect to a specific structural event

The best context to check an integrity constraint *ic* after applying an event of type *ev* over the IB is automatically drawn from the node where *ev* is assigned in the tree representing *ic*. We denote this node by $node_{ev}$.

For some constraints we may have several $node_{ev}$ nodes (i.e. two different nodes of the tree include the same event type *ev*, as an example see the event type *InsertET(Product)* in the *CorrectProduct* constraint; Figure 4.15). In such a case, we repeat the process explained in this chapter for each $node_{ev}$. This implies that we may end up (depending on the constraint tree structure) with two different alternatives of the constraint for the same event type, one for each different occurrence of the event type in the constraint tree. Then, as it will be explained in the next chapter, an issue of an event of that type will require verifying both alternatives since each event may induce the violation of a different set of entities in the IB, just as if they were different events. For instance, given a constraint stating that the salary of an employee must be lower than the salary of his/her boss, after updating the salary attribute of an employee *e*, if *e* is a boss we must check that the new salary is still

greater than the salary of the employees. If *e* is just an employee we must check that the new salary is still lower than the salary of his/her boss.

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

We define that a node *n* participates in a *collection condition* when *n* is used to compute an aggregate operator or a *select* iterator. Formally, when *n* verifies that:

{∃n'| n'∈ PathRoot(n) and ((n'.oclIsTypeOf(OperationCallExp) and n'.referredOperation ∈ {size, sum, count}) or (n'.oclIsTypeOf(IteratorExp) and n'.name='select'))}

where *PathRoot(n)* is defined as the ordered sequence of nodes encountered between *n* (the first node in the sequence) and the root of the tree (the last one). *ReferredOperation* is a navigation defined in the OCL metamodel (see Figure 4.2) that relates nodes of type *OperationCallExp* with the corresponding operation.

A node participates in an individual condition if it does not participate in a collection condition.

From Figure 4.14 we may see that all PSEs of *ValidShipDate* participate in individual conditions. The same with the PSEs of *CorrectProduct* (Figure 4.15). On the contrary, all PSEs of *AtLeastThreeCustomers* (Figure 4.17) and *NumberOfRestrictedProducts* (Figure 4.18) are included in collection conditions. In all cases, in their *PathRoot* we find a *size* operation. Finally, constraint *NotTooPendingSales* (Figure 4.16) has some events participating in individual conditions (*InsertRT(BelongsTo)* and *UpdateAttribute(maxPendingAmount)*) while the others participate in collection conditions. In particular, in the *PathRoot* of the $node_{ev}$ corresponding to the PSE *UpdateAttribute(amount)* we find a node representing the *sum* operation. In the *PathRoot* for the PSEs *InsertRT(Purchases)* and *UpdateAttribute(paymentDate)* we find a *select* iterator as well.

5.1.1. Best context type for events in individual conditions

Since individual conditions must hold for each individual entity restricted by the constraint, the best context type for a PSE included in an individual condition is the type of the entity (or relationship) modified (inserted/deleted/updated) by the PSE. Then, when an event of that type is issued, we may check the constraint just by applying the constraint body over the instance modified by the event.

Let *ev* be a PSE attached to the node $node_{ev}$. If $node_{ev}$ belongs to an individual condition then the best context is determined as follows:

- If *ev* is an *InsertET(ET)* event type, *ET* is the best context type.
- If *ev* is an *UpdateAttribute(attr,ET)* event type, the best context type is the type of the elements at the child node of $node_{ev}$. The type of the elements in a node *n* is retrieved with the expression *n.type* where *type* is a role defined in the OCL metamodel (see the association between the *OCLExpression* and *Classifier* metaclasses in Figure 4.1). For instance, the type of the elements of a node when the node is a *VariableExp* coincides with the type of the variable referred in the node. Similarly, the type of the elements in a *NavigationCallExp* node is the type of the participant entity type corresponding to the role traversed in the navigation. In general, this best context type coincides with *ET*, but when *ET* is part of an entity type taxonomy, the best context might be a subtype of *ET*.

  As an example assume the constraint "*context RestrictedProduct inv: self.price>10000*". Even though the *price* attribute is defined in the *Product* supertype, for *UpdateAttribute(price,product)* events the best context is the *RestrictedProduct* type (which is the type of the *self* variable, child node of the node where the update event is attached to) since only the restricted products must satisfy this constraint.
- If *ev* is an *SpecializeET(ET)* or *GeneralizeET(ET)* event type, the best context type depends on the kind of OCL expression represented by $node_{ev}$. When $node_{ev}$ is a *VariableExp*, the best context is the type of the referred variable (obtained as explained above). When $node_{ev}$ is an *oclIsKindOf* or *oclIsTypeOf* expression the best context depends on the type of the element/s of the child node, computed as commented before. For instance, given the constraint *context A inv: self.r1->forAll(v:X| v.oclIsTypeOf(Y))*, the best context for the specialization and generalization events is *X*, since every individual *X* instance (related with an *A* instance) must verify the *oclIsTypeOf* condition.
- If *ev* is an *InsertRT(RT)* event type, the best context type depends on the multiplicities of *RT*. When *RT* is binary relationship type and, at least, one of its roles has a maximum multiplicity of 1, we define as the best context the type of the participant playing the opposite role. Otherwise, *RT* is defined as the best context. In fact, both alternatives present the same efficiency (when using as a context the participant type, we may still obtain the single *RT* instance updated by the event since that participant type participates at most in one relationship). However, avoiding, when possible, to select *RT* as the best context permits to skip its reification (which later on will be necessary for all relationship types selected as best context types), and thus, to reduce the complexity of the processed CS.

It is not difficult to see why the best context type for a PSE included in an individual condition is the type of the modified instance. Given a constraint *c* defined over a type *t*

and with a node *n* (included in an individual condition) marked with an event *ev* affecting instances of type *t'*, the cost of evaluating the individual condition after applying an event of type *ev* when *c* is defined using *t'* as context type requires checking a single instance of *t'* (the modified one). Instead, evaluating the individual condition when *c* is defined over *t* requires first to navigate from *t* to the related *t'* instances. As a result of this navigation we may obtain several *t'* instances which results in a poorer efficiency in the integrity checking since then we are not considering just the modified one.

Note that, when determining the best context types for individual conditions, we have not mentioned the event types *DeleteRT* and *DeleteET*. Due to the transformation rules presented in Chapter 3, these events always appear in collection conditions. Since individual conditions must be verified for each existing entity individually, entities removed from the IB are not affected by these events. For instance, *CorrectProduct* constraint only restricts existing products, and thus, the removal of a product can never violate the constraint.

In our running example, the constraint *ValidShipDate* may be violated by three different structural events (Figure 4.14), all of them included in individual conditions: *InsertRT(DeliveredIn)*, *UpdateAttribute(plannedShipDate,Shipment)* and *UpdateAttribute(paymentDate,Sale)*. Their best context types are therefore *DeliveredIn* (the relationship type modified by the event), *Shipment* (the type of the *sh* variable appearing as a child node of the *AttributeCallExp* node labeled with the *UpdateAttribute(plannedShipDate)* event type) and *Sale* (the type of the *self* variable appearing as a child node of the *AttributeCallExp* for the *paymentDate* attribute) respectively. By a similar reasoning we obtain the best context types for the rest of PSEs participating in individual conditions (see also Table 5.1):

- *CorrectProduct*: the best context for all the PSEs is the *Product* entity type. Note that, for this constraint, both occurrences of the PSE *InsertET(Product)* generate the same best context type since both participate in an individual condition.

- *NotTooPendingSales*: the best context for the PSE *UpdateAttribute(maxPendingAmount)* is *Category*. The best context for the PSE *InsertRT(BelongsTo)* is the *Customer* entity type because of the multiplicities of the *BelongsTo* relationship type (the other end of the *BelongsTo* relationship type presents a maximum multiplicity of 1).

5.1.2. Best context type for events in individual conditions

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 and not by each single entity affected by the event. Thus, expressing the constraint using as a context type the type of the modified entity or relationship is not useful because, after every modification, the whole collection must be recomputed again and the other entities in the collection must

also be taken into account. For instance, in a constraint like *self.X->collect(attr)->sum()>val* (where *X* represents a set of navigations, *attr* an attribute and *val* a constant value) we do not force each *entity* $\in$ *X* to have a value in *attr* greater than *val*, we only require the sum of *attr* for the whole set of entities in *X* to be greater than *val*.

Therefore, to facilitate the verification of the constraint we should use as context type the type of the entity used as a starting point to obtain the collection of entities that must verify the collection condition. For instance, an *InsertRT(Purchases)* event (i.e. the assignment of a sale *s* to a customer *c*) may violate the constraint *NotTooPendingSales* (Figure 4.16). In this case, the *maxPendingAmount* condition must be satisfied by the set of sales of each customer; thus, after assigning *s* to the customer *c*, it is enough to check the set of sales of *c* to ensure that the IB still satisfies the constraint. Therefore, we may say that *c* is the entity used as a starting point to obtain the collection of entities (i.e. the set of sales) that must verify the *maxPendingAmount* collection condition. The type of the entity *c*, *Customer*, is the best context type for all PSEs included in this collection condition.

To automatically determine the best context type for collection conditions (i.e. the type acting as a starting point of the condition) we need, first, to define the auxiliary operation *PathVar(node)*. Given a node *n*, *PathVar(n)* is defined as the ordered sequence of nodes encountered between *n* (the first node) and the node representing the *self* variable or the *allInstances* operation (the last node) of the subtree to which *n* belongs. More precisely, *PathVar(n)* is computed as follows:

- The first node in the path is *n*.
- For each node *n* included in the path we also include its child node. When the node has two children and the node represents an iterator expression (as a *forAll,* a *collect* or a *select*) we include the left child in the path (the structure of the iterator body is irrelevant to compute the best context type). Otherwise, see the special treatment for collection operations at the end of this section.
- When a node *n* included in *PathVar* represents a variable other than *self* (i.e. variables used in *select, collect* or *forAll* iterators), we add as a left child 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 Figure 4.1).

Given an integrity constraint *ic*, a PSE *ev* and the path returned by $PathVar(node_{ev})$, then, the node origin of a collection condition $node_{or}$ is defined as:

- The left child of the first node $n \in$ PathVar($node_{ev}$) representing a *forAll* iterator, when a *select* iterator is not encountered between *n* and the last node in $PathVar(node_{ev})$.

- Otherwise, the last node in *PathVar(node$_{ev}$)* (i.e. the node representing the *self* variable or an *allInstaces* operation). Note that this is always the case when the last node is the *allInstances* operation (constraints with a *forAll* iterator after an *allInstances* operation have been simplified in Chapter 3 by means of reexpressing the constraint in terms of the *self* variable).

Let *ev* be an event included in a collection condition. If *node$_{or}$* does not represent the *allInstances* operation the type of the entities at *node$_{or}$* is then returned as the best context type. When *node$_{or}$* is an *OperationCallExp* referencing the *allInstances* operation, the current context type of the constraint is returned as the best context (in fact, other types could serve as well as a best context in this case, since the result of the *allInstances* operation is independent from the context type of the constraint, it only depends on the entity type where the operation is applied over).

Note that choosing as best context type a type different from the type *t* of the entities at *nodeor* does not improve the efficiency of the integrity checking. Given a PSE *ev* that modifies instances of type *t'* and included in a collection condition *col*, to check *col* after an event of type *ev*, we are forced to navigate from the modified *t'* instance to the related instance/s of type *t* to start evaluating *col* (since *t* is the starting point to obtain the collection of entities that must satisfy *col*). Moreover, when processing the schema as shown in the next chapter, using *t* as the best context type avoids some redundant verifications when the modification of the IB consists of several events affecting a set of *t'* instances (or any other instances of a type *t''* also involved in the collection condition) related with the same *t* instance.

As an example, Figure 5.2 shows the *PathVar* and the *node$_{or}$* values when *node$_{ev}$* is the node corresponding to the PSE *UpdateAttribute(paymentDate,Sale)* in the constraint tree of *NotTooPendingSales*. The numbers indicate the order in which the nodes are included in the *PathVar* sequence of nodes. In this example, the node representing a navigation through the customer role is the origin of the collection condition where this PSE participates (the *maxPendingAmount* condition must be satisfied fore each customer). Therefore, *Customer* (which is the type of the entities at that node) is the best context type for this event. *Customer* is also the best context for the other PSEs included in the same collection condition (updates of *amount* attributes and inserts of *Purchases* relationships; the other PSEs of the constraint are included in individual conditions, see the previous section).

The *PathVar* for the PSEs of constraints *AtLeastThreeCustomers* (Figure 4.17) and *NumberOfRestrictedProducts* (Figure 4.18) are much shorter and they just involve one or two nodes. In *AtLeastThreeCustomers* the best context for both PSEs (*DeleteRT(BelongsTo)* and *InsertET(Category))* is *Category*, which is the type of the *self* variable (*node$_{or}$* of the collection condition). For the second constraint the best context is

RestrictedProduct, already the original context type of the constraint since $node_{or}$ is the node corresponding to the *allInstances* operation.

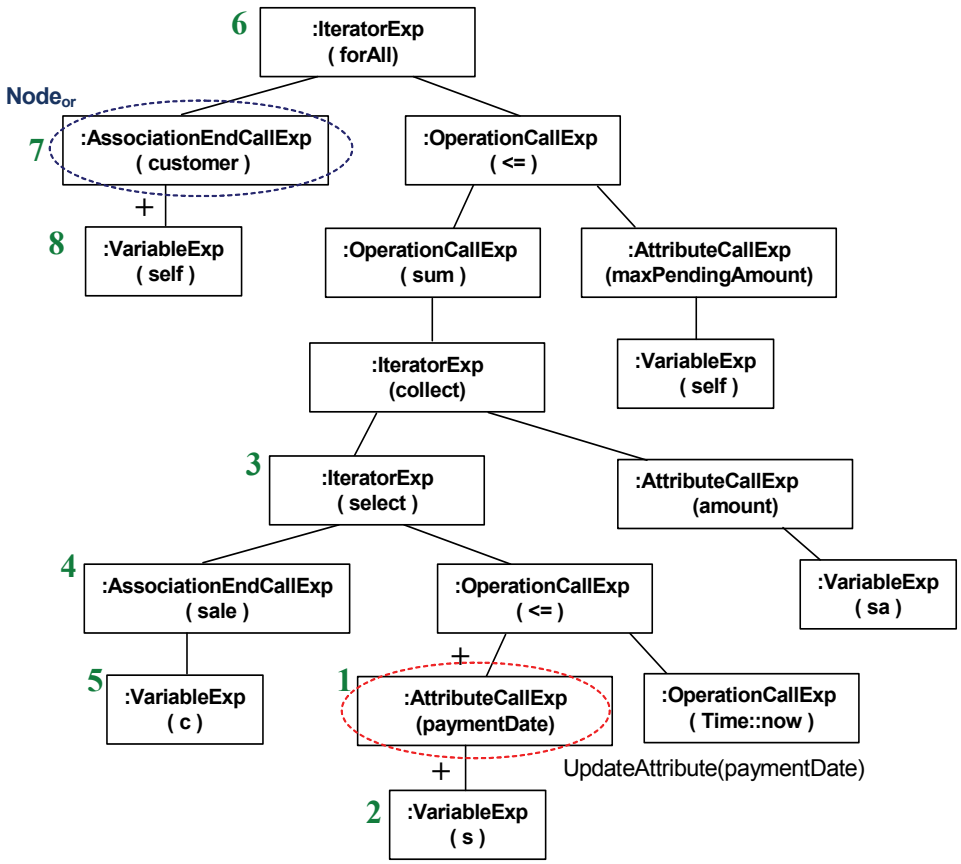


Figure 5.2. $Node_{or}$ for constraint *NotTooPendingSales*

For some constraints we do not have a single origin for the collection condition. This happens when the collection of entities *col* affected by the collection condition is obtained by means of the union (or intersection, difference,…) of several other collections $col_1..col_n$. Then, we have different possible *PathVars*, one for each collection col_i. These different *PathVar* sequences are obtained taking a different path every time we find a node representing one of the collection operations *union*, *intersection*, *difference*, *symmetric difference* or *product*.

In this case, to determine the best context for a PSE *ev*, we apply the previous process to each *PathVar*($node_{ev}$). Note that different *PathVar* sequences may result in different best contexts for the same PSE. However, each one of these best contexts is only relevant for the subset of *PathVar* sequences resulting in that best context. This information is stored together with the PSE in the constraint tree. We will take this information into account when computing the instances to check the constraint after the issue of the PSE in the next chapter.

As an example, consider the example CS of Figure 5.3. Over this example schema we could define the following constraint:

> ***context*** *Department* ***inv*** *WorkOverload: self.employee->forAll(e\| e.project->union(self.project)->collect(p\|p.budget) ->sum()<10000)*

stating that all employees of a department must verify that the sum of the budgets of his/her projects plus the budget of the projects assigned to the department as a whole must be lower than 10000 euros.

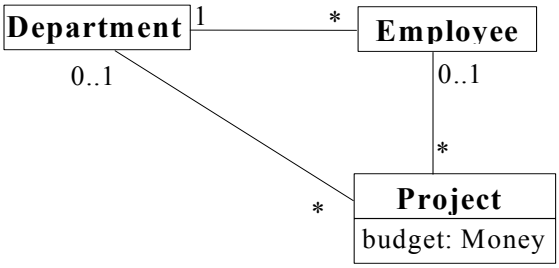


Figure 5.3. Example for collection conditions

One of the PSEs for this constraint is the event type *UpdateAttribute(budget,Project)*. When computing the *PathVar* for this event we have two possible solutions (Figure 5.4). The first one is the path corresponding to the *self.project* subexpression (nodes *{1,2,3,4,5B,6B}*) while the other corresponds to the *self.employee->forAll(e.project…)* subexpression (nodes *{1,2,3,4,5A,6A,7A,8A,9A}*).

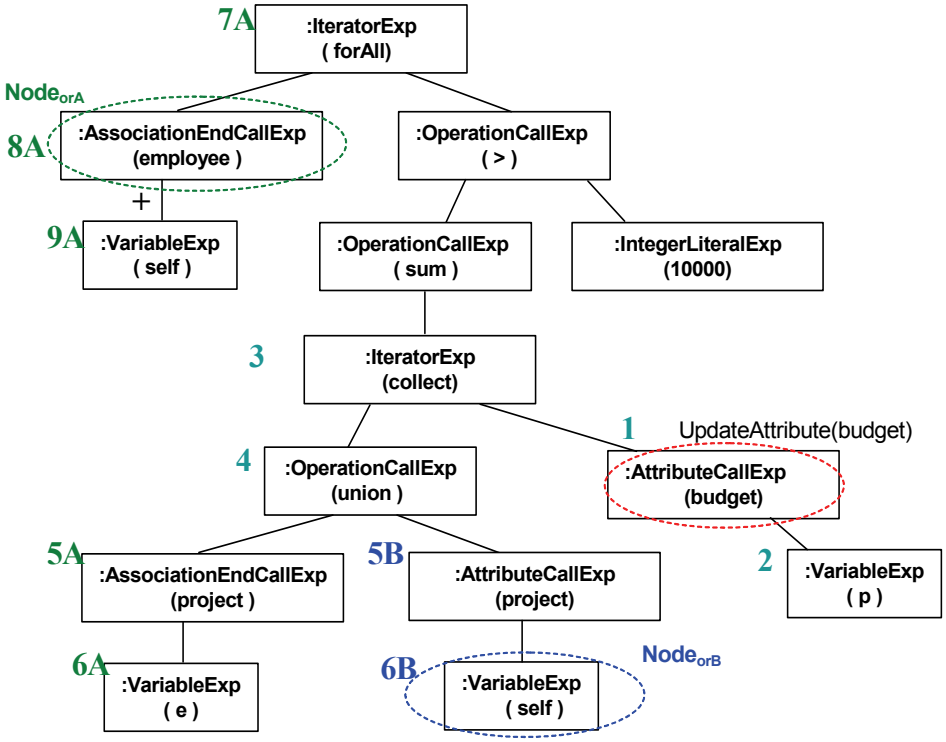


Figure 5.4. *PathVars* for the *UpdateAttribute(budget)* event type

In the first case, *Department* is returned as the best context type while in the second one *Employee* is the best context. This means than when issuing the update event over a project *p*, we will have to check:

1- The constraint as defined over *Department* for the department *d* related with *p* (if any). In this scenario we end up verifying that each employee working in *d* verifies the *budget* condition (due to the budget change of *p* some employee in *d* may violate the constraint).

2- The constraint as redefined over *Employee* for the employee *e* related with *p* (if any). In this case, we just need to check that *e* does not violate the budget condition. Other employees working in the same department are not affected.

5.2 Redefining a constraint in terms of a new context type

Once we have determined the best context type for each PSE *ev* (see Table 5.1) we must redefine the original constraint in terms of these new best contexts. Each new constraint representation must be semantically-equivalent to the original one, at least with respect to the particular event type *ev*. Two constraints c_1 and c_2 are semantically-equivalent with respect to a PSE when the application of an event of that type over a consistent state of the IB results in a new state of the IB that satisfies c_1 iff it also satisfies c_2.

Table 5.1. Summary of the best contexts for all PSEs of the constraints of the running example

Constraint	**PSE**	**Individual/ Collection**	**Best Context**
ValidShipDate	UpdateAttribute(paymentDate, Sale)	Individual	Sale
	UpdateAttribute(plannedShipDate, Shipment)	Individual	Shipment
	InsertRT(DeliveredIn)	Individual	DeliveredIn
CorrectProduct	InsertET(Product)	Individual	Product
	UpdateAttribute(price,Product)	Individual	Product
	UpdateAttribute(maxDiscount,Product)	Individual	Product
NotTooPendingSales	UpdateAttribute(maxPendingAmount,Category)	Individual	Category
	InsertRT(BelongsTo)	Individual	Customer
	InsertRT(Purchases)	Collection	Customer
	UpdateAttribute(amount, Sale)	Collection	Customer
	UpdateAttribute(paymentDate,Sale)	Collection	Customer
AtleastThreeCustomers	DeleteRT(BelongsTo)	Collection	Category
	InsertET(Category)	Collection	Category
NumberOfRestricted Products	InsertET(RestrictedProduct)	Collection	RestrictProd
	SpecializeET(RestrictedProduct)	Collection	RestrictProd

The first step is to decide which parts of the initial constraint have to be redefined. Since both constraints only need to be equivalent with regards to the particular PSE *ev*, when transforming the constraint to the new context type we need not worry about those literals of the original constraint that cannot be violated by events of type *ev*.

Given that the body of the initial integrity constraint *ic* follows the pattern L_1 *and* L_2 *and ... and* L_n (as *CorrectProduct* in Figure 4.15) and that *ev* can only induce a change in the truth value of L_1, the redefined constraint *c'* does not need to include the verification of the literals $L_2 ... L_n$. Since *ev* does not affect them, if those literals were true before *ev* was executed (we always assume, as usual, that the IB is in a consistent state prior to its modification) they will still hold after its execution. When they do not hold it is because some other event, *ev'*, has been applied. The constraint in charge of verifying the IB after events of type *ev'* will take care of this possible violation.

To prune the parts of the constraint tree that are irrelevant to an event *ev* (and thus, those parts that can be discarded during the redefinition of the constraint) we apply the following process. Let *ev* be an event attached to a node $node_{ev}$. A node n_{and} representing an *AND* condition may be pruned if:

$\{n_{and} \in PathRoot(node_{ev})$ *and* $\neg\exists n' \mid n' \in PathRoot(n_{and})$ *and* $n'.oclIsTypeOf(IteratorExp)$ *and* $n'.name=$*"select"*$\}$.

N_{and} nodes are replaced with the child node $n_{child} \in PathRoot(node_{ev})$. Consequently, the other child of n_{and} (i.e. the other condition) is removed from the tree.

As an example, consider the tree corresponding to the constraint *CorrectProduct* (Figure 4.15). Figure 5.5 shows the simplified trees for each PSE. For events of type *UpdateAttribute(price,Product)* we just need to verify the literal *self.price>0*. For the PSE *UpdateAttribute(maxDiscount,Product)* it is enough to consider the literal *self.maxDiscount<=60*. Note that the PSE *InsertET(Product)* appears in both literals, and thus, the complete original constraint needs to be verified after events of that type.

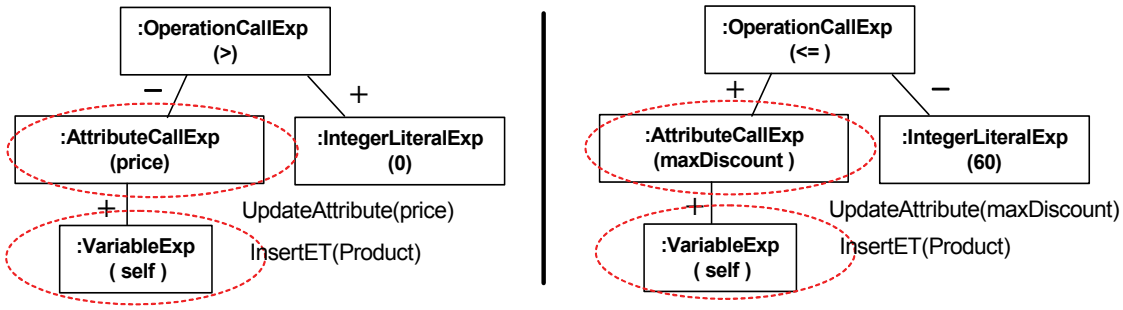


Figure 5.5. Simplified trees for the PSEs of constraint *CorrectProduct*

Afterwards, when the best context type *ct'* determined for the event *ev* is different from the initial context type *ct* of the constraint *c* (*context ct inv c: X)*, the second step is to redefine the relevant literals of *c* (i.e. the ones not pruned in the previous step) in terms of *ct'* to improve the efficiency of the integrity checking. The new constraint representation *c'* may be obtained applying the following transformation:

context ct' inv c': self.r_1*.*r_2*. ...* r_n*->notEmpty() implies self.*r_1*.*r_2*. ...* r_n *->forAll(v|X')*

where *X'* results from replacing all occurrences of *self* in *X* with *v* during the transformation to *c'* and $r_1 ... r_n$ are a sequence of roles that permit to navigate from *ct'* to *ct*. Obviously, in the CS there may exist several different sequences of navigations that reach *ct* from *ct'*. Each different sequence would generate a different transformation. However, for our purposes, we need to navigate traversing the roles opposite to the ones used in *c* to reach *ct'* (we know for sure that *ct'* is reached in *c*, otherwise *ct'*, *ct'<>ct*, would never be selected as the best context type for the PSE *ev*). More precisely, we need to navigate using the opposite roles of the roles appearing in $PathVar(node_{ev})$, being $node_{ev}$ the node where *ev* is attached to.

Therefore, given that in *c*, we navigate from *ct* to *ct'* using the sequence of roles $r_1'.r_2'.....r_n'$, our transformation does the reverse navigation, and thus, in the transformation r_1 is the opposite role of r_n', r_2 is the opposite role of r_{n-1}' and so forth.

Note also that when a role r_i' is a navigation from a subtype *sub* to another entity type using a relationship type defined in a supertype of *sub*, we must add the subexpression "*select(oclIsKindOf(sub))*" (or "*any(oclIsKindof(sub))*" when the result must be a single object) to the corresponding opposite role r_j to ensure that only the instances of the subtype *sub* are retrieved by the navigation. Similarly, we will need to add an explicit cast (using the *oclAsType* operator) to the particular subtype in order to access its attributes and roles.

We would like to remark that *c* and *c'* are semantically-equivalent since both apply the same condition (the condition *X*) to the restricted entities and apply it over all relevant entities of *ct*. Every time we evaluate *c'* over an instance of *ct'* we reach (and verify) some of the instances of *ct* restricted by *c* in the original constraint. It may happen that some instances of *ct* are never reached (if they are not related to any instance of *ct'*) and remain unchecked after the verification of the new state of the IB because of the application of an event of type *ev*. However, those instances are not possibly affected by events of type *ev* and thus they are irrelevant to the integrity checking due to events of type *ev*.

As an example, the constraint *ValidShipDate* (*self.shipment->forAll(sh| sh.plannedShipDate<=self.paymentDate+30*) must be redefined over *Shipment* to verify the IB after events of type *UpdateAttribute(plannedShipDate,Shipment)*. According to the previous transformation the redefined constraint $ValidShipDate_2$ is:

context *Shipment* ***inv*** *ValidShipDate$_2$: self.sale->notEmpty() implies self.sale->forAll(s| s.shipment->forAll(sh| sh.plannedShipDate<=s.paymentDate+30))*

where *sale* is the role that permits to navigate from a shipment to its related sales. The body of the first *forAll* coincides with the body of the original constraint once all references to the *self* variable have been replaced with references to the variable *s* of the *forAll* iterator expression.

We provide some rules to simplify the body of the new constraint *c'*. After simplifying the new constraint with this set of rules we can also apply the simplification rules of Chapter 3 to obtain a better result.

1. *self.r_1.r_2.…r_n->notEmpty()* → *true*, if the multiplicity of *self.r_1.r_2. … r_n* is at least one, i.e. if all the minimum multiplicities of *r_1.r_2. … r_n* are at least one. In such a case, we know for sure that the navigation will return a non-empty set, and thus, that the evaluation of the *notEmpty* operation will return a true value.

2. *self.r_1.r_2. … r_n->forAll(v|X)* → *X* (where all the occurrences of *v* in X are replaced with *self.r_1.r_2. … r_n*), if the multiplicity of *self.r_1.r_2. … r_n* is at most one, i.e. if all the maximum multiplicities of *r_1.r_2. … r_n* are at most one. Then, the *forAll* iterator is no longer necessary.

3. *self.r_1.r_2.….r_i.r_j… r_n->forAll(v|X)* → *self.r_1.r_2. ….r_{i-1}.r_{j+1}… r_n->forAll(v|X)*, when r_i and r_j are the two roles of the same binary association (see Figure 5.6). When the maximum multiplicity of r_j is one, the set of objects at r_j are the same than those at r_{i-1}, and thus, the navigations r_i and r_j are redundant (in this case the rule is applicable even if there is not a *forAll* iterator after r_n). Otherwise, we may have more objects at r_j, and, in general, this entails that these additional objects are not verified in the right hand expression of the rule. However, we can still apply the rule if the minimum multiplicity of all opposite roles from r_1 to r_{i-1} is at least one, since then, those objects must be related with a (different) instance of the context type, and thus, they will be checked when evaluating that instance. When r_i may have a zero minimum multiplicity, after the simplification we could be enforcing some objects not affected in the original constraint. Note that in such a case, the *notEmpty* clause of the general transformation rule will not be simplified by rule 1, and thus, we ensure that those objects will never be evaluated.

4. *self.r_1.r_2… r_n->forAll(v1,v2|X)* → *self.r_1.r_2… r_n->forAll(v2|X)* (where all occurrences of v_1 in *X* are replaced with *self*), if the type of the objects at r_n coincides with the type of the *self* variable and all the navigations from r_1 to r_n are redundant. This rule is similar to the rule to simplify the *allInstances* operation presented in Chapter 3. We cannot completely simplify the *forAll* iterator since the

constraint requires a comparison between an object of type *t* and a set of other objects of the same type *t*.

5. *self.r_1.r_2...r_i->forAll(v| v.r_j...r_n->forAll(v_2|X))* → *self.r_1.r_2....r_i.r_j...r_n->forAll(v_2|X))*, when *X* does not contain any reference to *v*. The two expressions are equivalent since in both we apply the condition *X* over the objects obtained at r_n. When *X* contains references to *v* they must be replaced with the expression $v_2.r_n'...r_j'$ where $r_n'..r_j'$ represent the opposite roles of $r_n...r_j$ (for instance, r_n' is the opposite of r_n). Note that when, the multiplicity of some r_k (where $j>=k<=n$) is greater than 1 then the left hand side must be replaced by the expression *self.r_1.r_2…….r_i.r_j ...r_n ->forAll(v_2| $v_2.r_n'...r_j'$->forAll(v_3| X))* where references to *v* in the original constraint are replaced with v_3. This later case only makes sense when r_i and r_j are the two roles of the same relationship type, which implies that the new expression can be simplified with rule 3 afterwards.

6. Given a reified entity type *RET* (see Figure 5.7): *X.ret.b.Y* → *X.b.Y*. According to the OCL standard we can navigate to *B* either by accessing first the reified type or directly using the role *b* of *B*. In both cases we obtain the same set of entities.

7. Given a reified entity type *RET*: *context RET inv: self.a.b.r_1..r_n->forAll(X)* → *context RET inv: self.b.r_1..r_n->forAll(X)*. Even though, given an entity *e* of the *RET* type, the right hand side expression may verify less entities than the left hand expression (since *e.b* may return less entities than *e.a.b*) those objects will be verified when evaluating other entities of *RET* .

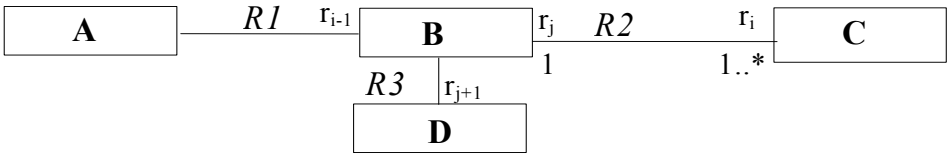


Figure 5.6. Abstract example schema for rule 3

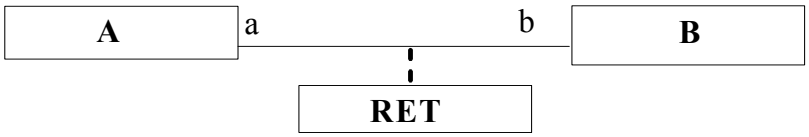


Figure 5.7. Example of a reified entity type

All rules can be applied regardless the other subexpressions forming the constraint body except for rule 3 when the constraint body is a disjunction of literals following the pattern: *self.r_1..r_n->forAll(X) or ... or self.r_1...r_n->forAll(Y)* (where $r_1..r_n$ represent exactly the same sequence of navigations in the disjunctions). In this case, only the literal/s affected by the event for which the generated constraint is the appropriate alternative may be simplified. Assuming that the event is included in the *X* condition, the simplified constraint would be:

X or self.r1…rn->forAll(Y). Note that the body "*X* or *Y*" would not be a correct solution since the original constraint does not state that all entities at r_n must satisfy *X* or *Y*, it states that either all entities at r_n satisfy *X* or all entities satisfy *Y*. Then, if an event over an entity *e* makes that *e* evaluates *X* to false, we need to verify that at least all entities (and not just *e*) verify Y.

Figures 5.8-5.10 show the new alternative constraint representations required as a result of the computation of the best context types for the PSEs of all constraints of the running example (see Table 5.1). The new generated constraints representations are: $ValidShipDate_2$ (*ValidShipDate* defined over *Shipment*), $ValidShipDate_3$ (*ValidShipDate* defined over *DeliveredIn*) and $NotTooPendingSales_2$ (*NotTooPendingSales* defined over *Customer*). For each constraint redefinition we show the initial result of its transformation plus the sequence of rules our method applies in order to obtain the final body of the constraint. For the sake of simplicity when applying rule 1 we have applied at the same time the sequence of rules: *true implies X* → *not true or X* → *false or X* → *X*, as defined in Chapter 3.

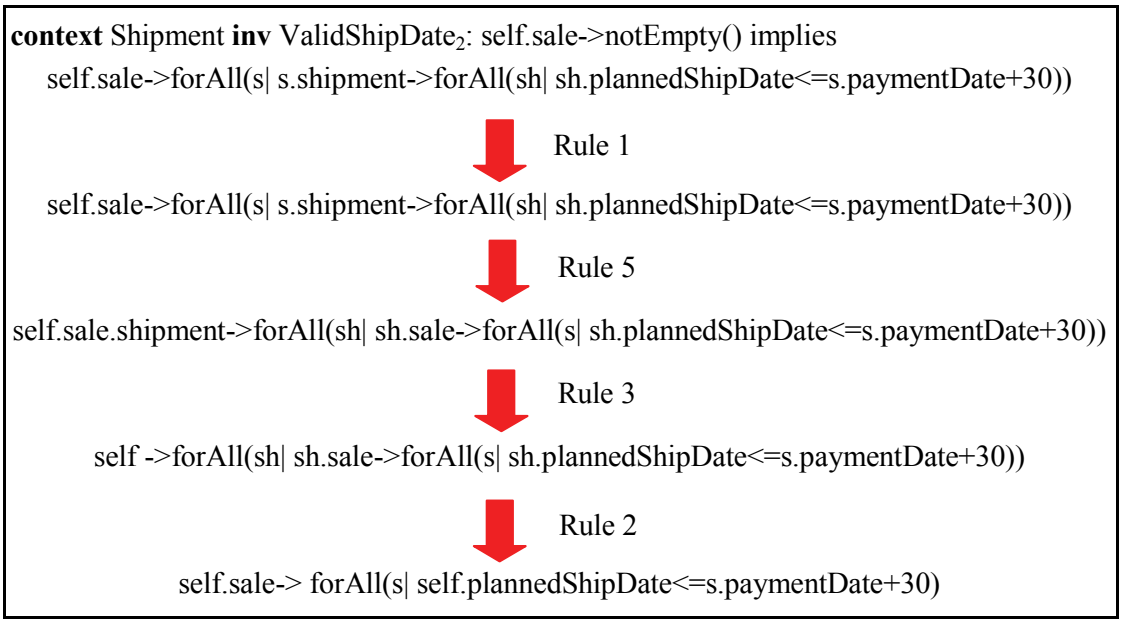


Figure 5.8 Simplification of *ValidShipDate* when defined over *Shipment*

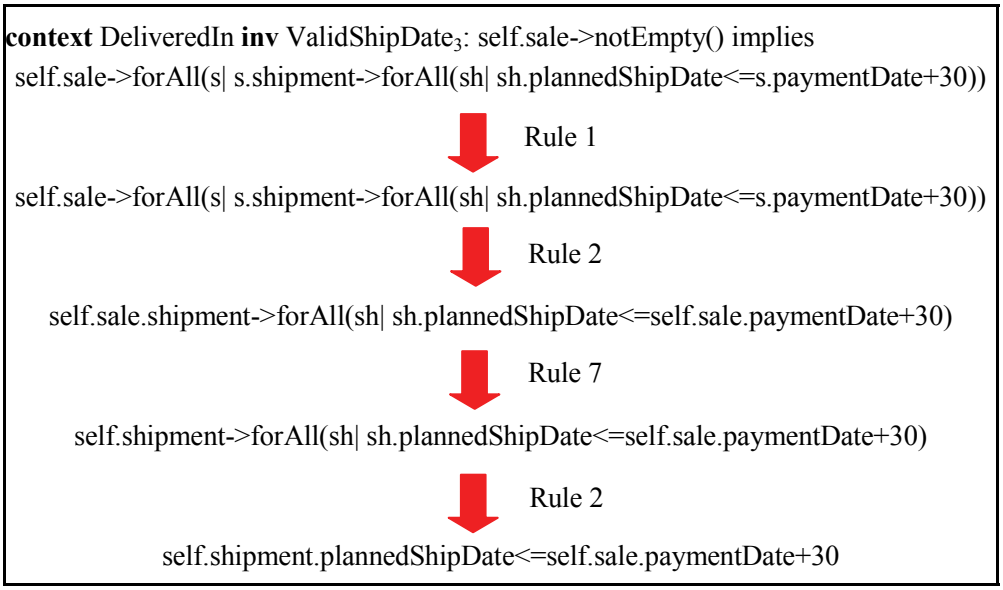


Figure 5.9 Simplification of *ValidShipDate* when defined over *DeliveredIn*

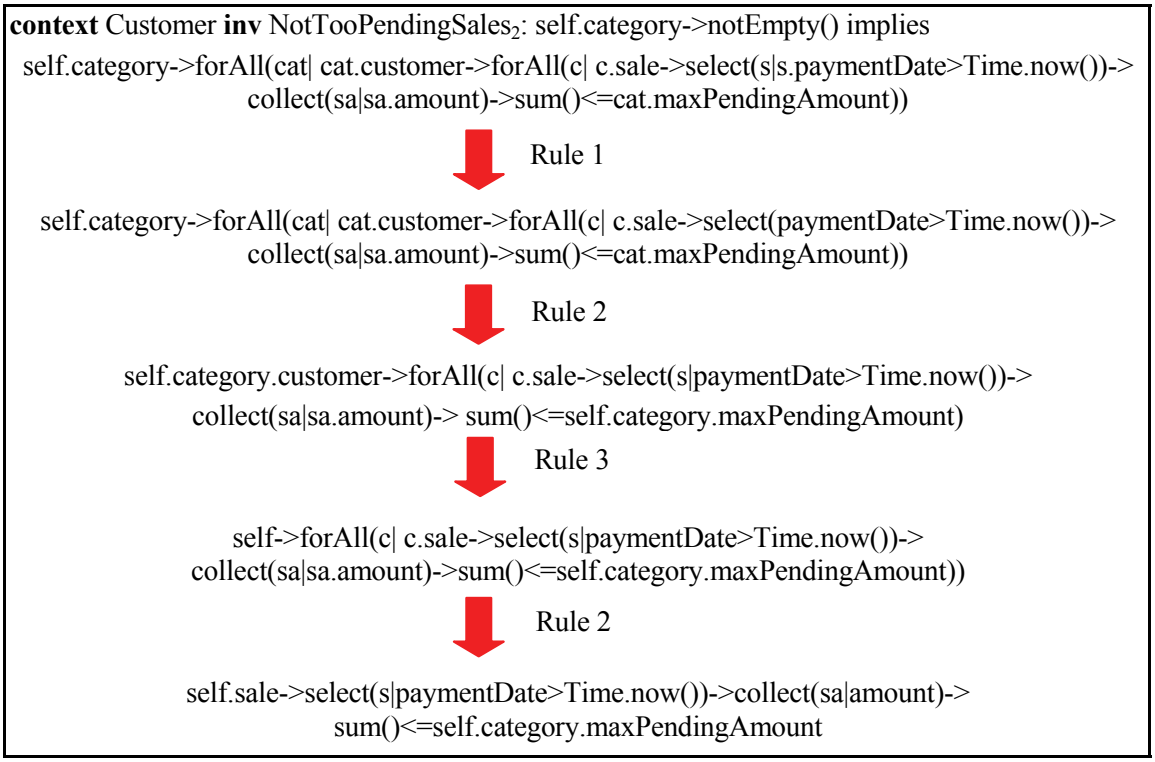


Figure 5.10 Simplification of *NotTooPendingSales* when defined over *Customer*

5.3 Summary

Table 5.2 summarizes the final results obtained after applying the whole process explained in this chapter over all constraints of our running example.

Although the number of different constraints has increased, we have improved the efficiency of the integrity checking since each constraint is specialized to be efficiently verified after the issue of some particular types of structural events.

Table 5.2. Summary of the best constraint representations for each PSE

Constraint	PSE	Best alternative
ValidShip Date	UpdateAttribute(paymentDate, Sale)	**context** Sale **inv** ValidShipDate: self.shipment->forAll(sh| sh.plannedShipDate<=self.paymentDate+30)
	UpdateAttribute(plannedShipDate,Shipment)	**context** Shipment **inv** $ValidShipDate_2$: self.sale->forAll(s| self.plannedShipDate<=s.paymentDate+30)
	InsertRT(DeliveredIn)	**Context** DeliveredIn **inv** $ValidShipDate_3$: self.shipment.plannedShipDate<=self.sale.paymentDate+30
Correct Product	InsertET(Product)	**context** Product **inv** CorrectProduct: self.price>0 and self.maxDiscount<=60
	UpdateAttribute(price, Product)	**context** Product **inv** $CorrectProduct_2$: self.price>0
	UpdateAttribute(maxDiscount,Product)	**context** Product **inv** $CorrectProduct_3$: self.maxDiscount<=60
NotTooPendingSales	UpdateAttribute(maxPendingAmount, Category)	**context** Category **inv** NotTooPendingSales: self.customer->forAll(c| c.sale->select(s| s.paymentDate>Time.now())->collect(sa|sa.amount)->sum()<=self.maxPendingAmount)
	InsertRT(BelongsTo)	**context** Customer **inv** $NotTooPendingSales_2$: self.sale->select(s| s.paymentDate>Time.now())->collect(sa|sa.amount)->sum()<=self.category.maxPendingAmount
	InsertRT(Purchases)	
	UpdateAttribute(Amount, Sale)	
	UpdateAttribute(paymentDate,Sale)	
Atleast Three Customers	DeleteRT(BelongsTo)	**context** Category **inv** AtLeastThreeCustomers: self.customer->size()>=3
	InsertET(Category)	
NumberOf Restricted Products	InsertET(RestrictedProduct)	**context** RestrictedProduct **inv** NumberOfRestrictedProducts: RestrictedProduct.allInstances()->size()<=20
	SpecializeET(RestrictedProduct)	

6. Evaluating the constraints over the relevant instances

A constraint *c* defined over a context type *t* must be satisfied by all instances of *t*. Nevertheless, it is not necessary to check that all instances of *t* satisfy *c* every time an event of type *ev* (where *ev* is one of the PSE for *c*) is applied over the IB. Only those instances relevant to (i.e. affected by) the issue of the event need to be considered. This incremental checking (incremental since we reason from the applied events in order to reduce the number of instances of *t* to consider) improves the efficiency of the integrity checking process.

Intuitively, the relevant instances are those instances of *t* that are related (directly or indirectly) to the instance modified by the event. The state of the other instances of *t* has not changed, and thus, they still satisfy *c* (we assume that the IB was in a consistent state prior the issue of the event).

As an example, consider again the *NotTooPendingSales*$_2$ constraint (one of the constraints generated in the previous chapter, see its tree representation in Figure 6.1). After the update of the *amount* of sale s_3 and the insertion of a new purchase for customer c_1, the relevant instances are just c_1 and s_3.*customer* (i.e. the customer that purchased sale s_3).

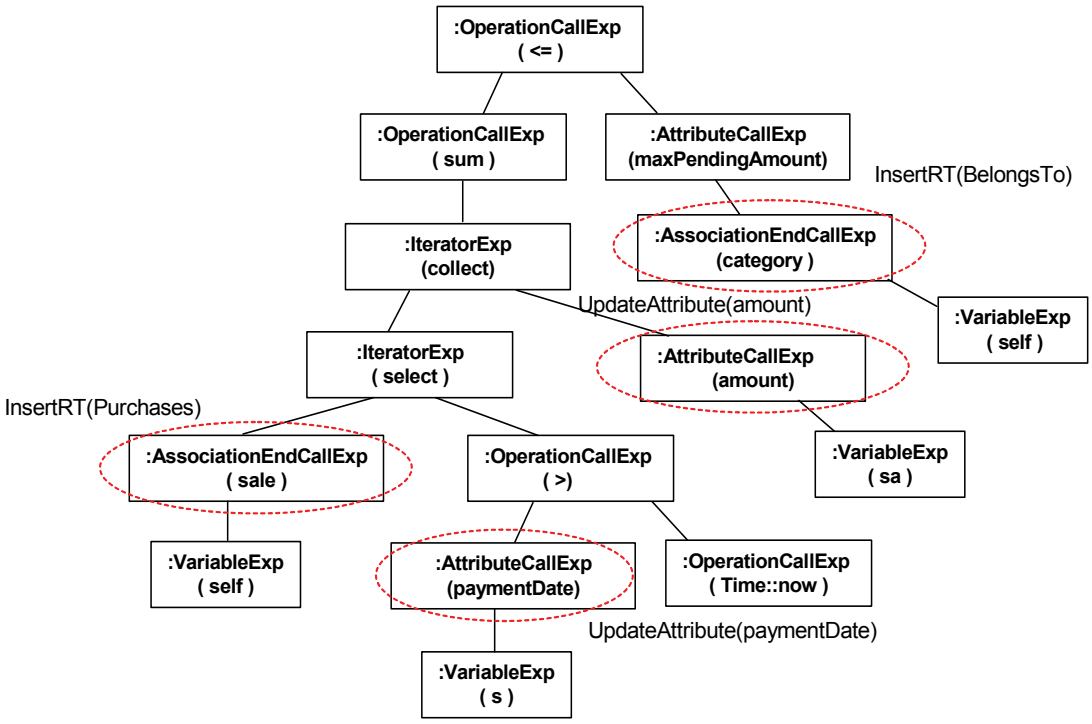


Figure 6.1. Constraint *NotTooPendingSales*$_2$

Given a conceptual schema *CS* with a set of integrity constraints, the aim of this chapter is to generate a *CS'* where the definition of all integrity constraints has been modified in order to automatically check them only in terms of the relevant instances. CS' adds to CS some auxiliary entity types necessaries to compute the relevant instances.

In particular, given a constraint *c*, we create a *structural event type* for each PSE of *c*. This type is in charge of recording the events of that type issued during the IB update. Then, the relevant instances are computed based on the population of these structural event types. Finally, our method redefines *c* so that only the relevant instances are considered during the integrity checking of *c*. The way we compute the relevant instances ensures that a constraint will not be verified if no PSE for the constraint has been issued during the IB update. Note that, when applying this part of the process over the constraints obtained at the end of the previous step (see Chapter 5), their PSEs may be a subset of the whole set of PSEs determined in Chapter 4 (for instance, the *UpdateAttribute(maxPendingAmount)* does not appear as a PSE of $NotTooPendingSales_2$ since this is not an appropriate alternative representation to check the IB after events of that type).

The rest of the chapter is structured as follows. Next section describes the number and structure of the *structural event types*. Then, Section 6.2 presents the modifications required over the CS to compute the relevant instances and to get an incremental evaluation of the constraints. These modifications depend on the structure of the constraint tree and on the placement of the PSEs inside the tree. Finally, section 6.3 applies the whole process over our running example.

6.1 Definition of structural event types

Structural event types are a specific set of entity types required to explicitly record the structural events issued during the update of the IB. More concretely, these types are devoted to record the information about the entities and relationships modified during the application of those events.

We must define a *structural event type* for each different structural event included in the set of PSEs for some constraint of the CS. Therefore, we create an *iET* structural event type for each *InsertET(ET)* event type, a *dET* type for each *Delete(ET)* event type, a *gET* type for each *GeneralizetET(ET)* event type, an *sET* type for each *SpecializeET(ET)* event type, an *uETAttribute* for each *UpdateAttribute(Attr,ET)* event type, an *iRT* type for each *InsertRT(RT)* event type and a *dRT* type for each *DeleteRT(RT)* events type. Note that we create at most one structural event type for each possible event type regardless the number of constraints having that event type as a PSE.

As an example, the list of structural event types we will define for the constraint $NotTooPendingSales_2$, according to its set of PSEs (see Figure 6.1), is the following: *uSalePaymentDate* (update of the payment date of a sale), *uSaleAmount* (update of the

amount of a sale), *iBelongsTo* (insertion in the relationship type *BelongsTo*) and *iPurchases* (insertion in the relationship type *Purchases*).

6.1.1 Structure of structural event types

In the definition of these types we assume that the IB corresponding to the CS is updated at run-time with the modifications produced by the structural events. Thus, we can avoid redundancies by not including in the structural event type the information about the changes produced by the event over the modified entity. We just need to know the modified entity. Moreover, all structural event types are stereotyped with the stereotype <<structural event>> to differentiate them from the other entity types of the CS.

Therefore, the structural event types recording insert (*InsertET*) or update (*UpdateAttribute*) events are defined as types without attributes and with just one relationship type relating the structural event type with the corresponding entity type. Through this reference we can access the entity modified by the structural event.

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

Figure 6.2 shows, as an example, the structural event type *uSalePaymentDate* corresponding to the *UpdateAttribute(paymentDate,Sale)* event type. Note that the only information recorded for each instance of *uSalePaymentDate* is a reference to the corresponding modified instance in the *Sale* type to access its information when required. Assuming that the population of the *Sale* entity type is the set of sales $\{s_1,s_2,\ldots,s_n\}$ and that the modification of the IB consist of two modification events over the *paymentDate* of sales s_1 and s_5, the population of the type *uSalePaymentDate* would be: $<u_1,s_1>$, $<u_2,s_5>$ where u_i represents the object identifier of the *uSalePaymentDate* instances and s_1 and s_5 the references to the updated sales. Note that the type *uSalePaymentDate* does not include the information about the new value of the updated sale; we may use the reference towards the *Sale* type to obtain this information.

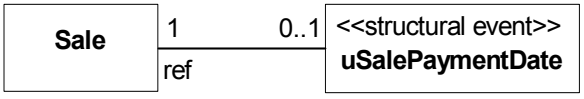


Figure 6.2 Structural event type for the event *uSalePaymentDate* over *Sale*

For structural event types in charge of recording deletion events we cannot follow the previous structure since we cannot relate the instance of the structural event type with the corresponding deleted entity of the entity type (since it does not exist). However, this does

not suppose a problem because to handle constraints including this kind of events as PSE (see section 6.2.4) it is enough with knowing whether such an event of that type has been issued or not. Therefore, to record *DeleteET* events we just create a new entity type with no attributes nor relationship types. Every time a deletion event is issued we create an empty instance in the appropriate *dET* type.

Structural event types corresponding to *InsertRT(RT)* or *DeleteRT(RT)* events do not contain attributes either. They contain as many relationship types as the number of participants in *RT*. Each one of these relationship types relates the structural event type with one of the participants of *RT*. 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 participant). Note that the types *dRT* (recording deletion events over *RT*) are perfectly possible since their instances do not point to the deleted relationship (which no longer exists in the IB) but to their participants.

As an example, Figure 6.3 shows the structural event type for the event *InsertRT(Purchases)*. The type *iPurchases* presents two relationship types, with *Customer* and *Sale*, since these entity types are the participants of *Purchases*.

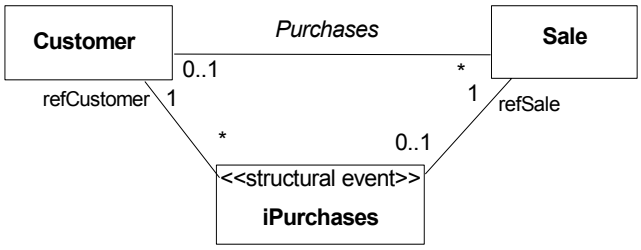


Figure 6.3 Structural event type for the event *insertRT* over *Purchases*

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

For *iRT* types, the multiplicity 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 '0..*' since, in general, an entity of a participant entity type can participate in many relationships of the relationship type (for instance, if we assign a set of sales to the same customer, several instances of *iPurchases* will refer to the same customer entity). We restrict this multiplicity to '0..x' when the instances of the participant entity type cannot participate in more than *x* relationships (as sale instances, which are related to at most a single customer).

For *dRT* types, the multiplicity on the participant type role becomes '0..1', because, after deleting the relationship, and thus, creating a new instance in the *dRT* type, it may happen

that later events delete also some of the participants of the relationship. This is not possible for *iRT* types since we cannot delete the participant without deleting before the relationship itself.

Note that we cannot remove the instance in *dRT* when deleting one of the relationship participants since we may still need the information about the deleted relationship to compute the relevant instances for constraints including this deletion event as PSE. We can only delete it when all participant entities are deleted. Constraints including as a PSE the event type *DeleteRT(RT)*, either navigate *RT* from E_1 to E_2 or from E_2 to E_1, where E_1 and E_2 are the participant entity types of *RT*. When, after deleting a relationship from *RT*, the participant E_1 is also deleted, the information about the deleted link is irrelevant for constraints that navigate *RT* from E_1 to E_2. In such a case, it is the deletion of E_1 what must be taken into account. However, for constraints navigating *RT* from E_2 to E_1, the information about the deleted relationship is required when computing the affected E_2 entities as part of the process of determining the relevant instances for the integrity checking process.

For instance, consider the constraint *AtLeastThreeCustomers*. The constraint can be violated by a deletion over *BelongsTo*. If we delete the relationship between a category *cat* and a customer *cus*, a new instance of *dBelongsTo* (Figure 6.4) is created. Even if, afterwards, we also delete the customer *cus*, the instance in *dBelongsTo* allows us to know that the category *cat* needs to be considered when checking the constraint.

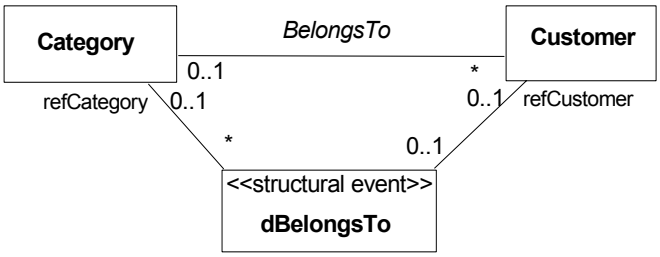


Figure 6.4 Structural event type for the event type *DeleteRT(BelongsTo)*

When an *InsertRT(RT)* or *DeleteRT(RT)* event type appears in a node having as a child node the *self* variable in the OCL tree for a constraint defined using *RT* as a context type, these events must also be treated as *InsertET* and *DeleteET* event types (and the corresponding structural event type must be generated) since *RT* is not only a relationship type but also a reified type, and thus, it has also an entity type facet. In fact, if that it is the only place where they appear, they do not need to be handled as *InsertRT* or *DeleteRT* events. When both structural event types must be created (i.e. the types for the *InsertRT* and the *InsertET* events or the types for the *DeleteRT* and the *DeleteET* events) we change the name of the structural event type for the *InsertET (DeleteET)* event into *iET'* (*dET'*) to avoid name conflicts with the structural event type for the *InsertRT (DeleteRT)* event (since ET is a reified type, *ET=RT*).

6.1.2 Instantiating the structural event types

In general, each structural event type will contain as many instances as events of that type have been executed over the corresponding entity or relationship type. For instance, the structural event type *uSalePaymentDate* will contain an instance for each sale that has changed its payment date during the update of the IB, *iPurchases* an instance for each new relationship between a customer and a sale, *dBelongsTo* an instance for each deleted relationship, etc.

To improve the efficiency of these types (i.e. to minimize their population, which results in fewer entities to consider during the integrity checking) we adapt the concept of *net effect* [21] and define two additional rules for insertions and deletions over structural event types:

- Before inserting an instance in an *uETAttribute* type we must check that the same instance does not appear previously in the types *iET* or *uETAttribute*, as well. For instance, if we update three times the payment date of the same sale during a single IB update, we only need to record this fact once.

- When deleting an entity or a relation, the corresponding instance is also deleted from the types *iET* (*iRT*), *gET*, *sET* and *uETAttribute* if existing. In addition, if the entity (relation) appears in *iET* (*iRT*) we do not need to record that it has been deleted. For instance, if we update the payment date of a sale *s* and later on, during the same modification of the IB, we delete *s* we do not need to worry about its payment date update. Moreover, if *s* was inserted during the same IB update we neither record its deletion.

6.2 Schema modification

After the application of a set of events over the IB, we must verify all constraints having as PSEs some of the issued events. The aim of this section is to modify the initial CS in order to obtain a CS' where the constraints are automatically verified only over the relevant instances (thanks to the event information recorded in the previous structural event types). This constraint redefinition process depends on the structure of the tree representing the constraint body and on the placement of its PSEs in that tree. In the following we classify the different types of constraints and explain the schema modifications for each type.

6.2.1 Constraint classification

We may distinguish three different types of integrity constraints: *instance*, *partial instance* and *class* constraints. Roughly, we classify a constraint as instance if we can always compute the exact subset of the population of its context type we need to take into account when checking it. A constraint is a class constraint if we have to consider the whole population of the context type to check the constraint. Finally, in some cases we may need to consider the whole context type population or just a subset of it depending on the

particular structural events issued during the IB update. In this case we say the constraint is a partial instance constraint.

NotTooPendingSales$_2$ is an example of instance constraint. On the contrary, *NumberOfRestrictedProducts* is a class constraint. The reason is that after inserting a new restricted product we need to count all instances of the entity type *RestrictedProduct* to verify the number of restricted products is still less than 20; it is not enough to consider only the new product to verify the constraint. As an example of a partial instance constraint consider the following constraint *MaximumCustomers*, stating that no category may hold more than half the total amount of customers: *context Category inv: self.customer->size()<= Customer.allInstances()->size()/2*. Note that if we assign a new customer to a category, we only need to check the constraint over that particular category. However, if we remove a customer from the IB, we need to verify all categories, including those where the removed customer did not work since now the total number of customers has decreased.

A constraint can be classified into exactly one of those types by examining the structure of the tree representing the constraint body and the placement of its PSEs in it. Intuitively, a constraint will be classified as *instance* if all its PSEs are included in a subtree depending on a contextual instance (i.e. a *self* variable). A constraint will be a *class constraint* when all subtrees are defined using the *allInstances* operation. A *partial instance constraint* is a constraint where some PSEs are included in subtrees related to a *self* variable and some in subtrees started by an *allInstances* operation.

More formally, given a constraint *c* with a set of PSEs set_{PSE} we define that *c* is an *instance constraint* when for each event type $ev \in set_{PSE}$, the last node *n* of $PathVar(node_{ev})$ is a node of type *VariableExp* having as a referred variable the *self* variable. As usual, $node_{ev}$ refers to the node where *ev* is attached.

On the other hand, we define that *c* is a *class constraint* when for each event type $ev \in set_{PSE}$, the last node *n* of $PathVar(node_{ev})$ is a node of type *OperationCallExp* having as a referred operation the *allInstances* predefined operation. The constraint is still a class constraint when it contains events included in a subtree starting with the *self* variable as long as the same events also appear in a node included in a subtree starting with the *allInstances* operation (if after the event we need to verify all instances of the context type, it is irrelevant to additionally check particular instances of the type as well).

A constraint is a partial instance constraint when it is neither an instance nor a class constraint (i.e. when some of its PSEs satisfy the first condition while others satisfy the second one).

Applying the previous definitions over the example integrity constraints generated at the end of the previous chapter (Table 5.2) we obtain that *ValidShipDate*, *ValidShipDate*$_2$,

ValidShipDate$_3$, *CorrectProduct*, *CorrectProduct*$_2$, *CorrectProduct*$_3$, *NotTooPendingSales*, *NotTooPendingSales*$_2$ and *AtLeastThreeCustomers* are instance constraints while *NumberOfRestrictedProducts* is a class constraint.

6.2.2 Schema modification for instance constraints

To ensure that an instance constraint is only evaluated over the relevant instances we create a new derived entity type meant to contain the exact set of instances of the context type that need to be verified.

This new entity type, called *ETConstraint* (i.e. the name of the entity type concatenated to the name of the constraint) is defined as a derived subtype of the original constraint context type. Then, we replace the original constraint with a new constraint with the same body but having as a context type the new *ETConstraint* type. This is possible because, as a subtype, *ETConstraint* contains all attributes and relationship types of its supertype. As an example, Figure 6.5 includes the redefinition of the constraint *NotTooPendingSales*$_2$ over the new *CustomerNotTooPendingSales*$_2$ entity type.

When the context type is a relationship type we are forced to reify it in order to be able to define this new subtype.

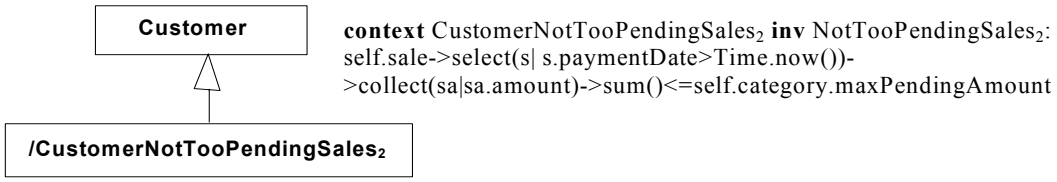


Figure 6.5. Redefinition of the *NotTooPendingSales*$_2$ constraint

Next, we need to address the computation of the population of the *ETConstraint* entity type, i.e. how to automatically define its derivation rule (section 6.2.2.2) using the set of events recorded in the structural event types. In short, the population of *ETConstraint* is the union of instances of the context type affected by each structural event appearing in the structural event types corresponding to the set of PSEs for the constraint (section 6.2.2.1).

6.2.2.1 Computing the instances of the context type affected by a structural event

Roughly, the relevant instances for a constraint *c* defined over a context type *t* after the issue of an event *ev* (where the type of *ev* is one of the PSEs for *c*) are the ones related with the instance *i* modified by *ev*.

Therefore, the basic idea is that the OCL expression required to compute such related instances will consist of the sequence of navigations *nav* required to navigate back from *i* to the instances of *t*. The application of *nav* over *i* returns the set of instances we need to verify because of *ev*.

Let *c* be a constraint defined over a context type *t* and *ev* a PSE (appearing in the $node_{ev}$) for *c*. Then, the sequence of navigations *nav* required to compute the relevant instances for an event of type *ev* are obtained with *Inverse(PathVar(*$node_{ev}$*))* where *Inverse*:

- Discards all nodes of the *PathVar* not representing navigations through relationship types (i.e. nodes not instance of the *NavigationCallExp* metaclass)
- Reverses all nodes of type *NavigationCallExp* by means of replacing the role associated to the node with the opposite role of the relationship type. If the node represents a navigation from a participant type to a reified type (i.e. a node of type *AssociationClassCallExp*) the opposite role is the one navigating back from the reified type to that participant. Reversely, the opposite role of a navigation from the reified type to one of the relationship participants is the navigation from the participant to the reified type.
- If necessary, adds the subexpression "*select(o| o.oclIsKindOf(t))*" at the end of the navigation path. When *t* belongs to a taxonomy, the previous computed navigation path may return a set of objects of type *t'*, where *t'* is a supertype of *t*. Since the context of the constraint is *t* only those objects that are instance of *t* (or instance of one of the subtypes of t) are relevant to the constraint. In such cases the *select* expression ensures that only objects of type *t* are considered.

As an example, consider the event *UpdateAttribute(paymentDate, Sale)* over the constraint $NotTooPendingSales_2$. Figure 6.6 shows the ordered sequence of nodes resulting from the application of *PathVar* over the node including the update *paymentDate* event type. The sequence contains a single node representing a navigation through an association end (the association end *sale* of the relationship type *Purchases*). Therefore, *Inverse* just returns the node *AssociationEndCallExp(customer)*, i.e. the opposite role of *sale* in *Purchases*. Then, to obtain the affected customers after a *paymentDate* update, we just navigate from the modified sale *s* to the related customer applying over *s* the navigation through the *customer* role.

When *Inverse* returns an empty sequence of navigations it means that the instance modified by the event is exactly the instance of the context type we need to take into account when verifying the constraint.

If the same PSE appears in different nodes of the tree, to compute the affected instances we repeat the process for each node and combine the result afterwards by means of the *union* operator. Similarly, when a node may have different *PathVar* expressions we combine the sets of affected instances computed for the indicated paths (see section 5.1.2).

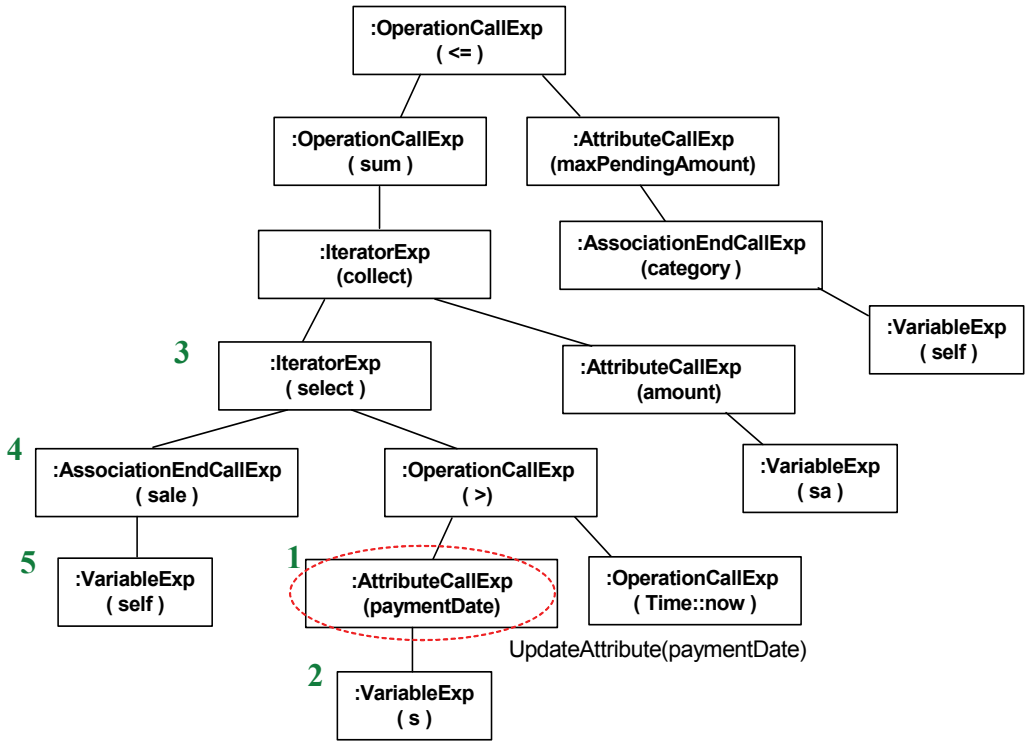


Figure 6.6. PathVar for the *UpdateAttribute(paymentDate,Sale)* event type

6.2.2.2 Derivation rule definition

The derivation rule for the *ETConstraint* entity type must ensure that the set of instances of the type are exactly the set of instances we need to check. It must include, for each instance of the structural event types corresponding to the PSEs of the constraint, the computation of the affected instances of the context type, as explained above.

Using the work of [64] we define the population of a derived entity type by means of redefining its predefined *allInstances* operation (i.e. the population of the derived type will be the set of instances returned by the *allInstances* operation).

We first obtain all instances of an structural event type *evt* (where *evt* records events included in the set of PSEs for the constraint) by means of the expression *evt.allInstances()*. Then, for each instance, we use the relationship types between the structural event types and its corresponding entity types to access the modified entities (see Section 6.1). Then, over the obtained set of modified entities we apply the sequence of navigations computed in section 6.2.2.1 to retrieve the relevant instances of the context type. We combine this set of instances with the results of repeating the process with the other structural event types corresponding to the PSEs of the constraint.

Note that when the structural event type corresponds to an *InsertRT* or a *DeleteRT* event type, we have different possibilities when accessing the modified instance since we may access any of the participants of the relationship type. The right participant to navigate to is

determined by the first navigation in the sequence of navigations required to compute the affected instances of the context type due to that event. If such navigation requires navigating to the participant *p*, from the instances of the structural event type we will access that participant.

As an example, consider the previous *NotTooPendingSales*$_2$ constraint. In this case, the derivation rule for the derived subtype *CustomerNotTooPendingSales*$_2$ must select, according to the PSEs for the constraint (see Figure 6.1), all customers that have purchased a sale (i.e. all customers participating in a new relationship of the *Purchases* relationship type, recorded in the *iPurchases* structural event type). These customers are obtained by means of the subexpression *iPurchases.allInstances().refCustomer*, where we first retrieve all new *Purchases* relationships and from them we obtain the related customers (the *Inverse(PathVar)* expression for the node including the *InsertRT(Purchases)* event type results in the *customer* role, which must be applied over the new purchases relationship). Similarly, we select the customers being assigned to a category (customers participating in a new relationship of the *BelongsTo* relationship type, recorded in the *iBelongsTo* type).

The derivation rule must also select those customers related to sales that have been modified the value of its payment date (*uSalePaymentDate type*) or *amount* (*uSaleAmount* type) attribute values. This last set of customers is obtained by applying the role *customer* over each updated sale (reached from the *uSalePaymentDate* and *uSaleAmount* types by means of the *ref* role).

Therefore, the derivation rule for *CustomerNotTooPendingSales*$_2$ is the following:

context *CustomerNotTooPendingSales*$_2$*::allInstances() : Set(Customer)*
body: *iPurchases. allInstances().refCustomer->union(*
iBelongsTo. allInstances().refCustomer->union(
uSalePaymentDate.allInstances().ref.customer->union(
uSaleAmount.allInstances().ref.customer)))->asSet()

We would like to remark that the derivation rule returns a set (and not a bag) of instances. This permits to avoid a redundant checking of the relevant instances even if they are affected by several of the events issued during the IB update (for instance, a customer that purchases a new sale and that it is related to an existing sale that has changed its *amount* attribute). Moreover, if none of the issued events is a PSE for the constraint the population of *ETConstraint* will be empty, and thus, the constraint will not be verified.

6.2.3 Schema modification for class constraints

For class constraints it is unnecessary to compute the affected instances after the issue of one of their PSEs since we always need to consider the whole population of the context type. However, it is still relevant to modify the schema in order to verify the constraint

only after modifications of the IB where at least an event *ev* (where the type of *ev* appears in the list of PSEs for the constraint) is applied over the IB.

Given a class constraint *c* defined over a context type *t*, with body *X* and with a set of PSEs set_{PSE} recorded in the structural event types $set_{SET} = \{evt_1, evt_2, ...evt_n\}$, the redefined constraint *c'* follows the pattern:

> ***context*** *t* ***inv****: if* evt_1*.allInstances()->notEmpty() or* evt_2*.allInstances()->notEmpty() or ...* evt_n*.allInstances()->notEmpty() then X endif*

Note that we do not retrieve the exact instances modified during the IB update; we just check if at least an event instance of one of the PSEs has been issued.

As an example, after processing the class constraint *NumberOfRestrictedProducts* we obtain the following result:

> ***context*** *RestrictedProduct* ***inv*** *NumberOfRestrictedProducts: if iRestrictedProduct.allInstances()->notEmpty() or sRestrictedProduct.allInstances()->notEmpty() then RestrictedProduct.allInstances()->size()<=20 endif*

6.2.4 Schema modification for partial instance constraints

A partial instance constraint *c* can be checked incrementally only when none of the PSE events applied over the IB appears related to subexpressions started by an *allInstances* operation in the constraint tree. Otherwise, we must check the constraint over all instances of the context type.

To process this kind of constraints we split their set of PSEs into two different groups: the set of *instance* PSEs and the set of *class* PSEs, depending on the kind of subexpression where they are included (i.e. depending on the type of last node of their *PathVar* expression as explained in section 6.2.1). If a PSE is included in both kinds of subexpressions is considered a class PSE.

For the set of instance PSEs we apply the same treatment explained in section 6.2.2, and thus, we create the new derived subtype *ETConstraint* and change the context of the constraint to *ETContraint*. There is only a slight difference regarding the generation of the derivation rule for the subtype, as we will explain below.

Afterwards, we create an additional derived subtype, called *ETConstraint'*, under the context type, and define also over *ETConstraint'* a copy of the original constraint. Its population will be the same population of the context type if some class PSE has been applied over the IB. Otherwise, its population will be empty. Therefore, the derivation rule for *ETConstraint'* is ($evt_1 \ldots evt_n$ represent the structural event types for the class PSEs):

context *ETConstraint'::allInstances(): Set(ET)*
body
if (evt_1.allInstances()->notEmpty() or evt_2.allInstances->notEmpty() or ...
evt_n.allInstances()->notEmpty()) then ETConstraint.allInstances() endif

where $evt_1..evt_n$ represent the structural event types corresponding to the *class* PSEs.

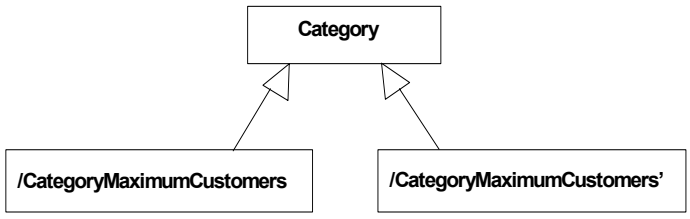
Finally, once we have created this new *ETConstraint'* subtype, we define the derivation rule for the *ETConstraint* type created for the instance PSEs. Its derivation rule will be:

if *(ETConstraint'.allInstances()->isEmpty())* ***then*** *dr* ***endif***

where *dr* is the derivation rule defined according to section 6.2.2.2 but considering only the instance PSEs.

This way we ensure that when the transaction includes a class PSE we check all instances of the original context entity type (*ETConstraint'* will contain the same instances as the context type) and avoid redundant checkings (*ETConstraint* will be empty). Otherwise, we may check the constraint incrementally (*ETConstraint* will contain the affected instances of the context type whereas *ETConstraint'* will be empty).

Figure 6.7 shows the constraint *MaximumCustomers* (*context Category inv: self.customer->size()<= Customer.allInstances()->size()/2*) after being processed as explained in this section. Notice that, when a *DeleteET(Customer)* event is issued, *dCustomer* becomes not empty. As a consequence, the population of *CategoryMaximumCustomers'* is equivalent to the population of the *category* type and we check all categories to verify that all of them satisfy the constraint. At the same time, *CategoryMaximumCustomers* becomes empty (even if some *InsertRT(BelonsTo)* event has been issued as well). On the contrary, when only events of type *InsertRT(BelonsTo)* are applied over the IB, *CategoryMaximumCustomers'* is empty and the integrity checking just considers the relevant categories included in *CategoryMaximumCustomers*.



-- The redefined constraints
context CategoryMaximumCustomers **inv** MaximumCustomers:
self.customer->size()<= Customers.allInstances()->size() / 2

context CategoryMaximumCustomers' **inv** MaximumCustomers':
self.customer->size()<= Customers.allInstances()->size() / 2

-- The derivation rules
context CategoryMaximumCustomers::allInstances():Set(Category)
body: if CategoryMaximumCustomers'.allInstances->isEmpty() then
iBelongsTo.allInstances().refCategory endif

context CategoryMaximumCustomers'::allInstances():Set(Category)
body: if dCustomer.allInstances->notEmpty() then
Category.allInstances() endif

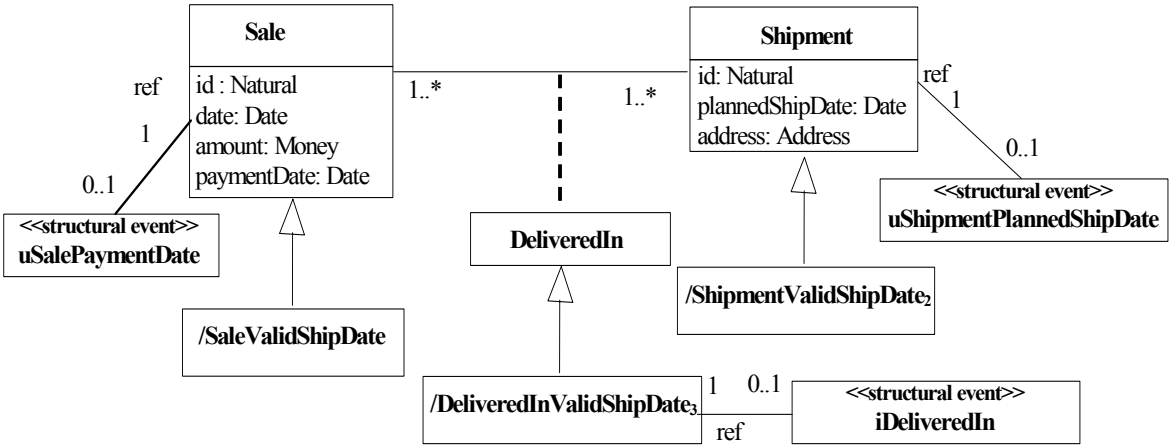
Figure 6.7. Schema modification for MaximumCustomers partial instance constraint

6.3 Application to the running example

In what follows we show the new conceptual schema generated as a result of processing all constraints obtained at the end of Chapter 5 (see Table 5.2) in order to ensure their incremental verification after all kinds of events.

To facilitate the presentation of the results we split the new schema into several subsets, one for each group of related constraints (each group refers to the different constraint alternatives produced for each one of the original constraints of the running example). The entity or relationship types not appearing in these figures remain unmodified from the original schema (Figure 1.1).

From Figure 6.8 it is worth to note the reification of the *DeliveredIn* relationship type, necessary to process the constraint *ValidShipDate*$_3$. For the same reason, the *InsertRT(DeliveredIn)* event type is treated as an *InsertET* event type when creating the corresponding structural event type *iDeliveredIn*.



-- The derivation rules

context SaleValidShipDate::allInstances() : Set(Sale) **body:** uSalePaymentDate.allInstances().ref

context ShipmentValidShipDate$_2$::allInstances() : Set(Shipment) **body:** uShipmentPlannedShipDate.allInstances().ref

context DeliveredInValidShipDate$_3$::allInstances() : Set(DeliveredIn) **body:** iDeliveredIn.allInstances().ref

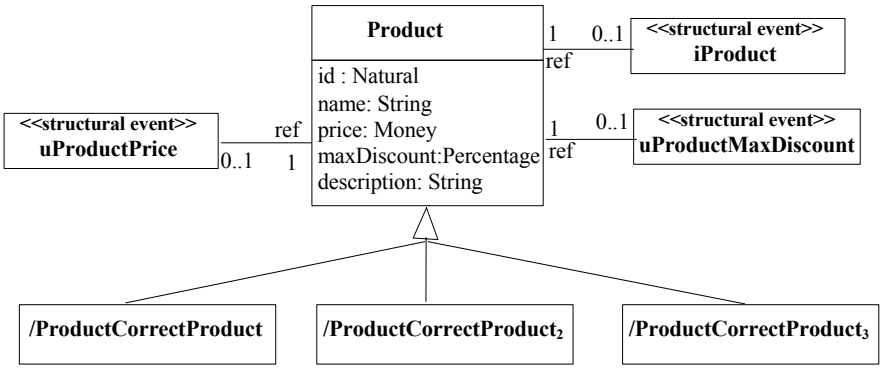
-- The redefined constraints

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

context ShipmentValidShipDate$_2$ **inv** ValidShipDate$_2$: self.sale->forAll(s| self.plannedShipDate<=s.paymentDate+30)

context DeliveredInValidShipDate$_3$ **inv** ValidShipDate$_3$: self.shipment.plannedShipDate<=self.sale.paymentDate+30

Figure 6.8. Schema modification for *ValidShipDate*, *ValidShipDate*$_2$ and *ValidShipDate*$_3$ integrity constraints



-- The derivation rules

context ProductCorrectProduct::allInstances() : Set(Product) **body:** iProduct.allInstances().ref

context ProductCorrectProduct$_2$::allInstances() : Set(Product) **body:** uProductPrice.allInstances().ref

context ProductCorrectProduct$_3$::allInstances() : Set(Product) **body:** uProductMaxDiscount.allInstances().ref

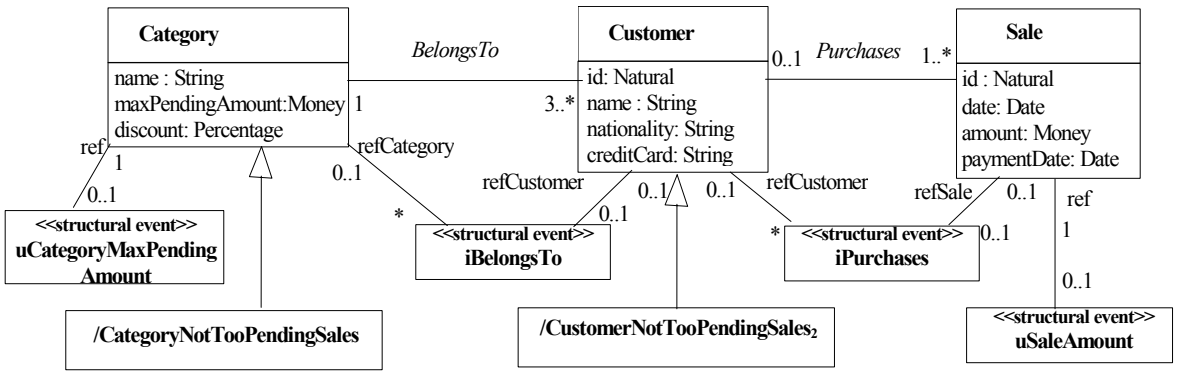
-- The redefined constraints

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

context ProductCorrectProduct$_2$ **inv** CorrectProduct$_2$: self.price>0

context ProductCorrectProduct$_3$ **inv** CorrectProduct$_3$: self.maxDiscount<=60

Figure 6.9. Schema modification for *CorrectProduct*, *CorrectProduct$_2$* and *CorrectProduct$_3$* integrity constraints



-- The derivation rules

context CategoryNotTooPendingSales::allInstances() : Set(Category)

body: uCategoryMaxPendingAmount.allInstances().ref

context CustomerNotTooPendingSales$_2$::allInstances() : Set(Customer)

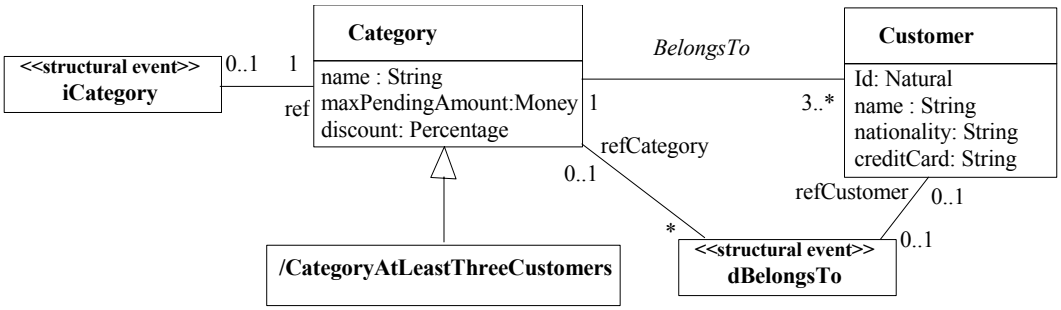
body: iBelongsTo.allInstances().refCustomer->union(iPurchases. allInstances().refCustomer->union(
uSalePaymentDate.allInstances().ref.customer->union(uSaleAmount.allInstances().ref.customer)))->asSet()

-- The redefined constraints

context CategoryNotTooPendingSales **inv** NotTooPendingSales: self.customer->forAll(c| c.sale->select(s|paymentDate>Time.now())->collect(sa|sa.amount)->sum()<=self.maxPendingAmount)

context CustomerNotTooPendingSales$_2$ **inv** NotTooPendingSales$_2$: self.sale->select(s| s.paymentDate>Time.now())->collect(sa|sa.amount)->sum()<=self.category.maxPendingAmount

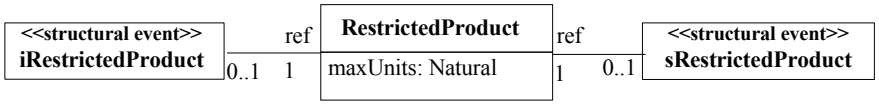
Figure 6.10. Schema modification for *NotTooPendingSales* and *NotTooPendingSales$_2$* integrity constraints. The type *uSalePaymentDate* appears in Figure 6.8 and it is not repeated here



-- The derivation rule
context CategoryAtLeastThreeCustomers::allInstances() : Set(Category)
body: iCategory.allInstances().ref->union(dBelongsTo.allInstances().refCategory)->asSet()

-- The redefined constraint
context CategoryAtLeastThreeCustomers **inv** AtLeastThreeCustomers: self.customer->size()>=3

Figure 6.11. Schema modification for *AtLeastThreeCustomers* integrity constraint



context RestrictedProduct **inv** NumberOfRestrictedProducts:
if iRestrictedProduct.allInstances()->notEmpty() or sRestrictedProduct.allInstances()->notEmpty() then
RestrictedProduct.allInstances()->size()<=20 endif

Figure 6.12. Schema modification for *NumberOfRestrictedProducts* integrity constraint

6.4 Summary and discussion of the results

With this step we complete the processing of the original constraints. As a result, our method has returned a new conceptual schema where all constraints have been redefined in order to get their incremental evaluation after arbitrary modifications of the IB. The cost of verifying the new constraints is much lower than the cost of verifying the original ones. As a trade-off, the size of the CS has been increased with the addition of new types, constraints and derivation rules. In the following we comment both aspects of the processed schema.

Defining the cost of checking a constraint as the number of entities that must be taken into account during its evaluation, Tables 6.1-6.5 illustrate, for each one of the original constraints, the differences between a direct checking of the constraint and the cost of checking the new version. Obviously, at design time we cannot determine the exact complexity of the constraints since the cost depends on the exact population of the entity and relationship types. We must represent these values by means of abstract variables. However, the abstract formulas we use are rich enough to stand out the cost differences. Although not explicited in the tables, an additional efficiency gain of the processed schema is that when the modification of the IB does not include any of the PSEs for a constraint *c*, *c* is not verified (i.e. the cost is zero). This is not restricted in the original schema.

In all tables, the column *PSE* shows the PSEs of the original constraint. Column *Incr Constraint* refers to the name of the specialized constraint generated for that PSE in the processed CS (note that even if the name of the new constraint coincides with that of the original one, in the processed schema the new constraint has been redefined to be evaluated over the relevant instances, and thus, their cost may be different). Column cost *old* and cost *new* refer to the cost of evaluating the original and the processed constraints after the issue of an event of the event type appearing in the first column.

In Table 6.1 (cost comparison for *ValidShipDate* constraint), P_s stands for the number of instances of *Sale*, N_{sh} for the average number of shipments per sale and N_s for the average number of sales per shipment. A direct verification of the original *ValidShipDate* constraint always involves considering all sales and for each sale all the related shipments. Therefore, the number of instances accessed during its evaluation is P_s plus P_s multiplied by the average number of shipments for each sale (N_{sh}). Instead, after a sale update, in the new schema we just access to that sale and its related shipments (because of the redefinition of *ValidShipDate* over the relevant instances). After a shipment update, we simply retrieve the updated shipment and then compare it with the assigned sales (thanks to the use of the new specialized constraint $ValidShipDate_2$). Finally, after the insertion of a new relationship between a sale *s* and a shipment *sh*, comparing *s* and *sh* suffices to verify $ValidShipDate_3$.

Table 6.1. Cost comparison for *ValidShipDate*

PSE	Incr Constraint	Cost old	Cost new
UpdateAttr(paymentDate, Sale)	ValidShipDate	$P_s + P_s \times N_{sh}$	$1+1xN_{sh}$
UpdateAttr(plannedShipDate, Shipment)	$ValidShipDate_2$	"	$1+1xN_s$
InsertRT(DeliveredIn)	$ValidShipDate_3$	"	2

Table 6.2 shows the costs for *CorrectProduct*. P_p stands for the number of instances of *Product*. The difference is that in the processed constraint we restrict the verification process to the inserted/updated product.

Table 6.2. Cost comparison for *CorrectProduct*

PSE	Incr Constraint	Cost old	Cost new
InsertET(Product)	CorrectProduct	P_p	1
UpdateAttribute(price,Product)	$CorrectProduct_2$	"	1
UpdateAttribute(maxDiscount,Product)	$CorrectProduct_3$	"	1

Table 6.3 shows the cost comparison for *NotTooPendingSales* constraint. In the table, P_{ca} stands for the number of instances of *Category*, N_{cu} for the average number of customers per category and N_{sa} for the average number of sales per customer. The cost of evaluating the original constraint always implies accessing all categories (P_{ca}), for each category all

its customers (resulting in a cost of P_{ca} x N_{cu}) and for each customer all its sales (this adds to the previous cost the number of accessed sales, determined by the expression P_{ca} x N_{cu} x N_{sa}). On the contrary, with the processed constraints the verification of the IB after a category update involves just that category instead of the whole population of the *Category* type. After the other events the cost is even lower since we merely access the affected customer and his/her sales. For instance, after the update of sale *s,* the number of involved instances is *s*, the customer that has purchased *s*, all sales of that customer and the category where the customer belongs to.

Table 6.3. Cost comparison for *NotTooPendingSales*

PSE	Incr Constraint	Cost old	Cost new
UpdateAttr(maxPendingAmount, Category)	NotTooPendingSales	$P_{ca} + P_{ca}$ x N_{cu} + P_{ca}x N_{cu}xN_{sa}	1+1xN_{cu} + 1xN_{cu}xN_{sa}
InsertRT(BelongsTo)	NotTooPendingSales$_2$	"	1+1xN_{sa}+1
InsertRT(Purchases)	NotTooPendingSales$_2$	"	1+1xN_{sa}+1
UpdateAttribute(paymentDate,Sale)	NotTooPendingSales$_2$	"	1+1+1xN_{sa}+1
UpdateAttribute(amount, Sale)	NotTooPendingSales$_2$	"	1+1+1xN_{sa}+1

In Table 6.4 we provide the cost comparison for *AtLeastThreeCustomers*. P_{ca} stands for the number of instances of *Category* and N_{cu} for the average number of customers per category. The cost of evaluating the original constraint always implies accessing all categories (P_{ca}) and for each category all its customers (resulting in a cost of P_{ca} x N_{ca}). After the redefinition, the verification involves just computing the number of customers of the modified category.

Table 6.4. Cost comparison for *AtLeastThreeCustomers*

PSE	Incr Constraint	Cost old	Cost new
InsertET(Category)	AtLeastThreeCustomers	P_{ca} +P_{ca} x N_{cu}	1+1xN_{cu}
DeleteRT(BelongsTo)	AtLeastThreeCustomers	"	1+1xN_{cu}

Finally, Table 6.5 shows the costs for *RestrictedProduct*. P_{rp} stand for the number of instances of restricted product. Since this is a class constraint, after the PSEs of the constraint, the cost of evaluating the processed constraint coincides with the cost of evaluating the original constraint. The difference is that in the processed schema, the constraint is only evaluated after IB updates containing at least a PSE for the constraint while in the original one this was not controlled.

Table 6.5. Cost comparison for *RestrictedProduct*

PSE	Incr Constraint	Cost old	Cost new
InsertET(RestrictedProduct)	RestrictedProduct	P_{rp}	P_{rp}
SpecializeET(RestrictedProduct)	RestrictedProduct	"	"

In order to get these efficiency improvements, we incur in additional costs with respect to the size of the CS and with respect to the small overhead required, at execution time, to record the issued events in the structural event types during the modification of the IB.

The size of the CS increases because of the addition of the structural event types, the derived subtypes (for *instance* and *partial instance* constraints), their derivation rules and the generated alternative constraint representations. However, the designer does not need to be aware of this additional complexity since the processed CS will be automatically processed by code-generation tools. Moreover, the number of new model elements in the CS is linear with respect to the number of integrity constraints in the CS. The number of structural event types depends on the number of different PSEs of the constraints (if two constraints share the same PSE, only a structural event type is created). The number of derived subtypes is equal to the number of instance constraints plus twice the number of partial instance constraints. The number of alternative constraints generated depends on the number of PSEs (counting as different PSEs those appearing more than once in the constraint body) for each constraint. At most, a different constraint for each PSE will be generated.

Nevertheless, we believe the efficiency gain we get with our method sufficiently justifies such additional complexity in a vast majority of situations. The exception would be those types with a low population expected at run-time. For them, the difference between a direct verification and an incremental one after some events may be small, and thus, the additional complexity required to do an incremental verification could not be justified.

7. Tool implementation

The method presented in this thesis has been implemented in a prototype tool. The tool can be downloaded from [16]. Some of the tool features have been developed by Carol Cervelló [25] and Raúl Solana [81].

Figure 7.1 shows the general picture of the tool architecture. Given an XMI file [70] representing the initial conceptual schema *CS*, and a set of constraints expressed in their concrete (textual) syntax, the tool loads the schema and the constraints information into main memory (with the help of the *MDR* tool [60] for the import/export of the XMI file and the *Dresden OCL* tool [31] for parsing and loading the constraints), process them and returns the XMI file corresponding to the processed schema *CS'* and a textual file with the set of redefined constraints and with the derivation rules for the new derived subtypes.

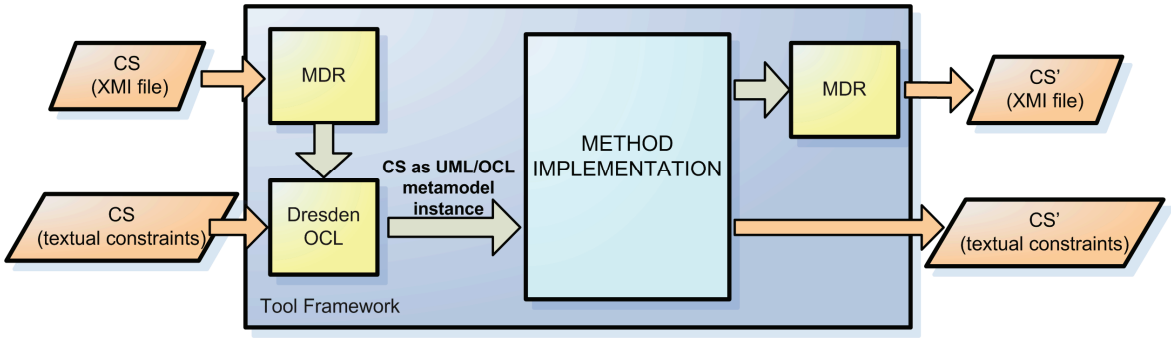


Figure 7.1. Tool architecture

In its implementation, our tool relies on different existing technologies and standards. In the following we briefly introduce each of them (Section 7.1) and explain how they fit in our overall architecture (Section 7.2). Note that, due to lack of support for the latest versions of the UML and OCL standards, our tool works with version [71] of the UML metamodel and version [67] of the OCL metamodel.

7.1 Underlying technologies

7.1.1. XMI

XMI (XML Metadata Interchange, [70]) is an OMG standard for sharing objects instances of a MOF-compliant metamodel (Meta-Object Facility, [72]) using XML documents. In our case, these objects are instances of the UML Metamodel. Therefore, through XMI we can interchange UML models by means of XML documents.

In particular, XMI defines which XML tags are used to represent serialized models in XML. Each MOF-compliant metamodel *met* is translated into a XML Schema or a DTD (XML Document Type Definitions). Then, models instance of *met* are translated into XML documents that are consistent with their corresponding DTD or XML Schema.

As an example, Figure 7.2 shows an excerpt of the XMI representation for our e-commerce schema used as a running example. The tags *<UML:Package>*, *<UML:Class>*, *<UML:Attribute>*, etc, are the ones defined by XMI to store UML models. The elements inside the tags represent the information about the actual conceptual schema. In this example, we may see that the schema contains a class called *Category* (*<UML:Class>* tag) member of the *Company* package (*<UML:Package>* tag) and with an attribute *name* (*<UML:Attribute>* tag) with multiplicity 1 (*<UML:MultiplicityRange>* tag).

```
<UML:Package xmi.id = '.:0000000000000888' name = 'company' visibility = 'public'
      isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract = 'false'>
      <UML:Namespace.ownedElement>
       <UML:Class xmi.id = '.:0000000000000834' name = 'Category' visibility = 'public'
        isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract = 'false'
        isActive = 'false'>
        <UML:Classifier.feature>
         <UML:Attribute xmi.id = '.:000000000000082D' name = 'name' visibility = 'private'
          isSpecification = 'false' ownerScope = 'instance' changeability = 'changeable'>
          <UML:StructuralFeature.multiplicity>
           <UML:Multiplicity xmi.id = '.:000000000000082C'>
            <UML:Multiplicity.range>
             <UML:MultiplicityRange xmi.id = '.:000000000000082B' lower = '1' upper = '1'/>
            </UML:Multiplicity.range>
           </UML:Multiplicity>
          </UML:StructuralFeature.multiplicity>
          <UML:StructuralFeature.type>
           <UML:Class xmi.idref = '.:0000000000000823'/>
          </UML:StructuralFeature.type>
         </UML:Attribute>
```

Figure 7.2. Partial XMI representation for the e-commerce example

7.1.1. JMI

The JMI (Java Metadata Interface, [46]) specification enables the implementation of a dynamic, platform-independent infrastructure to manage the creation, storage, access, discovery, and exchange of metadata based on a MOF-compliant metamodel, again, the UML and OCL metamodels in our case.

In particular, JMI defines the standard Java interfaces for all elements of the UML metamodel; an interface for each metaclass defined in the UML metamodel. For non-abstract metaclasses it specifies an additional interface to be used as a factory for creating

new instances of that metaclass. Through these interfaces, an application can discover, access and manipulate UML models. As an example, Figure 7.3 shows the interface corresponding to the *Generalization* metaclass, with the appropriate methods to get/set the parent type, the discriminator and the subtype.

JMI also provides metamodel and metadata interchange via XMI. This implies that JMI permits to import a UML model from an XMI file and export it back to its XMI representation after its modification by means of the methods defined in the set of Java interfaces defined by JMI.

```
public interface Generalization extends jmi.uml15.core.Relationship {
    public java.lang.String getDiscriminator();
    public void setDiscriminator(java.lang.String newValue);
    public tudresden.ocl20.core.jmi.uml15.core.GeneralizableElement getChild();
    public void setChild(tudresden.ocl20.core.jmi.uml15.core.GeneralizableElement newValue);
    public tudresden.ocl20.core.jmi.uml15.core.GeneralizableElement getParent();
    public void setParent(tudresden.ocl20.core.jmi.uml15.core.GeneralizableElement newValue);
    public tudresden.ocl20.core.jmi.uml15.core.Classifier getPowertype();
    public void setPowertype(tudresden.ocl20.core.jmi.uml15.core.Classifier newValue);
}
```

Figure 7.3. JMI interface for the *Generalization* metaclass

7.1.2. MDR

MDR (Metadata Repository, [60]) is an extended implementation of JMI. Since JMI is just a specification, it cannot be used on its own. Instead, we must rely on a JMI-compliant tool. The main function of this kind of tools (being MDR the most relevant representative) is to provide a class implementation for all interfaces specified by JMI.

MDR also offers an API with an extended functionality to query, create and manage UML models and its elements. For instance, with the API offered by MDR the user could create a completely new model from scratch.

Moreover, to implement the required XMI import/export functionality (as defined by JMI), MDR defines an internal repository (in main memory) where it stores all the information about the models and model elements loaded by the user.

7.1.4. Dresden OCL Toolkit

Dresden OCL Toolkit [31] complements MDR with appropriate support for OCL expressions. It uses JMI and MDR to specify and implement all the Java classes corresponding to the OCL metamodel. Then, it offers the possibility of transforming OCL expressions expressed in a concrete (textual) syntax form into their abstract form (i.e. as instances of the OCL metamodel).

The transformation process is split up in two different steps:

- Parsing: Creation of a concrete syntax tree from the textual constraint. The parser is based on a tailored, hand-optimized L-attributed grammar of OCL2.0. It enhances the popular LALR(1) parser generator *SableCC* to create a lexer, and a syntax analyzer. The parsing step permits to detect ill-formed OCL expressions (i.e. expressions inconsistent with the OCL grammar)

- Analysis: the attribute evaluator module performs the transformation from the concrete syntax tree to the abstract syntax tree (i.e. the representation of the constraints as an instance of the OCL metamodel). During this step the references to UML model elements appearing in the constraint are linked to appropriate objects created during the previous loading of the UML model from the XMI file. For instance, for nodes of type *AssociationEndCallExp* (Figure 4.2) the association *referredAssociationEnd* is linked to the corresponding *AssociationEnd* object. The toolkit is able to detect semantic errors in this phase. For instance, expressions referencing model elements not existing in the conceptual schema or with an incorrect type or an incorrect multiplicity (as, for instance, association ends with a maximum multiplicity > 1 that are treated in the OCL expression as if they returned a single object).

The toolkit handles all kind of OCL expressions, including integrity constraints, derived elements and pre/postconditions of operations. In principle, it supports the complete OCL syntax although the current version of the tool still presents some bugs that limit a little bit the expressions that are recognized as valid OCL expression for the tool. For an updated list of these limitations refer to the bug list in the project web page.

7.2 Tool architecture

Our tool is implemented as a set of Java classes extended with the libraries of the Dresden OCL toolkit (for the parsing and loading of OCL constraints) and MDR (for the import/export of UML models from XMI files).

As a first step, the tool permits to import both the UML model and the OCL constraints from an XMI file and a textual file, respectively (Figure 7.4). Then, the user can choose to generate the processed schema. The generation process starts by loading the UML/OCL model into main memory (with the previous auxiliary tools). The tool internally stores the CS and the constraints as instances of the UML and OCL metamodels. Then, it applies the steps described in chapters 3-6 over this metamodel representation.

Finally, it exports the generated schema to an XMI representation by means of executing the methods provided by the MDR library. Figure 7.5 shows part of the schema of Figure 1.1 once processed to get an incremental checking of the *ValidShipDate* constraint. *ArgoUML* [2] is used to display the exported XMI file. Note that *ArgoUML* does not graphically show multiplicities of associations ends when the multiplicity value is exactly

one. Moreover, due to limitations of the MDR implementation of the UML metamodel we cannot mark an entity type as derived; only derived attributes and relationship types are allowed. Therefore, we stereotype all created derived subtypes with the «derived» stereotype.

To save back the modified constraints and the derivation rules to a textual representation we have developed our own transformation algorithm since this functionality was not yet provided by the Dresden OCL tool (in fact, this part of our tool is now offered as part of the latest version of the Dresden OCL toolkit).

Apart from this basic functionality (generating the processed schema to get an incremental integrity checking of all constraints), the tool also provides some auxiliary operations that are useful for potential users willing to partially apply our method. For instance, the user may be interested in just knowing which event types are the PSEs for a single constraint (Figure 7.6) or in knowing the list of constraints that can be violated by certain type of event (Figure 7.7).

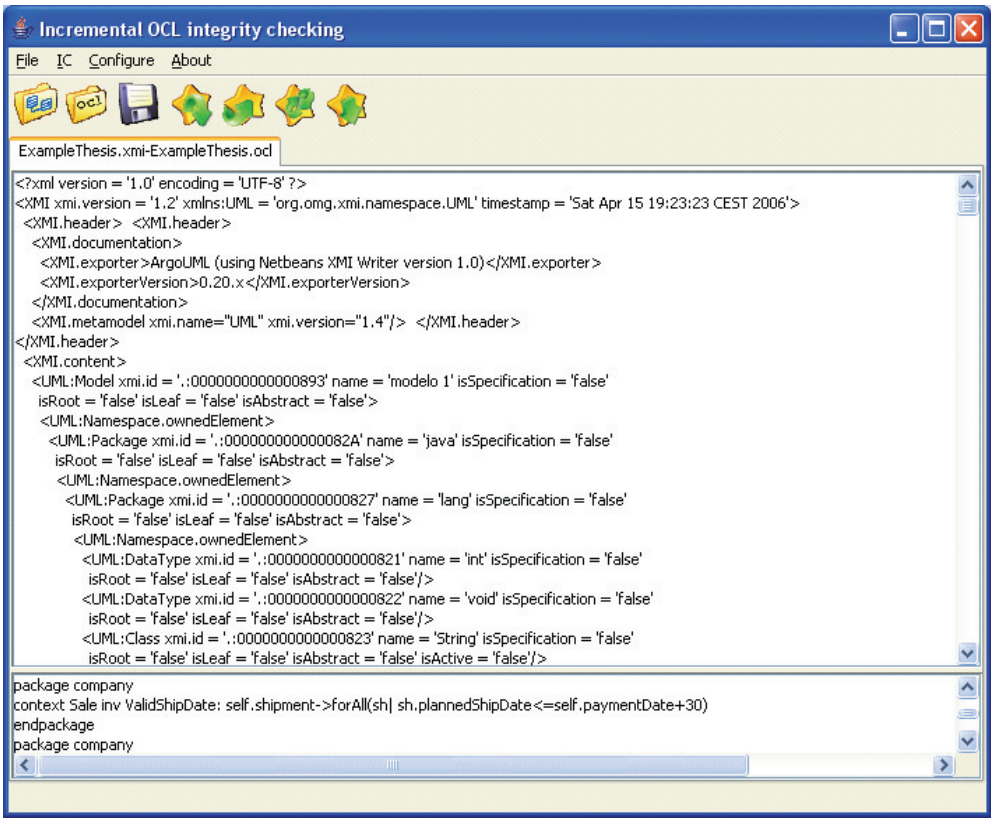


Figure 7.4. Main form of our tool

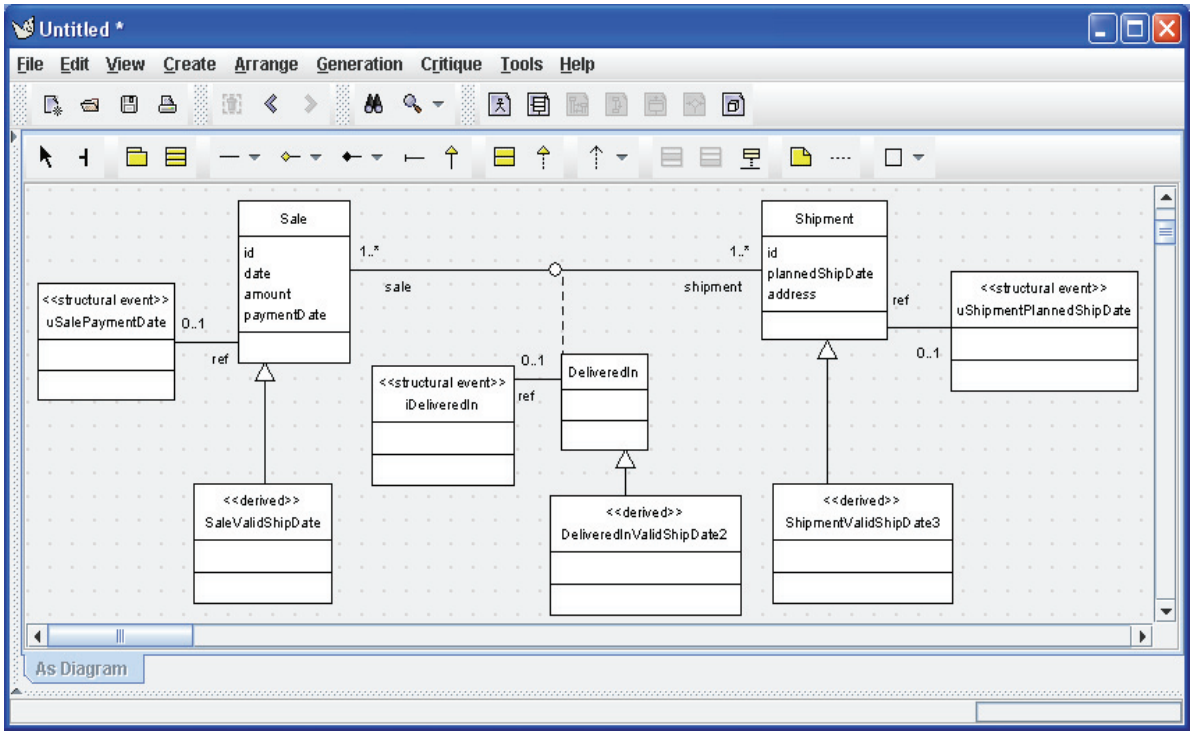


Figure 7.5. Processed schema

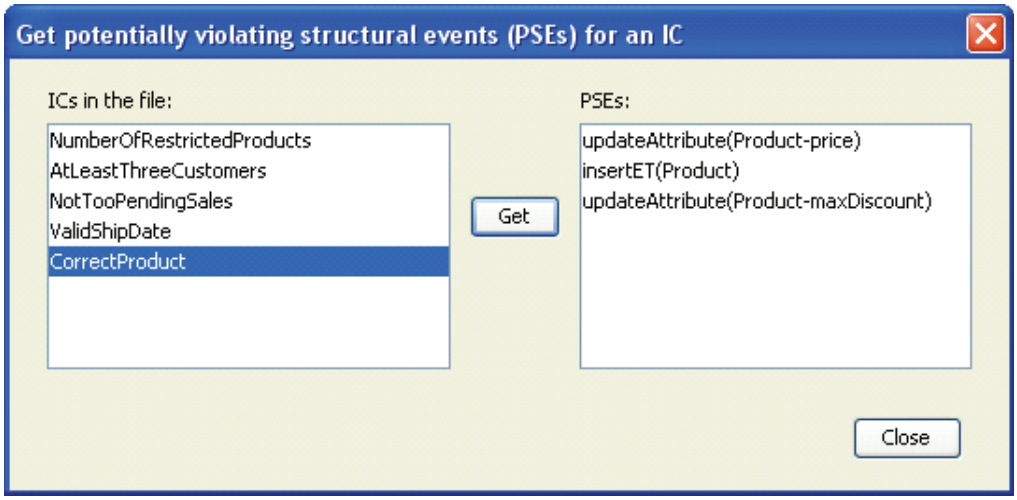


Figure 7.6. Retrieving information about the PSEs for *CorrectProduct*

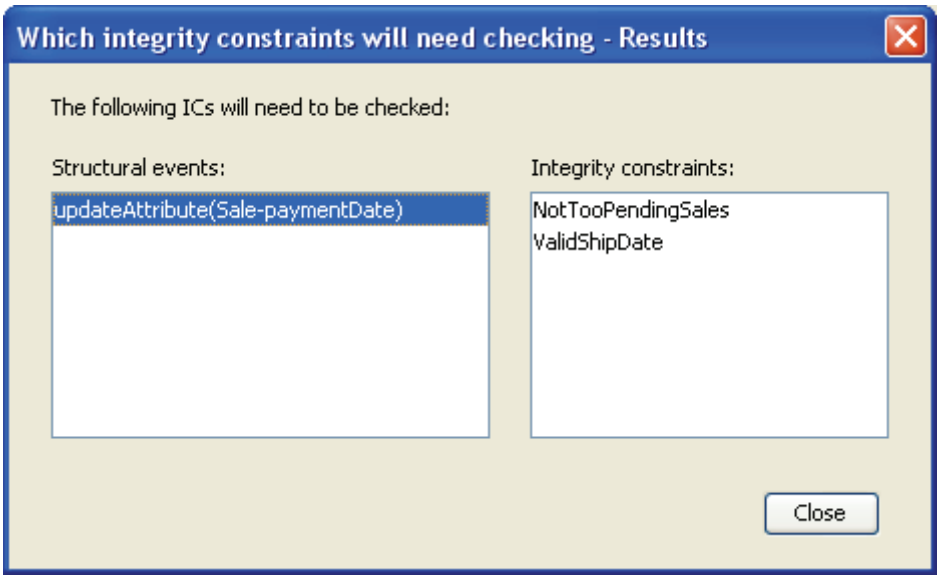


Figure 7.7. Information about the constraints that may be violated by the selected event type

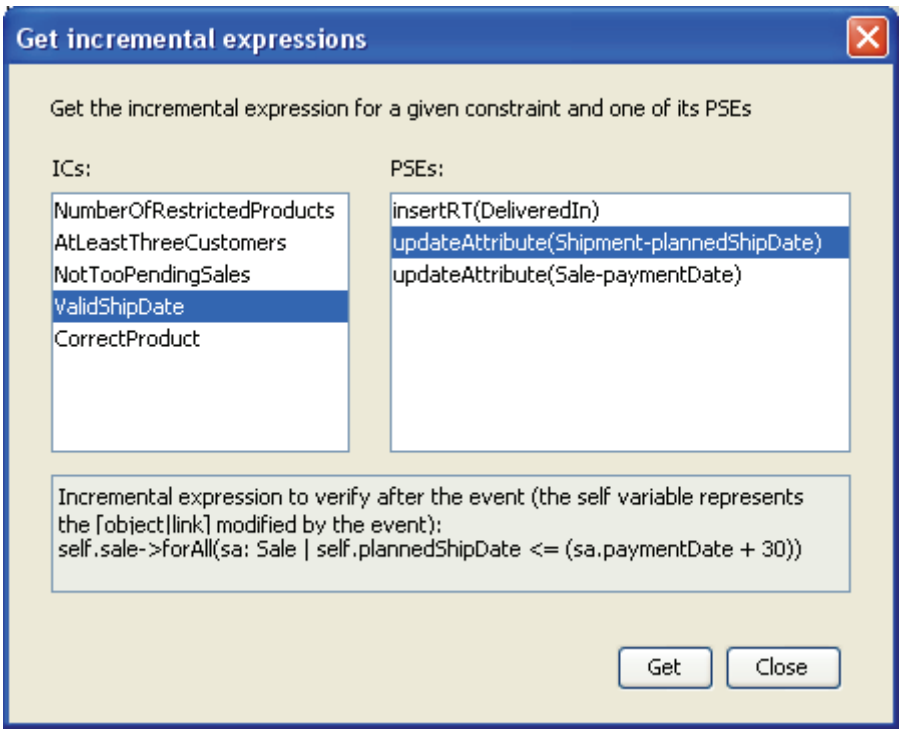


Figure 7.8. Incremental expression to verify *ValidShipDate* after UpdateAttribute(plannedShipDate,Shipment) events

Moreover, instead of generating the processed schema, the tool can also report the user about which OCL expression *exp* could be used instead of a constraint *c* in order to check *c* after the application of a single event over the IB [18]. Iff *exp* is satisfied by the entity affected by the event, the IB is still consistent with *c*. Otherwise the issue of the event has induced a constraint violation. As an example, the expression *exp* required to verify *ValidShipDate* after a *plannedShipDate* update over a shipment *sh* would be *sh.sale->forAll(sa: Sale | sh.plannedShipDate <= (sa.paymentDate + 30))* (see Figure 7.8; the *self* variable would correspond to the updated shipment *sh* in this example). We would like to remark that, with this functionality, the designer is the one in charge of generating an implementation of the schema that benefits from *exp* to incrementally check the constraint in the final technology platform (i.e., the designer is responsible for including the verification of *exp* whenever and wherever it is necessary).

8. Implementing the processed CS in a relational database

The main advantage of our method is that it is technologically-independent. This implies that any implementation of the processed CS results automatically in an incremental checking of all integrity constraints no matter the target technology platform. To show the feasibility of our approach, we show how a direct implementation of the processed CS in a relational database results in a database schema where constraints are checked incrementally.

The required steps to obtain the relational schema from our processed CS are the following:

1. Transformation of entity and relationship types
2. Transformation of structural event types
3. Automatic data maintenance for structural event types
4. Generation of derived subtypes
5. Generation of the integrity constraints

At the end of the last step, each constraint has been transformed into a relational view that returns those rows of the IB not verifying the constraint (an empty result indicates that the IB state is consistent with the constraint). Therefore, at the end of each IB update, we may detect if the new IB state satisfies all constraints and, when it does not, which are the constraints that do not hold and, for those constraints, which entities are the ones violating the constraint.

To be incremental, and according to the changes done to the original CS, the views corresponding to the processed constraints are defined over the population of the derived subtypes (also represented as views in the database). In its turn, the population of the derived subtypes is defined in terms of the population of the tables corresponding to the structural event types created to record the events issued during the IB update. Therefore, the process of querying the views to check the constraints is highly efficient since we only access a small portion of the whole IB (i.e. the part affected by the applied structural events).

The next subsections address each step. Each subsection is illustrated with partial examples. The full generation of the running example can be found at the end of the chapter.

We would like to remark that most of these transformation steps are not specific to our processed CS. Instead, any method generating a relational schema from a standard CS needs to deal with the same transformation steps, except for step 3.

8.1 Transformation of entity and relationship types

This first step is the most well-known. As usual (see [24],[10]), entity types are transformed into a corresponding set of tables (called *domain* tables) and relationship types are transformed into tables or foreign keys (depending on the multiplicities of the relationship type). For instance, the *Sale* table (Figure 8.1) represents the *Sale* entity type and the *Purchases* relationship type (represented by means of the *customer* column and the corresponding foreign key).

In a similar way, generalization/specialization relationships are also transformed as a foreign key between the subtype and the supertype (*vertical mapping* strategy [24]). As an example, see the creation of the tables *Product* and *RestrictedProduct* in Section 8.6.

```
create table Sale (
id Integer Primary Key, saledate Date, amount Decimal(8,2),
paymentDate Date,
Customer Integer REFERENCES Customer(id));
```

Figure 8.1. Sale *domain table*

8.2 Transformation of structural event types

Since structural event types are a special kind of entity types their basic transformation (as done for entity types) is to create a table for each structural event type. We call these tables *event tables*. The main specificity of event tables is that they must be empty at the beginning of each transaction. We can obtain this behavior automatically when defining the event tables as temporary tables (temporary tables are part of the SQL:1999 standard [58]).

Temporary tables are tables whose data is truncated at the end of each transaction (when specifying *on commit delete rows* in the table definition). Therefore, we create a temporary table for each structural event type. As for domain tables, the DBMS guarantees that concurrent users modifying the table do not interfere and that data is only visible to its own user.

Since structural event types merely contained references to the related entity type/s, the columns of event tables are just the set of columns required to store the primary key of the referenced type/s. The referenced type/s is not modified.

We neither define primary key nor foreign keys for event tables. It is not necessary since, as shown in the next section, their data is updated automatically, and thus, we can ensure their correctness.

As an example, Figure 8.2 shows the SQL sentence to create the temporary table corresponding to the structural event type *uSaleAmount*. The *id* column stores the *id* of the updated sale.

```
create global temporary table uSaleAmount (id Integer)
on commit delete rows;
```

Figure 8.2. Temporary table for the *UpdateAttribute(amount,Sale)* event type

8.3 Automatic updating of event tables

Data in these tables reflects the changes done by the user over the domain tables. Therefore, insertion and removal of tuples in event tables should be done automatically and transparently to the user.

If the relational database supports the definition of active rules (i.e. triggers) this automatic update can be addressed by means of monitoring the changes over the domain tables in order to record in the event tables the modifications issued by the user.

For each event table we create a trigger in the corresponding domain table. For *iET* (or *iRT* or *sET*) event tables we create an *after insert* trigger in the *ET* (*RT*) domain table. For *uETAttribute* event tables we create an *after update of attribute on ET* trigger over the ET table. For *dET* (or *dRT* or *gET*) event tables we create an *after delete* trigger over the *ET* (*RT*) table.

Triggers are defined as *after* triggers to avoid irrelevant insertions on the event tables. If the event issued by the user cannot be accomplished because of some exception raised by the database management system (for instance, due to a violation of a primary key) we do not need to register it.

As an example, to record in the event table *uSaleAmount* the references to sales updated during the transaction we define the trigger shown in Figure 8.3. For each row in the table *Sale* that has changed its value in the column *amount*, the trigger creates a new tuple in the *uSaleAmount* event table.

```
create trigger tuSaleAmount
after update of Amount on Sale
FOR each row
BEGIN
  Insert into uSaleAmount values (:new.id); END IF;
End;
```

Figure 8.3. Trigger for *UpdateAttribute(amount,Sale)* events

When working at the database level we must tackle the problem of distinguishing between insertion and specialization events over tables corresponding to subtypes in the conceptual schema. In both cases, these events end up inserting a new row in the domain table corresponding to the subtype entity type.

To detect if the new tuple in the subtype domain table appears because of an insertion event (event that must be recorded in the *iSubType* event table for integrity checking purposes) or because of an specialization event (that must be recorded in the *sSubType* event table) we must check if the related tuple in the supertype domain table has been inserted during the same transaction or it existed prior to starting the current transaction. In the former, the subtype insertion is due to an insertion event while in the latter is produced by a specialization event.

We know that the supertype entity existed already in the database when there is not a tuple in the *iSuperType* event table recording the insertion of that entity. As an example, consider the triggers for the event tables *iRestrictedProduct* and *sRestrictedProduct* in Section 8.6. Note that, this approach requires generating the event table for insertions over the supertype even if that event is not a PSE for any of the constraints.

In a similar way we may deal with the distinction between delete and generalization events. A deletion of a tuple in the subtype corresponds to a deletion event when the related tuple is also deleted from the supertype table and a generalization event otherwise. Note that, due to the foreign key between the subtype and the supertype, the user must delete first the tuple in the subtype and then, if necessary, the related tuple in the supertype. Then, to distinct both situations, we may initially consider all deletion events as generalization events and transform some of them to deletion events if the supertype tuple is also deleted afterwards.

Processing of events over relationship types that do not appear as a separate table in the database (for instance, the *BelongsTo* relationship type, represented by a foreign key in the *Customer* table) is slightly different. In such a case, triggers cannot be defined over the domain table (since it does not exist) and must be defined over the table containing the foreign key. We consider as insertions over the relationship type insertions on the table or updates of the foreign key column (the new value represents the new participant in the relationship). Deletions over the relationship type are induced by deletions over the table or

updates of the foreign key column (the old value refers to the participant of the deleted relationship).

As an example, Figure 8.4 shows the trigger in charge of monitoring the event *DeleteRT(BelongsTo)* (that must be recorded in the *dBelongsTo* event table) over the *Customer* entity type.

```
create or replace trigger tdBelongsTo
after delete or update of category on Customer
FOR each row
BEGIN
  Insert into dBelongsTo values (:old.category, :old.id);
End;
```

Figure 8.4. Trigger for *DeleteRT(BelongsTo)* events

When creating the triggers to populate the event tables we also need to take into account the rules proposed in Chapter 6 (Section 6.1.2) to minimize the population of the event tables. Their influence in the definition of the trigger effect is the following (see the examples in Section 8.6, including a new version for the trigger of Figure 8.4, extended according to these rules):

- Before inserting a tuple in an event table *uETAttrib* (recording updates over an attribute *attr* of an entity *e* of type *ET*) the trigger must check that *e* does not exist already in the same table or in the event table for the event *iET* (if existing).

- When deleting an entity or a relation, the trigger must delete the corresponding tuple in the event tables iET (or iRT), sET, gET and from all event tables uETAttribute, if existing. In addition, if the entity (relation) appeared in iET (iRT) the trigger do not need to record that it has been deleted in the dET (or dRT) table. This last scenario models a situation in which the same entity is first inserted and then deleted from the database during the same transaction.

We would also like to point out that these triggers only modify the state of the event tables and not the state of domain tables. Moreover, since no triggers are defined over the event tables, no termination problems occur. Confluence is also guaranteed (when an event fires more than one trigger each trigger modifies a different event table).

We have found useful to create triggers for all deletion events (even if the event is not a PSE for any constraint, and thus, we do not need to record it) in order to apply the second of the previous rules for avoiding verifications over inexistent objects. For instance, a trigger for deletions of sales could be useful to check if the removed sale has been previously inserted or updated during the same IB update. Since, afterwards, it has been deleted, the trigger may remove the corresponding tuples from the *iSale* and *uSaleAttrib* tables.

8.4 Generation of derived subtypes

Once the event tables have been created, we can define the derived subtypes as views over the data of the event tables referenced in the derivation rule of the subtype.

The view query expression may be obtained automatically from the OCL derivation rule of the derived subtype using existing translation patterns ([30], [24]).

Figure 8.5 shows the view corresponding to the *CustomerNotTooPendingSales*$_2$ subtype (see Figure 6.10). The solution may not be unique; depending on the translation patterns we may obtain different (equivalent) view definitions. The customers returned by the view are those appearing in the *iBelongsTo* and *iPurchases* event tables plus the ones related with sales included in the *uSaleAmount* and *uSalePaymentDate* event tables.

```
create view CustomerNotTooPendingSales2 as
select * from Customer where id IN
 ((select customer from iBelongsTo)  union
 (select customer from iPurchases) union
 (select s.customer from uSaleAmount u, Sale s where u.id=s.id) union
 (select s.customer from uSalePaymentDate u, Sale s where u.id=s.id));
```

Figure 8.5. View for the *CustomerNotTooPendingSales*$_2$ derived subtype

8.5 Generation of integrity constraints

A simple strategy when generating integrity constraints in a database is to generate them in the form of inconsistency predicates. 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. When defining constraints as inconsistency predicates we can report the user about the data that violates the constraint (the tuples retrieved querying the view are the ones violating the integrity constraint).

The SELECT clause of the view is generated from the constraint definition (in denial form) in a similar way as done for derived subtypes. To ensure an incremental verification of the constraint, the FROM clause of the view query is expressed in terms of the derived subtype used as a context type of the constraint.

As an example, Figure 8.6 shows the view corresponding to the *NotTooPendingSales2* constraint. Note that, since the constraint is defined over the view *CustomerNotTooPendingSales2*, it is only evaluated over the relevant customers (i.e. the ones affected by the events issued during the transaction). *Sysdate* is a predefined operation that returns the current date (equivalent to the *Time::now* operation used in the OCL definition of the constraint).

```
create view NotTooPendingSales2 as
select cu.* from CustomerNotTooPendingSales2 cu, Category c
where cu.category=c.name and c.maxPendingAmount <
   (select nvl(sum(s.amount),0) from sale s where s.customer=cu.id and
   s.paymentDate>sysdate);
```

Figure 8.6. View for the *NotTooPendingSales2* constraint

Class constraints (once modified as explained in Chapter 6, Section 6.2.3) are also transformed into views. However, for class constraints the views are defined over the domain table corresponding to the context type of the constraint. As an example, see the constraint *NumberOfRestrictedProducts* in Section 8.6.

Before committing the transaction, the user (or the application which is being executed) must query all generated views to check if all integrity constraints still hold. If any view is not empty, a violation occurred and the transaction must be rolled back (or an appropriate repair action must be triggered).

8.6. Transformation of the running example

In what follows we provide the complete implementation of the processed schema for our running example, as finally shown in Figures 6.8-6.12.

8.6.1 Creation of the domain tables

The following SQL scripts create the *domain tables* for the running example of Figure 1.1. These scripts (and all other scripts in this section) have been tested over an Oracle 9i database.

We use the *vertical mapping* strategy [24] for the *Product-RestrictedProduct* hierarchy.

```
-- Table representing the Category entity type
create table Category ( name varchar2(10) primary key, maxPendingAmount
Decimal(6,2), discount Decimal(6,2));

-- Table for the Customer entity type and the BelongsTo relationship type
(represented as a foreign key)
create table Customer ( id Integer Primary Key, name varchar2(30),
nationality char(3), creditCard char(10), category varchar2(10)
REFERENCES Category(name) not null);

-- Sale entity type and Purchases relationship type. The original date
attribute has been renamed to avoid conflicts with the Date reserved
word.
create table Sale ( id Integer Primary Key, saledate Date, amount
Decimal(8,2), paymentDate Date,Customer Integer REFERENCES Customer(id));

-- Shipment entity type
create table Shipment ( id Integer Primary Key, plannedShipDate Date,
address varchar2(50));
```

```
-- Table representing the DeliveredIn relationship type
create table DeliveredIn( sale Integer REFERENCES Sale(id),
shipment Integer REFERENCES Shipment(id), primary key (sale, shipment));

-- Product entity type
create table Product(
id Integer Primary Key REFERENCES Product (id), name varchar2(20), price
decimal(6,2), maxDiscount number(2), description varchar2(50));

-- RestrictedProduct entity type
create table RestrictedProduct(
id Integer Primary Key REFERENCES Product (id), maxUnits number(3));

-- SaleLine reified relationship type
create table SaleLine(
sale Integer REFERENCES Sale (id), product Integer REFERENCES Product
(id), quantity number(2), primary key (sale, product));
```

8.6.2 Creation of the event tables

For each event type appearing in Figures 6.8-6.12 we define its event table.

```
-- Table for InsertET(Category)
create global temporary table iCategory (id Varchar2(10)) on commit
delete rows;

-- Table for InsertET(Product)
create global temporary table iProduct (id Integer)on commit delete rows;

-- Table for InsertET(RestrictedProduct)
create global temporary table iRestrictedProduct (id Integer) on commit
delete rows;

-- Table for InsertRT(DeliveredIn)
create global temporary table iDeliveredIn
(Sale Integer, Shipment Integer) on commit delete rows;

-- Table for InsertRT(BelongsTo)
create global temporary table iBelongsTo
(Category Varchar2(10), Customer Integer) on commit delete rows;

-- Table for InsertRT(Purchases)
create global temporary table iPurchases
(Customer Integer, Sale Integer) on commit delete rows;

-- Table for SpecializeET(RestrictedProduct)
create global temporary table sRestrictedProduct (id Integer) on commit
delete rows;

-- Table for UpdateAttribute(amount, Sale)
create global temporary table uSaleAmount (id Integer)
on commit delete rows;

-- Table for UpdateAttribute(paymentDate, Sale)
create global temporary table uSalePaymentDate (id Integer) on commit
delete rows;
```

```
-- Table for UpdateAttribute(plannedShipDate,Shipment)
create global temporary table uShipmentPlannedShipDate (id Integer)
on commit delete rows;

-- Table for UpdateAttribute(price, Product)
create global temporary table uProductPrice (id Integer)
on commit delete rows;

-- Table for UpdateAttribute(maxDiscount, Product)
create global temporary table uProductMaxDiscount (id Integer)
on commit delete rows;

-- Table for UpdateAttribute(maxPendingAmount, Category)
create global temporary table uCategoryMaxPendingAmount (id Varchar2(10))
on commit delete rows;

-- Table for DeleteRT(BelongsTo)
create global temporary table dBelongsTo
(Category varchar2(10), Customer Integer) on commit delete rows;
```

8.6.3 Triggers for the automatic update of event tables

- Triggers for InsertET, InsertRT and SpecializeET events

```
-- Trigger for insertions over the iProduct table
create trigger tiProduct after insert on Product
FOR each row
BEGIN  Insert into iProduct values (:new.id); End;

-- Trigger for insertions over the iRestrictedProduct table. We check
whether the insertion on RestrictedProduct is due to an specialization
event or an insertion event
create trigger tiRestrictedProduct after insert on RestrictedProduct
FOR each row
DECLARE v_ExistsU number;
BEGIN
   Select count(*) into v_ExistsU From iProduct where id=:new.id;
   IF (v_ExistsU>0) THEN Insert into iRestrictedProduct values (:new.id);
END IF;
End;

-- Trigger for insertions over the sRestrictedProduct table
create trigger tsRestrictedProduct after insert on RestrictedProduct
FOR each row
DECLARE v_ExistsU number;
BEGIN
   Select count(*) into v_ExistsU From iProduct where id=:new.id;
   IF (v_ExistsU=0) THEN Insert into sRestrictedProduct values (:new.id);
END IF;
End;

-- Trigger for insertions over the iCategory table
create trigger tiCategory after insert on Category
FOR each row BEGIN  Insert into iCategory values (:new.name); End;

-- Trigger for insertions over iDeliveredIn
create trigger tiDeliveredIn after insert on DeliveredIn
```

```
FOR each row
BEGIN Insert into iDeliveredIn values (:new.sale, :new.shipment); End;

-- Trigger for insertions over iBelongsTo
create trigger tiBelongsTo after insert or update of category on Customer
FOR each row
BEGIN  Insert into iBelongsTo values (:new.category,:new.id); End;

-- Trigger for insertions over iPurchases
create trigger tiPurchases after insert or update of customer on Sale
FOR each row
BEGIN  Insert into iPurchases values (:new.customer,:new.id); End;
```

- Triggers for update attribute events

```
-- Trigger for insertions over the uSaleAmount table
create trigger tuSaleAmount
after update of Amount on Sale FOR each row
DECLARE v_ExistsU number;
BEGIN
   Select count(*) into v_ExistsU From uSaleAmount where id=:new.id;
   IF (v_ExistsU=0) THEN Insert into uSaleAmount values (:new.id);END IF;
End;

-- Trigger for insertions over the uSalePaymentDate table
create trigger tuSalePaymentDate
after update of PaymentDate on Sale FOR each row
DECLARE v_ExistsU number;
BEGIN
   Select count(*) into v_ExistsU From uSalePaymentDate where id=:new.id;
   IF (v_ExistsU=0) THEN Insert into uSalePaymentDate values (:new.id);
END IF;
End;

-- Trigger for insertions over the uShipmentPlannedShipDate table
create trigger tuShipmentPlannedShipDate
after update of PlannedShipDate on Shipment FOR each row
DECLARE v_ExistsU number;
BEGIN
   Select count(*) into v_ExistsU From uShipmentPlannedShipDate where
id=:new.id;
   IF (v_ExistsU=0) THEN Insert into uShipmentPlannedshipDate values
(:new.id);
   END IF;
End;

-- Trigger for insertions over uProductPrice
create trigger tuProductPrice
after update of Price on Product FOR each row
DECLARE v_ExistsI NUMBER; v_ExistsU number;
BEGIN
   Select count(*) into v_ExistsI From iProduct where id=:new.id;
   Select count(*) into v_ExistsU  From uProductPrice where id=:new.id;
   IF (v_ExistsU=0) and (v_ExistsI=0)  THEN
       Insert into uProductPrice values (:new.id);
   END IF;
End;
```

```
-- Trigger for insertions over uProductMaxDiscount
create trigger tuProductMaxDiscount
after update of MaxDiscount on Product FOR each row
DECLARE v_ExistsI NUMBER; v_ExistsU number;
BEGIN
   Select count(*) into v_ExistsI From iProduct where id=:new.id;
   Select  count(*)  into  v_ExistsU    From  uProductMaxDiscount  where
id=:new.id;
   IF (v_ExistsU=0) and (v_ExistsI=0)  THEN
       Insert into uProductMaxDiscount values (:new.id);
   END IF;
End;

-- Trigger for insertions over the uCategoryMaxPendingAmount table
create trigger tuCategoryMaxPendingAmount
after update of MaxPendingAmount on Category FOR each row
DECLARE v_ExistsI NUMBER; v_ExistsU number;
BEGIN
   Select count(*) into v_ExistsI From iCategory where id=:new.name;
   Select  count(*)  into  v_ExistsU   From uCategoryMaxPendingAmount where
id=:new.name;
   IF (v_ExistsU=0) and (v_ExistsI=0)  THEN
       Insert into uCategoryMaxPendingAmount values (:new.name);
   END IF;
End;
```

- Triggers for DeleteET, DeleteRT and GeneralizeET events

```
-- Trigger for deletions over the BelongsTo relationship type. BelongsTo
is represented as a foreign key in the Customer table
create or replace trigger tdBelongsTo
after delete or update of category on Customer FOR each row
DECLARE v_ExistsI NUMBER;
begin
   Select count(*) into v_ExistsI From iBelongsTo where customer=:old.id
and category=:old.category;
   if (v_ExistsI=0)  then
     Insert into dBelongsTo values (:old.category, :old.id);
  else   delete   from   iBelongsTo   where   category=:old.category   and
customer=:old.id;
   end if;
End;

-- Triggers to delete unnecessary rows of the structural event types
after delete SQL sentences. These triggers do not correspond to any PSE
but improve the efficiency of the final schema by means of removing
irrelevant tuples from the event tables.
create or replace trigger tdProduct
after delete on Product FOR each row
begin
  Delete from iProduct where id=:old.id;
  Delete from uProductPrice where id=:old.id;
  Delete from uProductMaxDiscount where id=:old.id;
End;

create or replace trigger tdRestrictedProduct
after delete on RestrictedProduct FOR each row
BEGIN
```

```
  Delete from iRestrictedProduct where id=:old.id;
  Delete from sRestrictedProduct where id=:old.id;
End;

create or replace trigger tdCategory
after delete on Category FOR each row
BEGIN
  Delete from iCategory where id=:old.name;
  Delete from uCategoryMaxPendingAmount where id=:old.name;
End;

create or replace trigger tdSale
after delete on Sale FOR each row
BEGIN
  Delete from uSaleAmount where id=:old.id;
  Delete from uSalePaymentDate where id=:old.id;
  Delete from iPurchases where sale=:old.id;
End;

create or replace trigger tdShipment
after delete on Shipment FOR each row
BEGIN
  Delete from uSaleAmount where id=:old.id;
  Delete from uSalePaymentDate where id=:old.id;
End;

create or replace trigger tdDeliveredIn
after delete on DeliveredIn FOR each row
BEGIN    Delete    from    iDeliveredIn    where    sale=:old.sale    and
shipment=:old.shipment; End;
```

8.6.4 Definition of derived subtypes

```
-- View representing the SaleValidShipDate derived subtype
create view SaleValidShipDate as
select * from Sale where id IN (select * from uSalePaymentDate);

-- View representing the ShipmentValidShipDate2 derived subtype
create view ShipmentValidShipDate2 as
select    *    from    Shipment    where    id    IN    (select    *    from
uShipmentPlannedShipDate);

-- View representing the DeliveredInValidShipDate3 derived subtype
create view DeliveredInValidShipDate3 as
select * from DeliveredIn where (sale,shipment) IN (select sale,shipment
from iDeliveredIn);

-- View representing the ProductCorrectProduct derived subtype
create view ProductCorrectProduct as
select * from Product where id IN (select * from iProduct);

-- View representing the ProductCorrectProduct2 derived subtype
create view ProductCorrectProduct2 as
select * from Product where id IN (select * from uProductPrice);

-- View representing the ProductCorrectProduct3 derived subtype
create view ProductCorrectProduct3 as
select * from Product where id IN (select * from uProductMaxDiscount) ;
```

```
-- View representing the CategoryNotTooPendingSales derived subtype
create view CategoryNotTooPendingSales as
select * from Category where name IN (select * from
uCategoryMaxPendingAmount) ;

-- View representing the CustomerNotTooPendingSales2 derived subtype
create view CustomerNotTooPendingSales2 as
select * from Customer where id IN
 ((select customer from iBelongsTo)  union
 (select customer from iPurchases) union
 (select s.customer from uSaleAmount u, Sale s where u.id=s.id) union
 (select s.customer from uSalePaymentDate u, Sale s where u.id=s.id));

-- View representing the CategoryAtLeastThreeCustomers derived subtype
create view CategoryAtLeastThreeCustomers as
select * from Category where name IN
((select * from iCategory) union (select category from dBelongsTo));
```

8.6.5 Definition of the integrity constraints

```
-- View representing the ValidShipDate constraint
create view ValidShipDate as
select * from SaleValidshipDate s where exists
 (select * from DeliveredIn d, Shipment sh where s.id=d.sale and
d.shipment=sh.id and
  not sh.plannedshipDate<=s.paymentDate+30);

-- View representing the ValidShipDate2 constraint
create view ValidShipDate2 as
select * from ShipmentValidshipDate2 sh where exists
 (select * from DeliveredIn d, Sale s where s.id=d.sale and
d.shipment=sh.id and
  not sh.plannedshipDate<=s.paymentDate+30);

-- View representing the ValidShipDate3 constraint
create view ValidShipDate3 as
select d.* from DeliveredInValidShipDate3 d, Shipment sh, Sale s where
d.sale=s.id and d.shipment=sh.id and not
sh.plannedshipDate<=s.paymentDate+30;

-- View representing CorrectProduct
create view CorrectProduct as
select * from ProductCorrectProduct p where not (p.price>0 and
p.maxDiscount<=60);

-- View representing CorrectProduct2
create view CorrectProduct2 as
select * from ProductCorrectProduct2 p where not (p.price>0);

-- View representing CorrectProduct3
create view CorrectProduct3 as
select * from ProductCorrectProduct3 p where not (p.maxDiscount<=60);

-- View representing NotTooPendingSales
create view NotTooPendingSales as
select * from CategoryNotTooPendingSales c where exists (
```

```
  select * from customer cu where cu.category=c.name and (select
sum(s.amount) from sale s where s.customer=cu.id and
s.paymentDate>sysdate) > c.maxPendingAmount);

-- View representing NotTooPendingSales2
create view NotTooPendingSales2 as
select cu.* from CustomerNotTooPendingSales2 cu, Category c where
cu.category=c.name and (select nvl(sum(s.amount),0) from sale s where
s.customer=cu.id and s.paymentDate>sysdate) > c.maxPendingAmount;

-- View representing AtLeastThreeCustomers
create view AtLeastThreeCustomers as
select * from CategoryAtLeastThreeCustomers c where not ((select count(*)
from customer cu where cu.category=c.name)>=3);

-- View representing NumberOfRestrictedProducts
-- If there is a tuple in iRestrictedProduct or sRestrictedProduct we
verify the constraint. The dual table is an auxiliary table defined by
Oracle containing a single tuple with empty values. We use this table to
show the error message.
create view NumberOfRestrictedProducts as
select 'ErrorInNumberOfRestrictedProductsConstraint' as error from dual
where exists(select * from iRestrictedProduct union select * from
sRestrictedProduct) and (select count(*) from RestrictedProduct)>20;
```

9. Related work

Two kinds of related work are relevant to this thesis. On the one hand, there is a long tradition of methods devoted to the problem of integrity checking in the database field (see Section 9.1). We will show that the efficiency of the incremental checking we get with our processed schema is comparable to the efficiency obtained with those methods.

On the other hand there are an increasing number of methods and tools that provide code-generation capabilities from CSs and that may include facilities for providing an efficient integrity checking mechanism when generating the implementation of the CS in the final technology platform (see Section 9.2). We will show that the efficiency provided by these methods is worse than the efficiency that could be obtained if directly implementing our processed schema. This is true even when comparing the efficiency of the generated implementations only for those technology platforms that these existing methods and tools are focused on. Besides, all of them always depart from the integrity constraints exactly as defined by the designer, and thus, the quality of their results depends on the particular syntactic definition of the constraint chosen by the designer when specifying the CS.

In what follows we compare our method with the most representative proposals of both groups. The comparison is based on the expressiveness of the constraint definition language (remember the classification between *intra*, *inter*, and *type* constraints in Section 1.3) and on the efficiency of the provided techniques for integrity checking.

9.1 Approaches in the database field

A lot of research has been devoted to the problem of guaranteeing the consistency of the database data with respect to the integrity constraints defined in the database schema.

There are different perspectives to deal with this problem. We can classify the methods according to the checking time (run-time or compile-time) and to the kind of response to constraint violations (checking or maintenance). Thus, we have four different families of methods [53]:

- Integrity checking at run-time: With this approach, whenever a transaction is to be committed, all integrity constraints are verified. If a constraint is violated in the new database state, the transaction is aborted.

- Integrity checking at compile-time: Theorem provers or proof assistants are applied over the set of predefined transactions that may be applied over the database in order to ensure that their application will never induce a constraint violation. This implies that at run-time it is not necessary to verify the constraints but, on the other hand, we must foresee all possible transactions that may be applied over the IB. Moreover, current methods usually restrict the kind of predefined transactions that the user may define.

- Integrity maintenance at run-time: When, before committing a transaction, the method detects that a constraint is not satisfied, additional modification events (also called repair actions) are generated in order to ensure that the database is left in a consistent state. ECA-rules and derivation rules are the most common mechanisms used to detect violations and generate repair actions. The method must ensure that the rule application terminates and that the generated updates preserve the semantic effect of the original transaction.

- Integrity maintenance at compile-time: The set of predefined transactions are analyzed at compile-time and extended with a set of appropriate repair actions. Again, the method must guarantee that the process terminates. Confluence is also desirable.

The approach closest to our own method is integrity checking at run-time (our processed constraints are checked at run-time and we do not provide any mechanism to generate the corresponding repair actions). Therefore, all methods included in the following comparison follow (or can be adapted to) this approach. For relevant references regarding the other families of approaches, we refer to [55], [86] (maintenance at run-time) and to [77] and [53] (maintenance at compile-time).

In general, all methods present two main limitations as compared to the one proposed in this thesis: 1 – limitations with respect to the expressivity of the constraint definition language or 2 – limitations with respect to the kind of structural event types considered.

Regarding the first problem, some of the proposals do not allow the definition of aggregate operators, select expressions or bag semantics, which frequently appear in the definition of OCL constraints.

Regarding the second one, most of them only consider insertion and deletion events over relations (which may correspond either to an entity type or a relationship type of the CS). Our richer set of event types allows us to provide more fine-grained results. For instance, assume that we distinguish between two kinds of event types *X* and *Y* while one of these previous methods *m* mixes them in a single event type *Z*. When applying *m* over a constraint *c* we may obtain that *c* must be verified after applying an event of type *Z*. Therefore, we will have to verify *c* after all database updates including *X* or *Y*, even if

possibly, only the operations including *X* (or *Y*) may really violate *c*, as it could be detected by our method.

For the sake of clarity, when presenting the different methods we distinguish among methods developed for relational databases, methods for deductive databases and methods for object(-relational) databases. We are aware that this classification is not necessarily strict.

9.1.1. Approaches for relational databases

Probably, the most relevant proposal in this field is [21]. In this proposal, constraints are defined as SQL predicates over the database state. They generate a production rule (similar to the trigger concept in current database systems) for each constraint. The rule is executed whenever a PSE for the constraint is applied over the database. When the rule is fired, the constraint condition is evaluated. If the condition does not hold a given *action* is taken. These *actions* are not automatically generated, they must be manually defined by the designer and may be as simple as rolling back the constraint.

As far as expressivity is concerned, this method is powerful enough to deal with all kind of constraints we can define in OCL (it supports all SQL constructs that would be required to map the different OCL operators into SQL ones).

However, this method lacks of precision when computing the PSEs for the constraints since it may consider as a PSEs for a given constraint certain structural events that may never violate it. For instance, the method would determine that the insertion of a new shipment may violate the constraint *ValidShipDate* of our example (Figure 1.2), when it is not the insertion of a shipment but the insertion of a new relationship between a shipment and a sale what can really violate it.

Another drawback of this method is that not all constraints are checked incrementally. Depending on the constraint, a complete recomputation of the constraint is required to check that the IB satisfies it. A constraint *c* defined over a type *t* can be incrementally checked only when all operators appearing in *c* are operators over attributes or relationships of *t* and no recursive navigations exist over *t*.

Lately, in [23], this method is improved to incrementally check all constraints but, as a trade-off, they need to restrict the constraint definition language (for instance, no aggregate operators can be used).

9.1.2 Approaches for deductive databases

There is a long tradition of methods devoted to the problem of integrity constraint checking in the deductive database field.

Some of the approaches do not focus on the problem of integrity checking itself but on the related problem of materialized view maintenance. A materialized view is a view whose tuples are stored in the database instead of being recomputed every time the view is queried. Then, the view materialization problem aims at incrementally updating the view data in response to changes in the tables underlying the view definition. Since integrity constraints can be represented as inconsistency predicates, i.e. as views that must be empty (a non-empty view indicates that the corresponding constraint has been violated), the problem of integrity checking can be regarded as a subset of the view maintenance problem.

These approaches share a similar core mechanism. They all represent integrity constraints as inconsistency predicates. As an example, the representation of *ValidShipDate* as an inconsistency predicate would be the following (where *S* stands for *Sale*, *Sh* for *Shipment, D* for *DeliveredIn*, *pd* for *paymentDate* and *psh* for *plannedShipDate*):

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

where $Ic_{ValidShipDate}$ becomes non-empty (i.e. it is violated) whenever two related sale and shipment instances verify that the *plannedShipDate* goes beyond 30 days after the *paymentDate* of the sale.

Then, these methods propose a set of rules to control the insertions over the predicate representing the constraint (the predicate $Ic_{ValidShipDate}$ in our example). Each rule identifies a situation that could possibly induce a constraint violation. Whenever one of the rules is found to be true, the constraint is considered violated. The methods differ in terms of the precision and efficiency of the rules they propose.

A common drawback of most of these proposals is that they are not powerful enough to deal with the whole expressiveness of the OCL (i.e. they do not support the constructs that would be required to map the OCL constraints to the logic language they use as a constraint definition language), as done by our method. In particular, OCL allows negation, bag semantics and aggregation operations while these methods hardly cover all these constructs (see [41] for a survey and a general discussion of their limitations).

In the following we present the most representative approaches in this field. Most recent approaches do not improve the efficiency of the results but adapt the methods to different contexts as: 1- distributed databases where we may not have access to the original materialized view when computing the changes required to maintain the view [82] or 2 - environments with autonomous data sources, where the result of the incremental computation is affected by interfering updates [83]. We would also like to remark the proposal of [29], where some of the ideas presented in the next subsection are adapted to be integrated within the technology possibilities provided by current database systems.

9.1.2.1 Urpí and Olivé's Method

This proposal [88],[89] is based on the computation of the insertion, deletion and modification internal events over a derived predicate. This derived predicate may represent the inconsistency predicate corresponding to an integrity constraint. In such a case, only insertion events over the derived predicate are relevant (the predicate is always empty when beginning the transaction).

Roughly, an insertion event over a derived predicate (i.e. indicating a violation of the constraint in our case) may be generated when the transaction includes a structural event that makes true a predicate p_i appearing in the rule body for the derived predicate. The possible structural events are an insertion event, a modification event or a deletion event (when in the rule body, p_i appears negated).

After p_i becomes true we have to evaluate the derived predicate to check whether now the whole rule body holds, and thus, a new fact must be inserted in the derived predicate. Since [89] the method takes into account inclusion dependencies, exclusion dependencies, alternative keys and referential integrity constraints among the different predicates p_i when proposing the rules that may generate an insertion event over the derived predicate.

Applied to the previous *ValidShipDate* constraint, the method would generate the following set of rules to incrementally check the constraint:

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

where *iX(y)* means that the entity *y* of type *X* has been inserted and *uX* means that it has been updated.

After applying our method to the same constraint, we obtain the following three incremental constraints (see Figure 6.8):

a. **context** SaleValidShipDate **inv**:
   self.shipment->forAll(sh| sh.plannedShipDate<=self.paymentDate+30)
b. **context** $ShipmentValidShipDate_2$ **inv**:
   self.sale->forAll(s| self.plannedShipDate<=s.paymentDate+30)
c. **context** $DeliveredInValidShipDate_3$ **inv**:
   self.shipment.plannedShipDate<=self.paymentDate+30

where *SaleValidShipDate* was created to hold the set of sales that have been updated, *ShipmentValidSihpDate* contains the modified shipments and *DeliveredInValidShipDate* the inserted *DeliveredIn* relationships.

When comparing both results, we may realize that the constraints we get are equivalent to rules 1, 3 and 2, respectively. Constraint *a* checks only the updated sales, as rule number 1, and for each updated sale both methods compare its *paymentDate* with the *plannedShipDate* of all its shipments. Similarly, constraint *b* and rule 3 are only evaluated over updated shipments, and for each one, they verify that their *plannedShipDate* is correct with respect to the *paymentDate* of the related sales. Finally, both, constraint *c* and rule 2, consider just the sale and the shipment participating in the inserted *DeliveredIn* relationship.

The main limitation of this method is that all constraints must be specified as closed first-order formulae, not expressive enough to represent all constraints that can be specified with OCL. As an example, among the five integrity constraints of our running example, only *ValidShipDate* and *CorrectProduct* can be handled by this method.

Our method can also be more efficient in the treatment of constraints with existential quantifiers after insertion or updating events. As an example, consider a constraint stating that at least a sale per shipment must be of an amount greater than 10000. After every single sale update resulting in a new amount under 10000, this method would check that at least another sale of the same shipment is still over 10000. Hence, a certain shipment *sh* may be reconsidered several times, one for each updated sale assigned to *sh*. On the contrary, we first compute the set of affected shipments after all sale updates and then check the constraint condition on each of them. In this way, we avoid rechecking the same shipment several times.

On the contrary, we must recognize that our method behaves worse than this method after deletion events in constraints whose definition includes a negated atomic literal representing a relationship type *RT*. For instance, this happens for constraints defining that every instance of an entity type *A* must be related with all instances of an entity type *B*. An OCL representation of such constraint could be: *context A inv: self.b->size()=B.allInstances()*. Its representation as an inconsistency predicate would be: $Ic \leftarrow \forall a,b\ A(a) \wedge \neg RT(a,b) \wedge B(b)$

In this situation, this method directly detects that the constraint is violated after the deletion of a relationship of RT not followed by the deletion of the corresponding instance in *B* while our method requires to check that the *A* participant of the deleted relationship is still related with all entities of the *B* entity type.

9.1.2.2 The Counting algorithm

The counting algorithm (included in [42]) can be used to maintain views that use negation, aggregation, bag semantics (i.e. views with duplicates) and the union operator. It neither supports recursive views nor the difference set operator.

Similarly to the previous method, it associates *n* delta rules to each derived predicate *p* (where *n* is the number of predicates p_i appearing in the body of the derivation rule for *p*). Each delta rule computes the changes over *p* due to the changes (insertions and deletions) over p_i during the transaction.

The application of the method over *ValidShipDate* produces the following set of delta rules. For the sake of simplicity we use the same notation as in the previous method.

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

There are some noteworthy differences with respect to our method. First of all, this method only distinguishes between insertion and deletion events, modification events are modeled as a deletion event followed by an insertion event. This hinders the precision of the computation of constraint violations. For instance, an update of the *amount* attribute of a sale *s* is transformed as a deletion of *s* plus an insertion of *s* with the new value in the *amount* attribute. Since we have generated an insertion event over *Sale*, according to rule 1, *ValidShipDate* will be verified by comparing the payment date of *s* with the planned ship date of all shipments related with *s*. Clearly, this is an unnecessary verification since the update of a sale amount (which was the original event intended by the user) cannot violate *ValidShipDate*.

Another difference is the behavior with respect to the aggregate operators. The counting algorithm is able to keep track of additional derived information stored in the entity types of the CS. For instance, we could record in the *Category* type the number of customers assigned to that category. Then, when verifying *AtLeastThreeCustomers*, this algorithm could detect whether the constraint is violated after the removal of a customer from a category *c*, without recalculating the number of customers still belonging to *c* (as required in our method). Thus, they improve efficiency of integrity checking but need to incur in the extra cost of having to materialize and keep up to date such derived information (i.e. after the customer removal, the value of the number of customers for the category *c* must be decreased).

9.1.2.3 The DRed algorithm and the Ceri and Widom's method

The *DRed* algorithm (included in [42]) incrementally maintains recursive views that use negation and aggregation but does not support bag semantics.

This algorithm is a three phase algorithm. In the first place, it deletes from the derived predicate all tuples related with deleted predicates during the transaction (even if there exist some other derivation that makes the derived predicate still true in the new state). Then, it puts back all the tuples that have alternative derivations. Finally, it inserts in the derived predicate tuples due to insertion events over the base predicates appearing in the rule body. To handle views defined by a recursive rule, the method applies these three

steps successively. The Ceri and Widom's method [22] (do not confuse with their method for relational databases reviewed in section 9.1.1) follows a similar approach but it does not support aggregate operations

Note that, in our case, only the third step is relevant (and similar to the previous methods) since the view corresponding to an integrity constraint is always empty before starting the database update.

The main limitations of this method are its lack of support for update events (see the comment in the previous section) and the redundant computation of aggregate operators. An aggregate (or exists) operator, as the *size* operator in the *AtLeastThreeCustomers* constraint, is computed after each customer removal to verify the category where the customer belonged still verifies the constraint. Instead, we first compute which is the set of affected categories and compute the operator over them. This way, when several customers belonged to the same category we avoid computing several times the size operator.

9.1.2.4 Christiansen and Martinenghi's method

This approach [26] is quite different from the ones reviewed up to now. Unlike previous approaches, which only consider single updates, this method is applied to predefined sets of events provided by the designer at compile-time. These sets of events are called *parametric transaction patterns*.

Once a specific transaction is proposed, and before it is executed, the corresponding pattern is instantiated and checked for consistency so that only consistency-preserving transactions are eventually passed on to the database.

The basic idea of the method is that when considering a transaction pattern instead of a single event, the rules generated to verify a given constraint may be more efficient than rules generated considering isolated events. Obviously, this forces the designer to provide in advance all relevant transaction patterns. An ad-hoc update of the database cannot benefit from this patterns, and thus, it would suffer from all problems commented for previous methods.

Another problem of the method is that it must face undecidability results since this approach has a direct correspondence with the query containment problem (which is also known to be undecidable in general). This makes impossible to achieve a general and optimal solution for all cases [26].

9.1.3 Approaches for object(-relational) databases

Few approaches are specific for object-oriented or object-relational databases [57]. Due to the use of an object model, they must face new problems, mainly dereferencing the references between objects, unnesting nested relations and handling inheritance relationships between the different relations.

[54] introduces an algorithm for incrementally maintaining views for object-relational databases. The object-relational view definition language may contain aggregates but it presents some limitations regarding the presence of nested select statements. The only events supported are insertion and deletion events. It neither distinguishes between specialization and insertion events either (nor between generalization and deletion events).

As a first step, the method replaces all object references with explicit joins. Then it creates two triggers for each table *t* appearing in the view query, one for insertions over *t* and the other for deletions. In this step, tables inheriting from *t* and inherited tables of *t* are also included. Then, whenever an insert (deletion) in *t* is detected the insert (deletion) trigger evaluates the atomic conditions in the view query affecting *t* and when the inserted (deleted) tuple satisfies them, the trigger activates the view maintenance algorithm. Finally, the view maintenance algorithm consists of joining the inserted (deleted) tuples of *t* with the rest of the tables in the view query. The result set is added (removed) from the view.

Note that the method always creates both triggers, without considering at this phase of the process if the kind of event may really violate the constraint.

To deal with views with aggregate operators, they first remove the operators from the view and apply the previous procedure. Next, the aggregate operators are applied over the result set and merged with the existing tuples in the view. Therefore, the semantics of the aggregate operators are not used to provide a better incremental maintenance.

[6] proposes a solution to the problem of incrementally maintaining (a subset of) materialized OQL views defined over an ODMG-compliant schema. In particular, the views may not contain aggregate operators, set operators (union, intersection, difference) nor duplicates (i.e. bag semantics). The view is transformed to an algebraic form.

When computing the events to be monitored (the ones that may change the view population) they lack of precision since they assume that all kinds of events over the elements referenced in the view (tables, object references, attributes) affect the view.

Then, for each event type they generate an incremental maintenance plan. The article provides a thorough discussion on the efficiency of the generated plans when applied over a real database and how its efficiency is influenced by the size of the tables, the selectivity of the view, the ratio of update events against query events and so forth.

9.2 Comparison with current CASE, MDA and MDD tools

We have recently witnessed an explosion of tools and methods promising a full and automatic generation of the application code from its specification. Even more, nowadays, code-generation capabilities of CASE tools are a key issue in their development and marketing strategy.

At the present moment, almost all methods and tools are able to generate the skeleton of Java classes or relational schemas from the CS. A few also generate the code of the application operations when its behavior is specified with state diagrams or action semantics [68].

Nevertheless, most methods and tools tend to skip the integrity constraints specified in the schema when generating the system implementation. All of them present important limitations regarding the expressivity of the constraints they can handle and/or the efficiency of the generated code [19].

Since they do not work purely at a conceptual level (as a result they do not provide a processed conceptual schema but the translation of the schema in terms of the target technology) to study their constraint code-generation capabilities we must examine the efficiency of the generated implementation in the final technology platform. We will focus on these two technologies: 1 – Relational databases and 2 – Object-oriented languages, in particular Java. Even though some tools also deal with other technologies (like .NET or C++), this decision does not restrict the set of tools to study since these two technologies are the most widely covered ones.

We have chosen the most representative examples from all different kinds of tools (from CASE tools extended with code-generation capabilities to full model-driven development methods). For each group we have selected the tools we believe are the most representative or the ones offering a better constraint support. Obviously, this classification is somewhat arbitrary and some of the tools could be classified in more than category. Moreover, we have included in the study all tools supporting a textual language to define integrity constraints, commonly OCL or similar. Support for such a language is required in order to be able to specify all possible kinds of constraints in a conceptual schema [33].

At the end, we provide a summary table for all methods and tools mentioned in this chapter.

9.2.1 CASE Tools

Even though the initial goal of CASE tools was to facilitate the modeling of software systems, almost all of them have extended their functionality to offer, at least to some extent, code-generation capabilities. From all CASE tools (see [63] for an exhaustive list) we have selected the following ones: *Poseidon*, *Rational Rose*, *MagicDraw, Objecteering/UML* and *Together*. In what follows we comment them in detail:

a) *Poseidon* [37] is a commercial extension of *ArgoUML* [2]. Its Java generation capabilities are quite simple. It does not allow the definition of OCL constraints and it does not take the multiplicity constraints into account either. Only distinguishes two different multiplicity values: ‘one’ and ‘greater than one’. In fact, when the multiplicity is greater than one the values of the multivalued attributed created in the corresponding Java class

are not restricted to be of the correct type (see in Figure 9.1 the *customer* attribute of the *Category* class corresponding to the *Category* entity type defined in our running example; the *customer* attribute could hold any kind of objects and not just customer instances).

The generation of the relational schema is not much more powerful either. Just the *primary keys* constraints are supported. The designer must explicitly indicate which attributes act as a primary key for the entity type by means of modifying the corresponding property in the attribute definition.

```
public class Category {

   private String name;
   private int maxPendingAmount;
   private int discount;
   public java.util.Collection customer = new java.util.TreeSet();
}

public class Customer {

   private int id;
   private String name;
   private String nationality;
   private String creditCard;
   public Category category;
   public java.util.Collection sale = new java.util.TreeSet();
}
```

Figure 9.1. *Category* and *Customer* classes as generated by Poseidon

b) *Rational Rose* [75]. The Java generation process is similar to that of *Poseidon*. The database generation is better because the class diagram can be complemented with the definition of additional properties. For instance, the *CorrectProduct* constraint can be specified as a property of the *price* (Figure 9.2) and *maxDiscount* attributes. Given this information, the tool adds to the *Product* table the constraint *check(price>0)* to control the correct product price (and likewise with *maxDiscount*). Unfortunately, the allowed expressivity of these additional properties is limited to simple restrictions over the values of individual attributes.

Recently, a Rational Rose plug-in [34] is available to permit the definition of OCL constraints on rose models. However, these constraints are not considered when generating the application code.

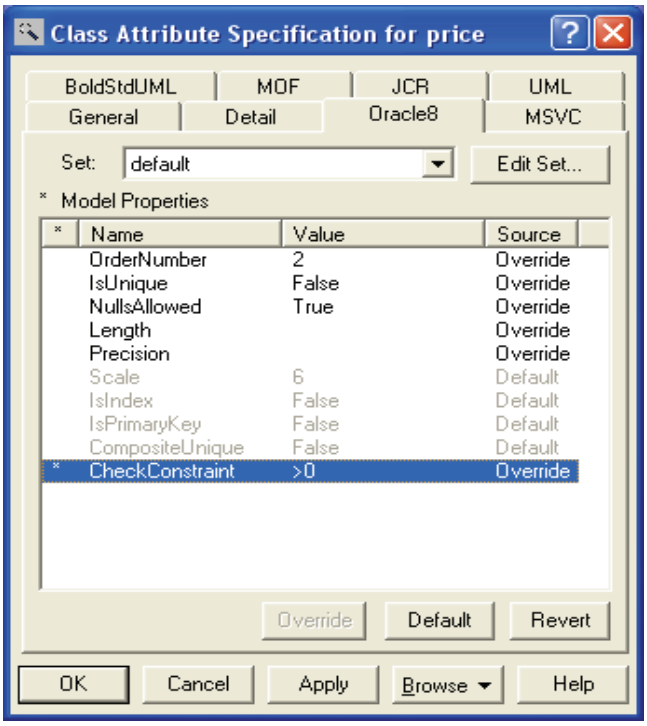


Figure 9.2. Properties of the *Price* attribute in Rational Rose

c) *MagicDraw* [61] offers a specific UML profile to define relational schemas which allows improving the code generation for that kind of databases. In this way, the user may annotate the class diagram with all the necessary information (*primary* and *foreign keys*, *unique* constraints and *checks* over attributes).

Figure 9.3 shows the (simplified) relational schema definition for the *Category* and *Customer* entity types, once annotated with the profile. The tool partially generates this relational schema from the initial CS. The schema includes the primary keys of each table and the foreign key from *Customer* to *Category* (relating the *cat* attribute of customer with the category's name). In the same way we may stereotype the attributes to include simple checks as the ones required in *CorrectProduct*. The other constraints cannot be specified since relational databases do not provide any predefined mechanism to check them (and *MagicDraw* neither generates itself any code fragment to do it).

Though *MagicDraw* allows the definition of OCL constraints, they are completely omitted in subsequent steps of the code generation process. For instance, when generating the relational schema corresponding to the initial conceptual schema, *MagicDraw* is unable to transform *CorrectProduct* in the corresponding *checks* in the relational schema. Even if we have first defined *CorrectProduct* in OCL, we are forced to manually define the constraint again in the relational schema.

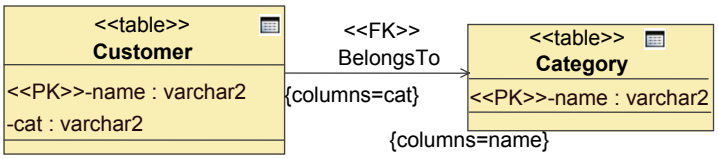


Figure 9.3. PSM for the relational schema in *MagicDraw*

d) *Objecteering/UML* [80] presents as a special feature with respect to the previous tools that supports (and generates the appropriate checking code) any multiplicity value in the relationship types. When generating the Java code it uses a predefined library to enforce the cardinality constraints. Moreover, it creates a set of triggers during the generation of the relational schema in the database. For instance, the trigger in Figure 9.4 checks that a category still contains more than three customers after the deletion of a customer. Otherwise, it stops the customer deletion process by raising an exception that rollbacks the transaction. The starting point for the database generation is, as in the previous tool, a schema annotated with a specific profile that can be semi-automatically obtained from the initial CS. It does not allow the definition of integrity constraints in OCL.

```
CREATE TRIGGER TI_cat_cust_DELET
 ON customer
 FOR DELETE AS
 IF NOT((SELECT COUNT(*)
    FROM customer, deleted
    WHERE employee.department = deleted.department) >=3)
 BEGIN
  ROLLBACK TRANSACTION
  RAISERROR 20501, "cust_cat : Deletion forbidden,
            cat_cust_FK minimum cardinality constraint violation"
 END
```

Figure 9.4. Trigger to control the minim number of customers per category

e) *Together* [12] offers similar capabilities to *Rational Rose* regarding the database generation. Moreover, it includes full OCL support to define constraints and pre/postconditions in the CS.

Nevertheless, when generating the Java code, only intra-entity constraints (see Section 1.3) are correctly generated. Moreover the generation is not efficient since constraints are verified after every single method of the class and not only after those methods possibly violating the constraints. As an example, see the Java class corresponding to the *Product* entity type in Figure 9.5. Even if we define the contract of the method *setName* (stating that the method just updates the *name* attribute) the generated class verifies that the value of the *price* attribute is correct after the method execution. The constraint is converted to a method (named *inv$0* in the Figure) returning a *true* boolean value if the constraint holds and *false* otherwise.

For the sake of clarity we have simplified a little bit the original code generated by *Together* as well as removed the code fragment in charge of verifying the pre and postcondition of the *setName* method.

```
public class Product {
  public int id;
  public String name;
  public int price;
  public int maxDiscount;
  public String description;

  void setName(String newName) {
     PrePost oclPreState = preSetName(this, newName);
     assert (PrePost.checkPost(oclPreState, null) &&  allInvariants(this));
  }

  static boolean inv$0(Product self) {
    java.lang.Boolean bool18 = java.lang.Boolean.valueOf(self.price > 0);
    return (bool18 != null ? (bool18).booleanValue() : false);
  }

  static boolean allInvariants(Product self) {return inv$0(self);}
}
```

Figure 9.5. *Product* class as generated by Together

Regarding inter-entity constraints, *Together* generates an uncompleted integrity checking code which does not suffices to detect constraint violations. For instance, when generating *ValidShipDate* constraint, the checking code will verify the constraint after all kinds of modifications over the attributes of *Sale* but, surprisingly, changes over shipment objects does not induce the verification of the constraint as well. This means that after updating the *paymentDate* of a Sale, the checking code would detect a constraint violation whereas after updates of the *paymentShipDate* of a *Shipment* no violation will ever be detected.

9.2.2 MDA Tools

Although, in fact, most of the tools evaluated in this whole section 9.2 are usually considered as MDA-tools we reserve this specific category to the tools closest to the MDA standard [69]. Therefore, we classify in this category tools having as their main goal to support the definition and execution of model transformations from PIMs to PSMs and from the PSMs to the final code. We evaluate in this section some of the most well-known MDA tools: *ArcStyler*, *OptimalJ* and *AndroMDA*.

ArcStyler [44] concentrates in the generation of Java, J2EE and .NET applications (with its *cartridge* architecture the designer can define additional transformations). When generating Java programs, the generated code is like the one in Poseidon, with the only difference that automatically creates a set of methods to modify the attributes representing

the associations of the original class diagram. *ArcStyler* also includes the *Dresden OCL* tool (see section 9.2.4) to define and generate the constraints.

OptimalJ [27] is devoted to generate J2EE applications where all the business logic concentrates in the Java classes (*Enterprise Java Beans* in this case). It only supports constraints over attributes using constant values or regular expressions. For more complex constraints, the designer must write the corresponding Java code directly.

AndroMDA [1] is an open source code generation framework that follows the MDA paradigm. According to the tool information, it takes model(s) from CASE-tool(s) and generates fully deployable applications. *AndroMDA* supports the definition of OCL query expressions and transforms them to the *Hibernate-QL* or *EJB-QL* query languages. However, no explicit support for OCL constraints is provided.

9.2.3 MDD Methods

In this section we group several well-known MDD methods although some of them may not follow the MDA approach nor use OMG standard languages.

OO-Method [35] is based on the formal language OASIS, although it admits the definition of UML class diagrams with constraints defined in an OCL-like language. Integrity constraints may include aggregate operators but type-level constraints are not allowed. Constraints are checked over the objects instance of the Java classes implementing the CS. Each time a method of a Java class is executed, all constraints defined on that class are verified (and not only the constraints that may be affected by that method execution). To check the constraints, they add a special method in each Java class (Figure 9.6). The method contains a set of conditions (one for each constraint defined on the class). When a condition is not satisfied, the method throws an exception.

```
Protected void checkIntegrityConstraints() throws Error
{
 if (! ((price<0) || (maxDiscount>60)))
 throw new error ("Constraint Violation. Invalid product");
}
```

Figure 9.6. Java method on the *Product* class verifying the *CorrectProduct* constraint

WebML [24] is specialized in the generation of web applications. It presents little support for defining integrity constraints. It only admits the definition of *validity predicates* on the web page forms. A validity predicate is a boolean expression that checks the correctness of the value entered by the user in a form included in a web page. The boolean expression may consist of boolean operators, arithmetic operators, comparisons (=,>,<,…) and constant values.

Executable UML [56] proposes to specify the behavior of an application in sufficient detail so that it can be directly executed. Specifications in executable UML consist merely of

class diagrams, state diagrams and action semantics to describe the operation behavior. Using a model compiler, then, the specification is internally transformed into Java or C++. It supports a predefined set of constraints like cardinality constraints, unique constraints or checks over the attribute's values. These constraints are afterwards expressed using the *Action Language* they provide. For more general ones, the designer must define them using this Action Language directly. That is, the designer is forced to define them in an imperative way and not declaratively (although action languages may contain query expressions they are basically an imperative language). Figure 9.7 shows the *NumberOfRestrictedProducts* constraint expressed in the action language. Tools following this approach (like *BridgePoint* [3] or *iUML* [20]) are mainly used in the real time and embedded domains.

```
select many restrictedproducts
from instances of RestrictedProduct
Return (cardinality restrictedproducts)<=20)
```

Figure 9.7. *NumberOfRestrictedProducts* defined with an Action Language

9.2.4 OCL Tools

This section evaluates all tools generating code from OCL constraints. Tools supporting OCL with other purposes (as model validation [40] or verification [4]) are not considered.

Dresden OCL [31] generates the Java classes corresponding to the entity types in the CS, including all constraints except for the type-level constraints, which are not supported. Integrity constraints are checked only after modifications over the attributes and associations (represented also as attributes in the Java classes) referenced in the constraint definition. This represents an efficiency improvement regarding previous methods, but, as shown in Chapter 4, this strategy is still inefficient since not all kinds of changes over the associations may violate a given constraint. For instance, they would determine that *AtLeastThreeCustomers* may also be violated when assigning a customer to a category. This is exactly the same limitation of [90].

OCLtoSQL is another tool comprised in the previous toolkit, based on the method proposed in [30]. It generates a relational schema implementing the CS. Additionally, for each constraint, it creates an SQL view. Similarly to the methods seen in section 9.1, 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. As an example, Figure 9.8 shows the view corresponding to the *CorrectProduct* constraint. Note that the view selects those products *not* verifying the *price* or the *maxDiscount* condition. The views are not efficient since they examine the whole table population instead of considering only those tuples modified during the transaction (in the example, the view accesses all products and not just the inserted or updated ones).

```
CREATE OR REPLACE VIEW CorrectProduct as
(select * from Product SELF where not (SELF.price>0) or not
(SELF.maxDiscount<=60);
```

Figure 9.8. View for the *CorectProduct* constraint

The code-generation capabilities of Octopus [48] are more limited. For each integrity constraint, it creates a new method in the Java class corresponding to the context type of the constraint. To know whether the constraint holds we must execute this method. If the constraint does not hold the method throws an exception. However, the decision about *when* the constraint needs to be verified (i.e. when we should call this method) is left to the designer, no hints are provided. *OCLE* [8] and *KMF* [47] provide a similar functionality.

OCL2J [32] generates a Java implementation of the CS including all intra-entity and inter-entity integrity constraints. As in *Together*, the constraint verification is inefficient since constraints are checked *before* and *after* executing any method of the class.

OCL4Java [93] forces the designer to explicitly link the integrity constraints with the methods that may violate them. Then, when generating the Java code for the methods, the constraints are added as preconditions and postconditions for the method (i.e. the constraint is verified at the beginning and at the end of the method execution).

BoldSoft [11] permits to execute an OCL expression over a set of objects stored in main memory or in the database (in this latter case, the expressivity is restricted, for instance, operators as *count*, *collect*, *difference*, *asSet*, *asBag* and so forth are not allowed). However, the tool is focused in the definition of derived elements and not in the integrity checking of constraints.

Finally, we would like to mention a couple of methods that, instead of generating the Java code required to verify the constraints, transform the body of each OCL constraint in terms of one of the constraint languages used in design by contract tools for Java, as *iContract* [49] (see [9] for the *OCL-iContract* translation) or JML [51] (see [43] for the *OCL-JML* translation). These tools allow annotating the Java classes with information about the invariants, pre and postconditions of the class. Then, a tool precompiler transforms these annotations in pure Java code. Nevertheless, the final code is inefficient as well since the precompiler transforms the class invariants by means of adding their verification to all (public) methods of the class, as done by the previous reviewed methods providing a direct implementation of the OCL constraints.

9.2.5 Summary information

The following table summarizes the comparison of the different tools. For each tool we indicate its expressivity and efficiency regarding the Java and relational database generation of the integrity constraints (ICs).

In the *expressivity* columns, the symbol *X* means that the tool does not support any kind of constraint definition while the symbol √ means a full constraint support and *n/a* indicates that the tool does not generate code for checking the defined constraints in that technology. Otherwise, we explicitly indicate the type of constraints admitted. Likewise for *efficiency* columns. In the *DB efficiency* column, cells are defined as *DBMS* when the tool relies on the constraint constructs offered by the database-management system (*primary keys*, *checks*…) to check the constraints.

Table 9.1. Tool comparison

Tools	Java		DB	
	Expressivity	**Efficiency**	**Expressivity**	**Efficiency**
Poseidon	X	n/a	PK	DBMS
Rational Rose	X	n/a	PK, intra	DBMS
Magic Draw	X	n/a	PK, intra	DBMS
Objecteering	cardinality	√	PK,cardinality	√
Together	√	ICs are verified after every method	PK, intra	DBMS
ArcStyler	Uses DresdenOCL	n/a	PK, intra	DBMS
OptimalJ	intra	√	PK, intra	DBMS
AndroMDA	X	n/a	PK, intra	DBMS
OO-Method	Intra, inter	ICs are verified after every method	PK	DBMS
WebML	intra	√	PK	DBMS
ExecutableUML	intra,predefined IC types	√	n/a	n/a
DresdenOCL	Intra, inter	ICs are verified after methods modifying the constrained elements	n/a	n/a
OCLtoSQL	n/a	n/a	√	Views evaluate all table population
Octopus	Intra, inter	n/a	n/a	n/a
OCLE	Intra, inter	n/a	n/a	n/a
KMF	Intra, inter	n/a	n/a	n/a
OCL2J	Intra, inter	ICs are verified before and after every method	n/a	n/a
OCL4Java	intra, inter	ICs must be manually linked to problematic methods	n/a	n/a

10. Conclusions and further research

10.1 Conclusions

The specification of a complete CS must include the definition of all relevant integrity constraints. Consequently, most CSs require the definition of a large number of constraints. These constraints must be taken into account when generating the implementation of the information system. This generation should be done automatically and yield an information system that efficiently (i.e. incrementally) checks all integrity constraints.

Current methods and tools offer little support for defining and generating constraints in CSs. Most tools only admit certain predefined constraint types. The few ones that allow full expressivity in the constraint definition language produce an inefficient implementation regarding the constraint integrity checking.

These limitations difficult the use of constraints in CSs. Designers are forced to either manually generate the (efficient) implementation of the constraints or search for an alternative way to represent the constraints in the CS, usually as postconditions in the contracts of system operations. A postcondition states a set of conditions that must be satisfied by the IB when an operation is completed. When constraints are not supported, designers must include in the postcondition of an operation *op* the verifications of all integrity constraints that could possibly be violated after executing *op*. This is a tedious and error-prone task.

This thesis has presented a fully automatic method for generating an efficient implementation for all kinds of integrity constraints in a CS. The generated implementation checks all constraints incrementally. By incremental we mean that the integrity checking process exploits available information about the applied structural events to consider as few entities of the IB as possible during the verification of the integrity constraints. We believe that our method is a new step towards the fulfillment of the goal of automating information systems, which is still a grand challenge for information systems research [65]. An implementation of our method is available at [16].

The main characteristic of our method is that it works at a conceptual level. As a result, it produces a standard CS, and thus, the method is technology-independent. Therefore, unlike earlier approaches, our results can be used regardless of the final technology platform chosen to implement the CS. In fact, any code-generation method or tool able to generate

code from a CS could be enhanced with our method to automatically generate incremental constraints with only minor adaptations.

We have divided our method into different steps to facilitate its integration and adoption. Some tools may prefer to incorporate just part of the developed techniques to get a partial efficiency improvement. This is especially important because processing a CS with our method implies creating new entity and relationship types and new constraints that must also be considered when implementing the system in the final technology platform. Although the number of new model elements is linearly proportional to the number of constraints in the original CS, this does increase the size and the complexity of the final implementation. Therefore, when a low population is expected at run-time for some entity and relationship types (meaning that there is not much difference between an incremental and a direct checking of the related constraints), it may be preferable to only partially process the constraints defined over those types (or not process them at all). [13] offers some more considerations on this topic.

Another important aspect of our method is that it handles integrity constraints at an event-grained level. For each PSE of a constraint *c*, the method provides a specialized version of *c* to get the maximum efficiency when verifying *c* after the application of that particular event. This implies that, even when a constraint cannot be verified incrementally after some of its PSEs, the method ensures an incremental verification for the rest of its PSEs.

The efficiency of the generated constraints is comparable to that of existing methods for relational and deductive databases. In this sense, we may regard our method as a leverage of those previous methods. Our method combines their efficiency with the technology-independence benefits of working at a higher-abstraction level.

The different techniques developed as part of the method presented in this thesis can be applied to solve similar problems in related areas. For instance, [7] partially adapts our method to incrementally verify the consistency of CSs and presents some experimental results to show the efficiency gain obtained when applying incremental techniques.

10.2 Further research

We would like to comment four possible lines of further research. The first two would extend our method to adapt other techniques from the database field as [76] and [52]. The adoption of these techniques at a conceptual level could improve the integrity checking of some constraints. The third line of research pretends to reuse the method developed in this thesis as a basis for solving the related problem of *materialization of derived types*. The fourth line would study in detail the applicability of our method to the problem of *model consistency checking*. A model (i.e. conceptual schema) is correct when verifies the *well-formedness rules* of the conceptual modeling language used to specify the schema. These

rules are usually expressed as constraints over the metamodel formalizing the modeling language. Each research line is sketched below in more detail.

10.2.1 Extending the CS with summary information.

[76] proposes a method to improve integrity checking of constraints containing aggregate operators in relational databases. This method adds several new attributes to the entity types of the CS to store certain information that can speed up the integrity checking.

As an example, given the constraint *AtLeastThreeCustomers* (*context Category inv: self.customer->size()>=3*; Figure 1.2), when we remove a customer from a category *cat* we need to count again all the customers belonging to *cat* to see if the constraint still holds. To avoid this recomputation, this method proposes adding a new attribute in the *Category* entity type to record the number of customers belonging to that category. Then, after removing a customer from a category *cat* we could verify the constraint simply by checking that *cat.numberOfCustomers – 1 >= 3* where *numberOfCustomers* is the name of the new attribute that records the number of customers in the category. Not all aggregation operators can benefit from this technique [73].

This technique improves integrity checking. However, in a way, it only moves the efficiency problem, since now the attribute *numberOfCustomers* must be efficiently kept up to date, which is not a trivial task (see Section 10.2.3). [76] also proposes a method to determine when this is worthwhile, but the decision is based on technologically dependent parameters (such as the number of page I/O operations). Nevertheless, the applicability of their ideas to the problem of integrity checking in CSs deserves further investigation.

10.2.2 Evaluating a pre-test before verifying a constraint

Our computation to determine when a constraint must be verified takes into account the type of the events applied during the modification of the IB. The method proposed in [52] refines the process by considering not only the type of the events but also the parameters of the applied events when deciding whether a given constraint must in fact be evaluated.

For each event this method proposes to evaluate a pre-test on the parameters of the event. If the pre-test is successful, then we need not check the constraint. If it fails nothing can be said about it and the usual integrity checking must be performed. Obviously, the pre-test is useful only as long as the cost of evaluating the pre-test is cheaper than the cost of evaluating the whole constraint. Pre-tests are obtained from the syntactic definition of the integrity constraint.

This method is unable to generate pre-tests for constraints including aggregate operators. However, pre-tests could be especially useful in such cases. For instance, given a constraint *context Sale inv: self.saleLine->select(quantity>10)-> size()<5* (stating that sales must not have five or more sale lines with more than 10 products each) a pre-test could serve to detect if an event *ev* of type *UpdateAttribute(quantity, SaleLine)* over a sale

line *sl* forces to check the constraint. In this case, the pre-test would consist in evaluating if the *quantity* attribute of *sl* satisfies the *select* condition. Only when the pre-test fails (i.e. when the new value of the *quantity* attribute of *sl* is greater than 10) the constraint must be checked. Otherwise, if the pre-test succeeds (i.e. the new value of the *quantity* attribute of *sl* is lower or equal to 10), the event does not increase the number of sales lines selected for the sale *s* in which *sl* is included (since *sl* does not satisfy the select condition), and thus, the constraint may not be violated.

10.2.3 Materialization of derived types

In general, CSs contain many derived entity and relationship types together with their corresponding derivation rules [64]. For efficiency reasons, some of these types may be *materialized*. When a derived type is materialized, its population is explicitly stored in the CS instead of being recomputed each time the type is queried. Then, changes in the population of base types referenced in its derivation rule may imply changes in the stored population of the materialized type. The propagation of these changes should be completely automatic. According to [65], this is still an open issue.

The problem of an incremental maintenance of materialized derived types is closely related to the problem of incremental integrity checking addressed in this thesis. Our problem can be regarded as a subset of this problem (in fact, incremental integrity checking is addressed in this manner in some of the proposals in the database field, see Chapter 9).

We believe that the techniques developed in this PhD Thesis can feasibly be extended to cover this more general problem as well.

10.2.4 Model consistency checking

Models (i.e. CSs) must be consistent with the *well-formedness* rules of the conceptual modeling language used to specify them. These rules restrict the possible combinations of the different model elements in the CS. For instance, all schemas specified in UML must be consistent with all well-formedness rules defined in the UML language. These rules include: “When an association specializes another association both must have the same number of participants”, “an association class cannot be defined between itself and something else”, “ a multiplicity must define at least one valid cardinality greater than zero” and so forth [68].

This problem has become more relevant due to the growing interest in tools for defining new modeling languages for particular domains [59], [5], [50] (known as Domain-Specific Languages) and the increasing number of large CSs.

Since well-formedness rules are usually defined as integrity constraints over the metamodel that formalizes the modeling language, our approach could be integrated with these tools to improve the efficiency of model consistency checking.

A first attempt in this direction has already been done [7], where the authors adopt steps 1 (determination of PSEs) and 3 (computation of relevant instances) of our method in the development of the SAP (meta)modeling infrastructure. We believe it is worth to study also the impact of step 2 and other parameters and techniques that may be specific of this particular problem (as the appropriate checking time or the minimum size of the models and/or the metamodels so that the application of incremental techniques is worthwhile).

References

1. AndroMDA 3.1. www.andromda.org/
2. ArgoUML v. 0.20. http://argouml.tigris.org/
3. Accelerated Technology. Nucleus BridgePoint Development Suite. http://www.acceleratedtechnology.com/embedded/nuc_bridgepoint.html
4. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P. H.: The KeY tool, Integrating object oriented design and formal verification. Software and Systems Modeling 4 (2005) 32-54
5. Alanen, M., Porres, I.: Coral: A Metamodel Kernel for Transformation Engines. In: Proc. 2nd European Workshop on Model Driven Architecture, (2004) 165-170
6. Ali, M. A., Fernandes, A., Paton, N. W.: MOVIE: An incremental maintenance system for materialized object views. Data & Knowledge Engineering 47 (2003) 131-166
7. Altenhofen, M., Hettel, T., Kusterer, S.: OCL support in an industrial environment. In: Proc. OCL for (Meta-)Models in Multiple Application Domains. Workshop of the MODELS'06 Int. Conf., (2006)
8. Babes-Bolyai. Object Constraint Language Environment 2.0. http://lci.cs.ubbcluj.ro/ocle/
9. Baresi, L., Young, M.: Toward Translating Design Constraints to Run-Time Assertions. Electronic Notes Theoretical Computer Science 116 (2005) 73-84
10. Blaha, M. R., Premerlani, W.: Object-Oriented Modeling and Design for Database Applications. Prentice Hall (1997)
11. Borland. Bold for Delphi. http://info.borland.com/techpubs/delphi/boldfordelphi/
12. Borland. Borland® Together® Architect 2006. http://www.borland.com/us/products/together/
13. Brambilla, M., Cabot, J.: Constraint tuning and management for Web applications. In: Proc. 6th. Int. Conf. on Web Engineering (ICWE'06), (2006) 345-353
14. Cabot, J., Teniente, E.: Determining the Structural Events that May Violate an Integrity Constraint. In: Proc. 7th Int. Conf. on the Unified Modeling Language (UML'04), LNCS, 3273 (2004) 173-187
15. Cabot, J., Teniente, E.: Computing the Relevant Instances that May Violate an OCL constraint. In: Proc. 17th Int. Conf. on Advanced Information Systems Engineering (CAiSE'05), LNCS, 3520 (2005) 48-62
16. Cabot, J., Teniente, E. A Tool for the Incremental Evaluation of OCL Constraints. Available: www.lsi.upc.edu/~jcabot/research/IncrementalOCL
17. Cabot, J., Teniente, E.: Transforming OCL Constraints: a context change approach. In: Proc. 2006 ACM symposium on Applied computing, (2006) 1196-1201

18. Cabot, J., Teniente, E.: Incremental Evaluation of OCL Constraints. In: Proc. 18th Int. Conf. on Advanced Information Systems Engineering (CAiSE'06), LNCS, 4001 (2006) 81-95
19. Cabot, J., Teniente, E.: Constraint Support in MDA tools: a Survey. In: Proc. 2nd European Conference on Model Driven Architecture, LNCS, 4066 (2006) 256-267
20. Carter, K. iUML 2.2. http://www.kc.com/products/iuml/index.html
21. Ceri, S., Widom, J.: Deriving Production Rules for Constraint Maintenance. In: Proc. 16th Int. Conf. on Very Large Databases (VLDB'90), (1990) 566-577
22. Ceri, S., Widom, J.: Deriving incremental production rules for deductive data. Information Systems 19 (1994) 467-490
23. Ceri, S., Fraternali, P., Paraboschi, S., Tanca, L.: Automatic generation of production rules for integrity maintenance. ACM Transactions on Database Systems 19 (1994) 367-422
24. Ceri, S., Fraternali, P., Bongio, A., Brambilla, M., Comai, S., Matera, M.: Designing Data-Intensive Web Applications. Morgan Kaufmann (2002)
25. Cervelló, C.: Ajuda a la comprovació eficient de restriccions d'integritat (in catalan). UPC-FIB Degree Thesis, (2006)
26. Christiansen, H., Martinenghi, D.: Simplification of database integrity constraints revisited: a transformational approach. In: Proc. 13th Int. Symposium on Logic Based Program Synthesis and Transformation, LNCS, 3018 (2003) 178-197
27. Compuware. OptimalJ. http://www.compuware.com/products/optimalj/
28. Correa, A., Werner, C.: Refactoring object constraint language specifications. Software and Systems Modeling 6 (2006)
29. Decker, H.: Translating Advanced Integrity Checking technology to SQL. In: J. Doorn and L. Rivero, (eds.): Idea Group (2002) 203-249
30. Demuth, B., Hussmann, H., Loecher, S.: OCL as a Specification Language for Business Rules in Database Applications. In: Proc. 4th Int. Conf. on the Unified Modeling Language (UML'01), LNCS, 2185 (2001) 104-117
31. Dresden. Dresden OCL Toolkit. http://dresden-ocl.sourceforge.net/index.html
32. Dzidek, W. J., Briand, L. C., Labiche, Y.: Lessons Learned from Developing a Dynamic OCL Constraint Enforcement Tool for Java. In: Proc. MODELS 2005 Workshops, LNCS, 3844 (2005) 10-19
33. Embley, D. W., Barry, D. K., Woodfield, S.: Object-Oriented Systems Analysis. A Model-Driven Approach. Yourdon Press Computing Series. Yourdon (1992)
34. EmPowerTec. OCL-AddIn for Rational Rose. http://www.empowertec.de/products/rational-rose-ocl.htm
35. Fons, J., Pelechano, V., Albert, M., Pastor, Ó. Development of Web Applications from Web Enhanced Conceptual Schemas. In: Proc. 22nd Int. Conf. on Conceptual Modeling (ER'03), LNCS, 2813 (2003) 232-245
36. Frias, L., Queralt, A., Olivé, A.: EU-Rent Car Rentals Specification. LSI Research Report, LSI-03-59-R (2003)
37. Gentleware. Poseidon for UML v. 4. http://www.gentleware.com
38. Giese, M., Larsson, D.: Simplifying Transformations of OCL Constraints. In: Proc. 8th Int. Conf. on Model Driven Engineering Languages and Sysmtes (MODELS'05), LNCS, 3713 (2005) 309-323
39. Gogolla, M., Richters, M.: Expressing UML Class Diagrams Properties with OCL. In: A. Clark and J. Warmer, (eds.): Object Modeling with the OCL. Springer-Verlag (2002) 85-114

40. Gogolla, M., Bohling, J., Richters, M.: Validation of UML and OCL Models by Automatic Snapshot Generation. In: Proc. 6th Int. Conf. on the Unified Modeling Language (UML'03). LNCS, 2863 (2003) 265-279
41. Gupta, A., Mumick, I. S.: Maintenance of materialized views: problems, techniques, and applications. In: Materialized Views Techniques, Implementations, and Applications. The MIT Press (1999) 145-157
42. Gupta, A., Mumick, I. S., Subrahmanian, V. S.: Maintaining views incrementally. In: Proc. 1993 ACM SIGMOD Int. Conf. on Management of Data, (1993) 157-166
43. Hamie, A.: Translating the Object Constraint Language into the Java Modelling Language. In: Proc. 2004 ACM symposium on Applied computing, (2004) 1531-1535
44. Interactive Objects. ArcStyler v.5. http://www.interactive-objects.com/
45. ISO/TC97/SC5/WG3: Concepts and Terminology for the Conceptual Schema and Information Base. ISO, (1982)
46. Java Community Process: The JavaTM Metadata Interface (JMI) Specification v.1.0. JSR-000040 (2002)
47. Kent Modelling Framework. Kent OCL Library. http://www.cs.kent.ac.uk/projects/kmf/
48. Klasse Objecten. Octopus: OCL Tool for Precise Uml Specifications. http://www.klasse.nl/octopus/index.html
49. Kramer, R.: iContract-the Java™ design by Contract tool. In: Proc. Technology of Object-Oriented Languages, TOOLS 26, (1998) 295-307
50. Lara, J. d., Vangheluwe, H.: AToM3: A Tool for Multi-Formalism Modelling and Meta-Modelling. In: Proc. 5th Int. Conf. on Fundamental Approaches to Software Engineering (FASE'02), LNCS, 2306 (2002) 174-188
51. Leavens, G. T., Baker, A. L., Ruby, C.: JML: A Notation for Detailed Design. In: H. Kilov, B. Rumpe, and I. Simmonds, (eds.): Behavioral Specifications of Businesses and Systems. Kluwer (1999.) 175-188
52. Lee, S. Y., Ling, T. W.: Further Improvements on Integrity Constraint Checking for Stratifiable Deductive Databases. In: Proc. 22th Int. Conf. on Very Large Data Bases (VLDB'96), (1996) 495-505
53. Link, S.: Consistency Enforcement in Databases. In: Proc. 2nd Int. Workshop on Semantics in Databases, Dagstuhl Seminar, LNCS, 2582 (2001) 139-159
54. Liu, J., Vincent, M., Mohania, M.: Maintaining views in object-relational databases. In: Proc. 9th Int. Conf. on Information and knowledge management (CIKM'00), (2000) 102-109
55. Mayol, E., Teniente, E.: A Survey of Current Methods for Integrity Constraint Maintenance and View Updating. In: Proc. Advances in Conceptual Modeling: ER '99 Workshops, LNCS, 1727 (1999) 62-73
56. Mellor, S. J., Balcer, M. J.: Executable UML. Object Technology Series. Addison-Wesley (2002)
57. Melton, J.: Advanced SQL: 1999 - Understanding Object-Relational and Other Advanced Features. Morgan Kaufmann (2002)
58. Melton, J., Simon, A. R.: SQL:1999, Understanding Relational Language Components. Morgan Kaufmann (2002)
59. MetaCase. MetaEdit+. http://www.metacase.com/mep/
60. NetBeans. Metadata Repository Project. http://mdr.netbeans.org/
61. NoMagic Inc. MagicDraw UML v. 10.5. http://www.magicdraw.com/

62. Nunamaker, J. F. j., Konsynski, B. j. R., Ho, T., Singer, C.: Computer-aided analysis and design of information systems. Communications of the ACM 19 (1976) 674-687
63. ObjectsbyDesign. List of UML tools. Available: http://www.objectsbydesign.com/
64. Olivé, A.: Derivation Rules in Object-Oriented Conceptual Modeling Languages. In: Proc. 15th Int. Conf. on Advanced Information Systems Engineering (CAiSE'03), LNCS, 2681 (2003) 404-420
65. Olivé, A.: Conceptual Schema-Centric Development: A Grand Challenge for Information Systems Research. In: Proc. 17th Int. Conf. on Advanced Information Systems Engineering (CAiSE'05), LNCS, 3520 (2005) 1-15
66. Olivé, A.: A method for the definition of integrity constraints in object-oriented conceptual modeling languages. Data & Knowledge Engineering, in press, available online November 2005 (2005)
67. OMG: UML 2.0 OCL Specification. OMG Adopted Specification (ptc/03-10-14) (2003)
68. OMG: UML 2.0 Superstructure Specification. OMG Adopted Specification (ptc/03-08-02) (2003)
69. OMG: MDA Guide Version 1.0.1. (2003)
70. OMG: XML Metadata Interchange (XMI) Specification v.2.0. OMG Adopted Specification (formal/03-05-02) (2003)
71. OMG: Unified Modeling Language Specification v.1.5. OMG Available Specification (formal/03-03-01) (2003)
72. OMG: MOF Core Specification. OMG Available Specification (formal/06-01-01) (2006)
73. Palpanas, T., Sidle, R., Cochrane, R., Pirahesh, H.: Incremental Maintenance for Non-Distributive Aggregate Functions. In: Proc. 28th Int. Conf. on Very Large Data Bases (VLDB'02), (2002) 802-813
74. Pastor, O., Gómez, J., Insfrán, E., Pelechano, V.: The OO-Method approach for information systems modeling: from object-oriented conceptual modeling to automated programming. Information Systems 26 (2001) 507-534
75. Rational. Rational Rose. http://www-306.ibm.com/software/rational/
76. Ross, K. A., Srivastava, D., Sudarshan, S.: Materialized view maintenance and integrity constraint checking: trading space for time. In: Proc. 1996 ACM SIGMOD Int. Conf. on Management of Data, (1996) 447-458
77. Schewe, K.-D., Thalheim, B.: Towards a theory of consistency enforcement. Acta Informatica 36 (1999) 97-141
78. Selic, B.: The pragmatics of model-driven development. IEEE Software 20 (2003) 19-25
79. Shlaer, S., Mellor, S.: Recursive design of an application-independent architecture. IEEE Software 14 (1997) 61-72
80. Softeam. Objecteering/UML v. 5.3. http://www.objecteering.com/products.php
81. Solana, R.: Generación de restricciones OCL alternativas (in spanish). UPC-FIB Degree Thesis, (2006)
82. Staudt, M., Jarke, M.: Incremental Maintenance of Externally Materialized Views. In: Proc. 22th Int. Conf. on Very Large Data Bases (VLDB'96), (1996) 75-86
83. Sze, E. K., Ling, T. W.: Efficient View Maintenance Using Version Numbers. In: Proc. 12th Int. Conf. on Database and Expert Systems Applications (DEXA'01), LNCS, 2113 (2001) 527-536

84. Teichroew, D.: Methodology for the Design of Information Processing Systems. In: Proc. 4th Australian Computer Conference, (1969) 629-634
85. Teichroew, D., Sayani, H.: Automation of System Building. Datamation 17 (1971) 25-30
86. Teniente, E., Urpí, T.: On the abductive or deductive nature of database schema validation and update processing problems. Theory and Practice of Logic Programming 3 (2003) 287-327
87. Türker, C., Gertz, M.: Semantic integrity support in SQL:1999 and commercial (object-)relational database management systems. The VLDB Journal 10 (2001) 241-269
88. Urpí, T., Olivé, A.: A Method for Change Computation in Deductive Databases. In: Proc. 18th Int. Conf. on Very Large Data Bases (VLDB'92), (1992) 225-237
89. Urpí, T., Olivé, A.: Semantic Change Computation Optimization in Active Databases. In: Proc. 4th Int. Workshop on Research Issues on Data Engineering-Active Database Systems (RIDE-ADS'94), (1994) 19-27
90. Verheecke, B., Straeten, R. V. D.: Specifying and implementing the operational use of constraints in object-oriented applications. In: Proc. Tools Pacific 2002, (2002) 23-32
91. Wiebicke, R.: Utility Support for Checking OCL Business Rules in Java Programs. Dresden University, Master Thesis, Department of Computer Science, (2000)
92. Wieringa, R.: A survey of structured and object-oriented software specification methods and techniques. ACM Computing Surveys 30 (1998) 459-527
93. Wolschon, M., Johner, C. OCL4Java. http://www.ocl4java.org
94. Zave, P.: The operational versus the conventional approach to software development. Communications of the ACM 27 (1984) 104-118

Appendix. Case study

As an additional example, in this appendix we show the complete application of our method over the CS of Figure A.1. This CS is an excerpt of the *EU-Rent Car Rentals Specification* [36], an in-depth specification of the EU-Rent case study, which is a widely known case study being promoted as a basis for demonstration of product capabilities. EU-Rent presents a car rental company with branches in several countries that provides typical rental services. EU-Rent was originally developed by Model Systems, Ltd.

The excerpt of Figure A.1 contains information about the rentals (*RentalAgreement* entity type), the company branches (*Branch* entity type), the rented cars (*Car*) and the people related with the company either as a renter (*Customer*), as a driver (*EU_RentPerson*) or even as a blacklisted person (*BlackListed*). For each rental, the CS includes information about the customer, the rental date, the initial and ending date of the rental, the car being rented and the branches where the car will be pick up and drop off. Rentals may be canceled (*CanceledReservation*) or closed (*ClosedRental*) when the rental has been successfully finished.

Additionally, [36] defines the constraints shown in Figure A.2 for this part of the EU-Rent system (yet in [36] they are specified as proposed in [66], we reexpress them as standard OCL invariants).

In sections A.1-A.4 we apply the different steps of our method to this case study. Then, in section A.5 we discuss the efficiency improvements of the integrity checking in the processed CS with respect to the efficiency of the integrity checking in the original CS of figure A.1.

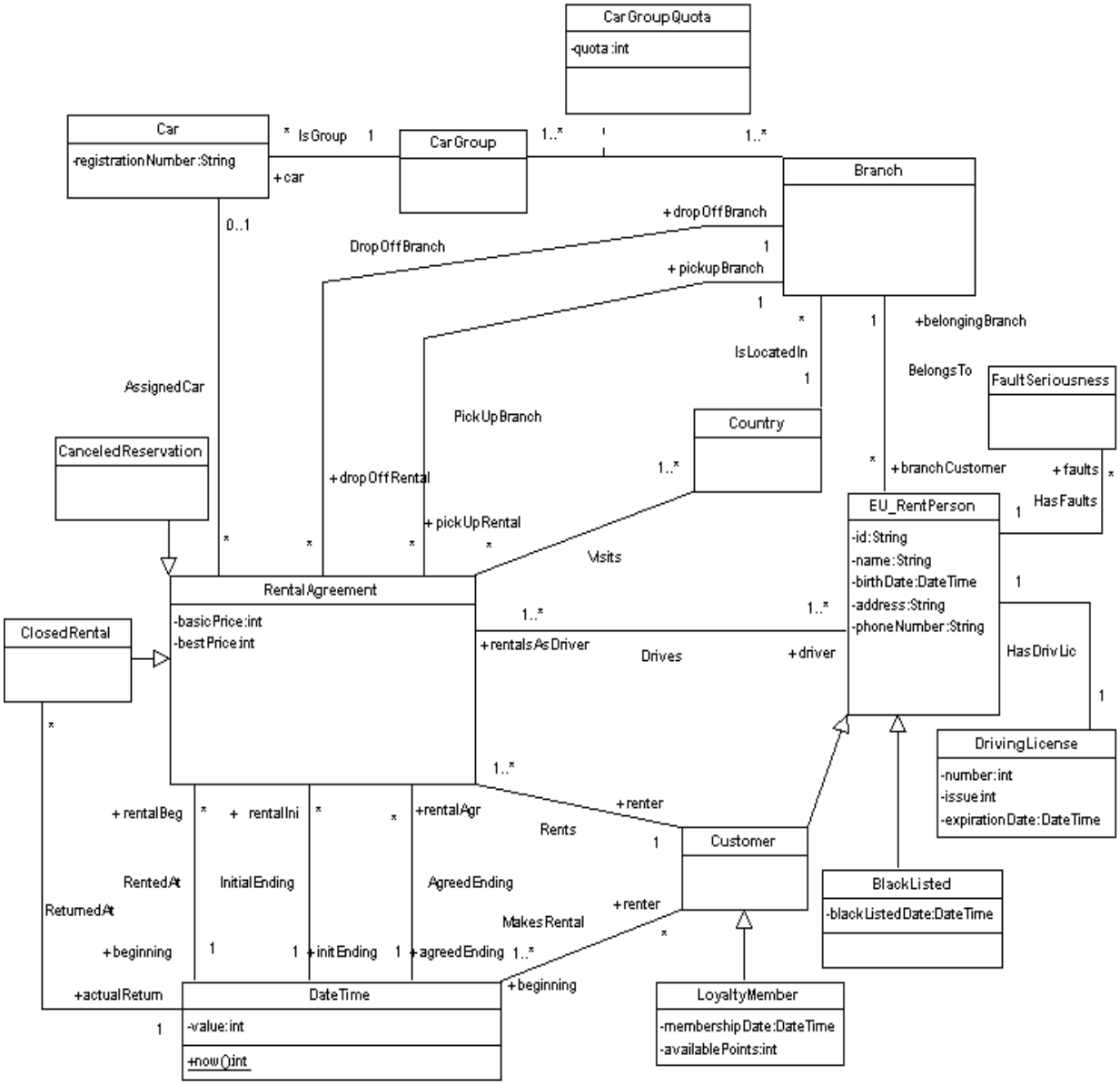


Figure A.1. EU-Rent CS

```
-- The dates for the rental are correct
context Closedrental inv CorrectInterval:
        self.beginning.value< self.initEnding.value and self.actualReturn.value>self.beginning.value

-- The pick up and drop off  countries appear in the list of visited  countries for the rental
context RentalAgreement inv VisitsBranchCountries:
        self.country->includes(self.pickupBranch.country) and self.country->includes(
        self.dropOffBranch.country)

-- An EU_RentPerson  must be over 25
context   EU_RentPerson inv Is25OrOlder: (DateTime::now()-self.birthDate) >=25*365

-- The id is an identifier for the EU_RentPerson type
context   EU_RentPerson inv IdIsKey:
        EU_RentPerson::allInstances()->select(p|p.id=self.id)->size()<2

-- No two rentals may exist for the same customer in an overlapping date interval
context Customer inv RentalsDoNotOverlap:
        not self.rentalAgreement-> reject(rA| rA.oclIsKindOf(CanceledReservation))->exists(rA |
        self.rentalAgreement->select(rAOther | rAOther.beginning.value> rA.beginning.value)-
        >exists(rAOther| rAOther.beginning.value <= rA.agreedEnding.value))

-- Loyalty members must have no faults and, at least, a rental done  during the last year
context LoyaltyMember inv MeetsLoyalPerformance:
        self.rentalAgreement.beginning->exists(dT| dT.value>(DateTime::now()-365)) and
        self.faults->isEmpty()

-- A car can only be assigned to a single active rental
context Car inv OnlyOneAssignment:
        self.rentalAgreement->select(rA |not(rA.oclIsTypeOf(CanceledReservation)) and
        not(rA.oclIsTypeOf(Closedrental)))->size()<=1

-- Cargroups must have a quota value defined for each branch
context   CarGroup inv QuotaForAllBranches:
        self.CarGroupQuota->size()=Branch::allInstances()->size()

-- BlackListed people cannot rent
context  BlackListed inv NoRentals:
        self.rentalsAsDriver->select(rA| rA.beginning.value > self.blackListedDate)->
        forAll(rA2|rA2.oclIsTypeOf(CanceledReservation))

-- Drivers need one-year of experience and a license with an expiration date beyond the rental date
context   DrivingLicense inv ValidLicense:
        (DateTime::now()-self.issue)> 365 and self.eu_RentPerson.rentalsAsDriver.agreedEnding->
        forAll(d|d.value<self.expirationDate)
```

Figure A.2. OCL constraints for the EU-Rent CS

A.1 Step 0: Simplification of the original constraints

Before applying our method we use the rules of Chapter 3 in order to simplify the body of the original constraints.

As a result, four of the previous ten constraints (*VisitsBranchCountries*, *RentalsDoNotOverlap*, *MeetsLoyalPerformance* and *NoRentals*), have been simplified. Figure A.3. shows the new body for these constraints.

VisitsBranchCountries has been simplified by means of the rule *X->includes(o) → X->count(o)>0* applied on both *includes* operators. For *RentalsDoNotOverlap*, we have applied the following list of rules (some of them several times): *not X->exists(Y) → X->forAll(not Y)*, *X->reject(Y) → X->select(not Y)*, *X->select(Y)->forAll(Z) → X->forAll(Y implies Z), X implies Y → not X or Y, not (not X) → X, not X>Y → X<=Y* and *not X<=Y → X>Y*. After these rules we obtain the following (intermediate) body: *self.rentalAgreement->forAll(rA:RentalAgreement | rA.oclIsKindOf(CanceledReservation) or self.rentalAgreement->forAll(rAOther: RentalAgreement | (rAOther.beginning.value <= rA.beginning.value) or (rAOther.beginning.value > rA.agreedEnding.value)))*. Over this body, we finally apply rule *X->forAll(v| Y [and|or] X->forAll(v2| Z)) → X->forAll(v,v2| Y [and|or] Z)* to obtain the final body shown in Figure A.3.

To simplify *MeetsLoyalPerformance* we just use *X->exists(Y) → X->select(Y)->size()>0* (to remove the *exists* iterator) and *X->isEmpty() → X->size()=0* (to remove the *isEmpty* operator). Finally, *NoRentals* is simplified by means of rule *X->select(Y)->forAll(Z) → X->forAll(Y implies Z)* (to remove the *select* iterator) and rules *X implies Y → not X or Y* and *not X>Y → X<=Y* (to remove the *implies* operator introduced by the first simplification rule).

```
context RentalAgreement inv VisitsBranchCountries:
        (self.country->count(self.pickupBranch.country)) > 0) and ((self.country->
        count(self.dropOffBranch.country)) > 0

context Customer inv RentalsDoNotOverlap:
        self.rentalAgreement-> forAll(rA, rAOther| rA.oclIsKindOf(CanceledReservation) or
         (rAOther.beginning.value>rA.beginning.value or rAOther.beginning.value <=
        rA.agreedEnding.value))

context LoyaltyMember inv MeetsLoyalPerformance:
         (self.rentalAgreement.beginning->select(dT: DateTime | dT.value > ((DateTime::now()) -
        365))->size()) > 0) and ((self.faults->size()) = 0

context  BlackListed inv NoRentals:
        self.rentalsAsDriver->forAll(rA2: RentalAgreement | (rA2.beginning.value <=
        self.blackListedDate) or rA2.oclIsTypeOf(CanceledReservation))
```

Figure A.3. Simplified constraints

A.2 Step 1: Determining the potentially-violating structural events

According to the part of the method defined in Chapter 4, the PSEs for the constraints of Figure A.2 (or Figure A.3 for the simplified ones) are the following:

- *CorrectInterval*. This constraint may be violated when "closing", i.e. specializing, an existing *RentalAgreement* instance since the new *ClosedRental* instance may not verify the constraint. Changes in the different dates involved in the comparison may also violate the constraint. Therefore, the list of PSEs is the following:
    - InsertRT(RentedAt)
    - InsertRT(InitialEnding)
    - UpdateAttribute(value, DateTime)
    - InsertRT(ReturnedAt)
    - SpecializeET(ClosedRental)
- *VisitsBranchCountries*. Removals of links from a rental to related *Branch* instances, probably as a previous step to change the pick up or drop off branch, may violate the constraint. A reduction on the number of countries visited during the rental may also violate it since the discarded country could be the one where the pick up or drop off the car was planned. List of PSEs:
    - DeleteRT(Visits)
    - DeleteRT(PickUpBranch)
    - InsertET(RentalAgreement)
    - DeleteRT(DropOffBranch)
    - DeleteRT(IsLocatedIn)
- *Is25OrOlder*. Only the insertion of a new person or changes on the *birthDate* attribute of an existing person may violate this constraint. List of PSEs:
    - InsertET(EU_RentPerson)
    - UpdateAttribute(birthDate, EU_RentPerson)
- *IdisKey*. Changes on the *id* attribute or insertions of a new person may violate this constraint. List of PSEs:
    - InsertET(EU_RentPerson)
    - UpdateAttribute(id, EU_RentPerson)
- *RentalsDoNotOverlap*. Apart from changes over the dates of existing rentals, a reactivation, i.e. generalization, of a closed or cancelled reservation may induce a constraint violation. List of PSEs:
    - InsertRT(RentedAt)
    - GeneralizeET(RentalAgreement)
    - UpdateAttribute(value,DateTime)
    - InsertRT(AgreedEnding)
    - InsertRT(Rents)
- *MeetsLoyalPerformance*. The relevant changes for this constraint are the removal of a rental from a *LoyaltyMember*, the transformation of a customer into a loyal member or the insertion of a new fault for a loyal member. List of PSEs:

  - DeleteRT(Rents)
  - DeleteRT(RentedAt)
  - UpdateAttribute(value,DateTime)
  - InsertRT(HasFaults)
  - InsertET(LoyaltyMember)
  - SpecializeET(LoyaltyMember)

- *OnlyOneAssignment*. The assignment of a new rental to a car or the reactivation of a closed or canceled rental may induce its violation. List of PSEs:
  - InsertRT(AssignedCar)
  - GeneralizeET(RentalAgreement)

- *QuotaForAllBranches*. Any change on the number of quotes for the group or on the number of existing branches may induce its violation. List of PSEs:
  - InsertRT(CarGroupQuota)
  - DeleteRT(CarGroupQuota)
  - InsertET(Branch)
  - DeleteET(Branch)
  - InsertET(CarGroup)

- *NoRentals*. Adding rentals to a blacklisted customer, transforming a customer into a blacklisted one or changing the dates of related rentals may violate the constraint; reactivating an existing rental may violate it as well. List of PSEs:
  - InsertRT(Drives)
  - InsertRT(RentedAt)
  - UpdateAttribute(blackListedDate, BlackListed)
  - GeneralizeET(RentalAgreement)
  - UpdateAttribute(value, DateTime)
  - SpecializeET(BlackListed)

- *ValidLicense*. A license may become invalid when changing its *issue* or *expiration date* or when relating the license with a new rental that may not satisfy the date condition stated in the constraint.
  - UpdateAttribute(issue, DrivingLicense)
  - InsertRT(HasDrivLic)
  - InsertRT(Drives)
  - InsertRT(AgreedEnding)
  - UpdateAttribute(value, DateTime)
  - UpdateAttribute(expirationDate, DrivingLicense)
  - InsertET(DrivingLicense)

A.3 Step 2: Determining an appropriate syntactic representation for each constraint

Once we know the PSEs for each constraint *c*, we can determine for each PSE *ev* of *c*, an appropriate representation of *c* regarding *ev*, as presented in Chapter 5.

Tables A.1-A.10 show the results of this step. In all tables, the first column contains the PSEs for the constraint, the second column the best context type to express the constraint with respect to that particular PSE and the third column the final constraint representation, which may be either the original one or a new generated alternative.

Table A.1. Alternative constraint representations for *CorrectInterval*. Thanks to the multiplicities of the relationships type between *RentalAgreement* and *DateTime*, we can use *ClosedRental* as best context for all PSEs except for the update of the value of a *DateTime* instance. Note that this PSE appears four times in the OCL tree representing the constraint (once for every access to the value attribute). All of them share *DateTime* as the best context. However, not all of them generate the same alternative representation since their *PathVar* expression differs.

PSE	**Context**	**Best alternative**
InsertRT(RentedAt)	ClosedRental	**context** Closedrental **inv** CorrectInterval: self.beginning.value< self.initEnding.value and self.actualReturn.value>self.beginning.value
SpecializeET(ClosedRental)	ClosedRental	
InsertRT(InitialEnding)	ClosedRental	**context** Closedrental **inv** $CorrectInterval_2$: self.beginning.value< self.initEnding.value
InsertRT(ReturnedAt)	ClosedRental	**context** Closedrental **inv** $CorrectInterval_3$: self.actualReturn.value>self.beginning.value
UpdateAttribute(value, DateTime)	DateTime	**context** DateTime **inv** $CorrectInterval_4$: self.rentalBeg->select(r| r.oclIsKindOf(ClosedRental)) -> forAll(r| r.beginning.value< r.initEnding.value and r.oclAsType(ClosedRental).actualReturn.value>r.beginning.value)
		context DateTime **inv** $CorrectInterval_5$: self.rentalIni->select(r| r.oclIsKindOf(ClosedRental)) -> forAll(r| r.beginning.value< r.initEnding.value)
		context DateTime **inv** $CorrectInterval_6$: self.closedRental-> forAll(r| r.actualReturn.value>r.beginning.value)

Table A.2. Alternative constraint representations for *VisitsBranchCountries*. All PSEs share the same context type. However, depending on the PSE, we avoid checking the whole constraint.

PSE	Context	Best alternative
DeleteRT(Visits)	RentalAgreement	**context** RentalAgreement **inv** VisitsBranchCountries: (self.country->count(self.pickupBranch.country)>0) and (self.country-> count(self.dropOffBranch.country)) > 0
DeleteRT(IsLocatedIn)	RentalAgreement	
InsertET (RentalAgreement)	RentalAgreement	
DeleteRT (DropOffBranch)	RentalAgreement	**context** RentalAgreement **inv** $VisitsBranchCountries_2$: (self.country-> count(self.dropOffBranch.country)) > 0
DeleteRT (PickUpBranch)	RentalAgreement	**context** RentalAgreement **inv** $VisitsBranchCountries_3$: (self.country->count(self.pickupBranch.country)>0)

Table A.3. Alternative constraint representations for *Is25OrOlder*.

PSE	Context	Best alternative
InsertET(EU_RentPerson)	EU_RentPerson	**context** EU_RentPerson **inv** Is25OrOlder: (DateTime::now()-self.birthDate) >=25*365
UpdateAttribute(birthDate, EU_RentPErson)	EU_RentPerson	

Table A.4. Alternative constraint representations for *IdISKey*

PSE	Context	Best alternative
InsertET(EU_RentPerson)	EU_RentPerson	**context** EU_RentPerson **inv** IdIsKey: EU_RentPerson::allInstances()->select(p|p.id=self.id) ->size()<2
UpdateAttribute(id, EU_RentPerson)	EU_RentPerson	

Table A.5. Alternative constraint representations for *RentalsDoNotOverlap*. Since this constraint states a condition that must be satisfied for every pair of rentals done by a customer, for most of its PSEs the best context is *RentalAgreement*.

PSE	Context	Best alternative
InsertRT(Rents)	RentalAgreement	**context** RentalAgreement **inv** RentalsDoNotOverlap: self.customer.rentalAgreement->forAll(rAOther| self.oclIsKindOf(CanceledReservation) or ((rAOther.beginning.value <= self.beginning.value) or (rAOther.beginning.value > self.agreedEnding.value)))
GeneralizeET (RentalAgreement)	RentalAgreement	
InsertRT (AgreedEnding)	RentalAgreement	
InsertRT(RentedAt)	RentalAgreement	
UpdateAttribute(value, DateTime)	DateTime	**context** DateTime **inv** RentalsDoNotOverlap$_2$: self.rentalBeg->forAll(r: RentalAgreement | r.customer.rentalAgreement->forAll(rAOther| rA.oclIsKindOf(CanceledReservation) or ((rAOther.beginning.value <= r.beginning.value) or (rAOther.beginning.value > r.agreedEnding.value))))
		context DateTime **inv** RentalsDoNotOverlap$_3$: self.rentalAgr->forAll(r: RentalAgreement | r.customer.rentalAgreement->forAll(rAOther| rA.oclIsKindOf(CanceledReservation) or ((rAOther.beginning.value <= r.beginning.value) or (rAOther.beginning.value > r.agreedEnding.value))))

Table A.6. Alternative constraint representations for *MeetsLoyalPerformance*. Both literals of the constraint are collection conditions, which makes *LoyaltyMember* the best context for all the PSEs. We distinguish between PSEs affecting one or both literals.

PSE	Context	Best alternative
InsertET (LoyaltyMember)	LoyaltyMember	**context** LoyaltyMember **inv** MeetsLoyalPerformance: (self.rentalAgreement.beginning->select(dT: DateTime | dT.value > ((DateTime::now()) - 365))->size()) > 0) and ((self.faults->size()) = 0
SpecializeET(Loyalty Member)	LoyaltyMember	
DeleteRT(Rents)	LoyaltyMember	**context** LoyaltyMember **inv** MeetsLoyalPerformance$_2$ self.rentalAgreement.beginning->select(dT: DateTime | dT.value > ((DateTime::now()) - 365))->size()) > 0
DeleteRT(RentedAt)	LoyaltyMember	
UpdateAttribute(value,DateTime)	LoyaltyMember	
InsertRT(HasFaults)	LoyaltyMember	**context** LoyaltyMember **inv** meetsLoyalPerformance$_3$ self.faults->size() = 0

Table A.7 Alternative constraint representations for *OnlyOneAssignment*.

PSE	Context	Best alternative
InsertRT(AssignedCar)	Car	**context** Car **inv** OnlyOneAssignment: self.rentalAgreement->select(rA |not(rA.oclIsTypeOf(CanceledReservation)) and not(rA.oclIsTypeOf(Closedrental)))->size()<=1
GeneralizeET (RentalAgreeement)	Car	

Table A.8. Alternative constraint representations for *QuotaForAllBranches*.

PSE	Context	Best alternative
InsertRT(CarGroupQuota)	CarGroup	**context** CarGroup **inv** quotaForAllBranches: self.CarGroupQuota->size()=Branch::allInstances()->size()
DeleteRT(CarGroupQuota)	CarGroup	
InsertET(CarGroup)	CarGroup	
InsertET(Branch)	CarGroup	
DeleteET(Branch)	CarGroup	

Table A.9. Alternative constraint representations for *NoRentals*. For PSEs modifying a blacklisted instance, we have to verify all his/her rentals. However, for PSEs inserting or modifying a rental it is more efficient to verify just that rental.

PSE	Context	Best alternative
SpecializeET (BlackListed)	BlackListed	**context** BlackListed **inv** NoRentals: self.rentalsAsDriver->forAll(rA2: RentalAgreement | (rA2.beginning.value <= self.blackListedDate) or rA2.oclIsTypeOf(CanceledReservation))
UpdateAttribute(blackListedDate)	BlackListed	
InsertRT(RentedAt)	RentalAgreement	**context** RentalAgreement **inv** $NoRentals_2$: self.driver->select(d| d.oclIsKindOf(BlackListed))-> forAll(d| self.beginning.value <= d.oclAsType(BlackListed).blackListedDate or self.oclIsTypeOf(CanceledReservation))
GeneralizeET (RentalAgreement)	RentalAgreement	
UpdateAttribute(value, DateTime)	DateTime	**context** DateTime **inv** $NoRentals_3$: self.rentalBeg->forAll(r| r.driver->select(d| d.oclIsKindOf(BlackListed))-> forAll(d| r.beginning.value<=d.oclAsType(BlackListed).blackListedDate or r.oclIsTypeOf(CanceledReservation)))
InsertRT(Drives)	Drives	**context** Drives **inv** $NoRentals_4$: (self.rentalsAsDriver.beginning.value <= self.driver->any(d| d.oclIsKindOf(BlackListed)). oclAsType(BlackListed).blackListedDate or (self.rentalsAsDriver.oclIsTypeOf(CanceledReservation))

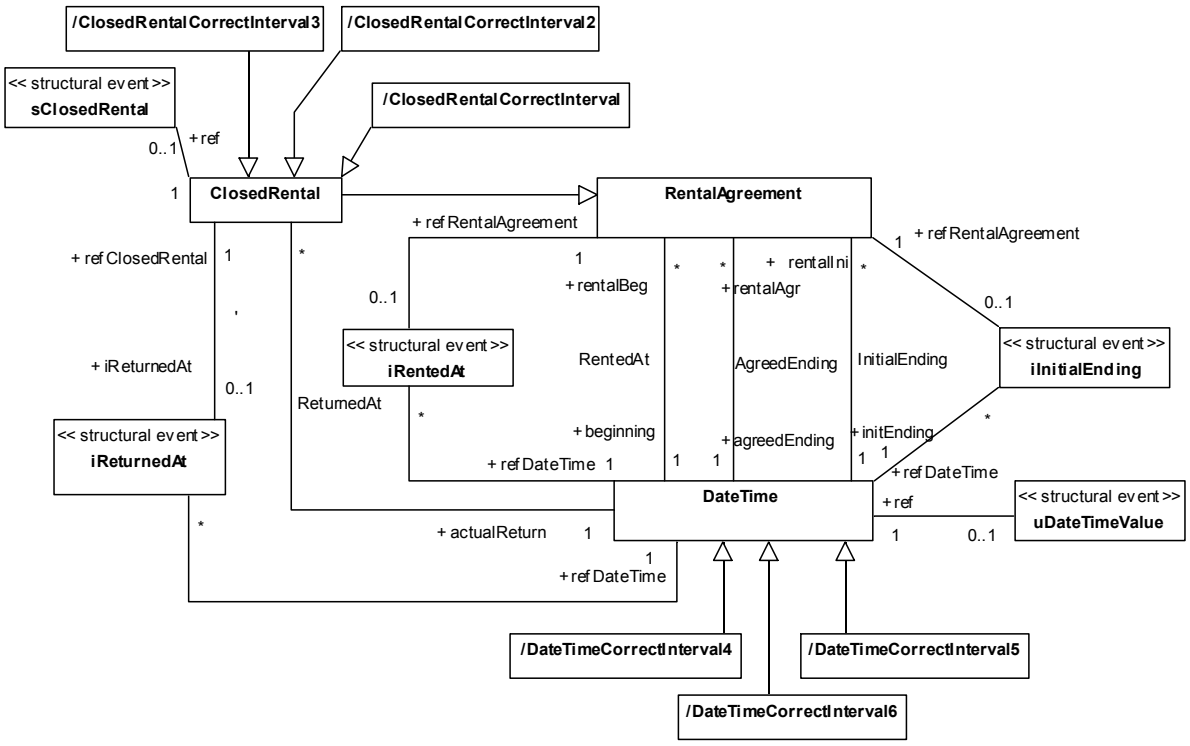
Table A.10. Alternative constraint representations for *ValidLicense*. When changing the driver license we must check all related rentals. Instead, when adding a new rental we just need to verify that rental. When updating an existing rental we must compare the rental with all its drivers.

PSE	Context	Best alternative
InsertET(DrivingLicense)	DrivingLicense	**context** DrivingLicense **inv** ValidLicense: (DateTime::now()-self.issue)> 365 and self.eu_RentPerson.rentalsAsDriver.agreedEnding-> forAll(d|d.value<self.expirationDate)
UpdateAttribute(issue,DrivingLicense)	DrivingLicense	**context** DrivingLicense **inv** $ValidLicense_2$: (DateTime::now()-self.issue)> 365
UpdateAttribute(expirationDate, DrivingLicense)	DrivingLicense	**context** DrivingLicense **inv** $ValidLicense_3$: self.eu_RentPerson.rentalsAsDriver.agreedEnding-> forAll(d|d.value<self.expirationDate)
InsertRT(HasDrivLic)	DrivingLicense	
InsertRT(Drives)	Drives	**context** Drives **inv** $ValidLicense_4$: self.rentalsAsDriver.agreedEnding.value<self.driver.drivingLicense.expirationDate
InsertRT(AgreedEnding)	RentalAgreement	**context** RentalAgreement **inv** $ValidLicense_5$: self.driver.drivingLicense->forAll(d| self.agreedEnding.value < d.expirationDate)
UpdateAttribute(value,DateTime)	DateTime	**context** DateTime **inv** $validLicense_6$: self.rentalAgr->forAll(r| r.driver.drivingLicense->forAll(d| r.agreedEnding.value<d.expirationDate))

A.4 Step 3: Redefining the constraints to evaluate the relevant instances

As a last step, our method modifies the CS to ensure that each alternative resulting from step 2 is only evaluated over the instances of its context type affected by events of one of the event types included in its particular subset of PSEs. As explained in Chapter 6, this implies extending the CS with a set of structural event types and derived subtypes, generating the appropriate derivation rules for the derived subtypes and redefining the constraints by means of using them as context types.

Figures A.4-A.13 show the processed schema for each group of constraints (i.e. for each set of alternatives generated from the same original constraint). In each figure, only the relevant part of the schema is shown.



-- The derivation rules

context ClosedRentalCorrectInterval::allInstances() : Set(ClosedRental) **body:** sClosedRental.allInstaces().ref->union(iRentedAt.allInstances().refRentalAgreement->select(r| r.oclIsKindOf(ClosedRental)))->asSet()

context ClosedRentalCorrectInterval$_2$::allInstances() : Set(ClosedRental) **body:** iInitialEnding.allInstances().refRentalAgreement->select(r|r.oclIsKindOf(ClosedRental))

context ClosedRentalCorrectInterval$_3$::allInstances() : Set(ClosedRental) **body:** iReturnedAt.allInstances().refClosedRental

context DateTimeCorrectInterval$_4$::allInstances() : Set(DateTime) **body:** uDateTimeValue.allInstances().ref

context DateTimeCorrectInterval$_5$::allInstances() : Set(DateTime) **body:** uDateTimeValue.allInstances().ref

context DateTimeCorrectInterval$_6$::allInstances() : Set(DateTime) **body:** uDateTimeValue.allInstances().ref

-- The redefined constraints

context ClosedRentalCorrectInterval **inv** CorrectInterval:
self.beginning.value< self.initEnding.value and self.actualReturn.value>self.beginning.value

context ClosedRentalCorrectInterval$_2$ **inv** CorrectInterval$_2$: self.beginning.value< self.initEnding.value

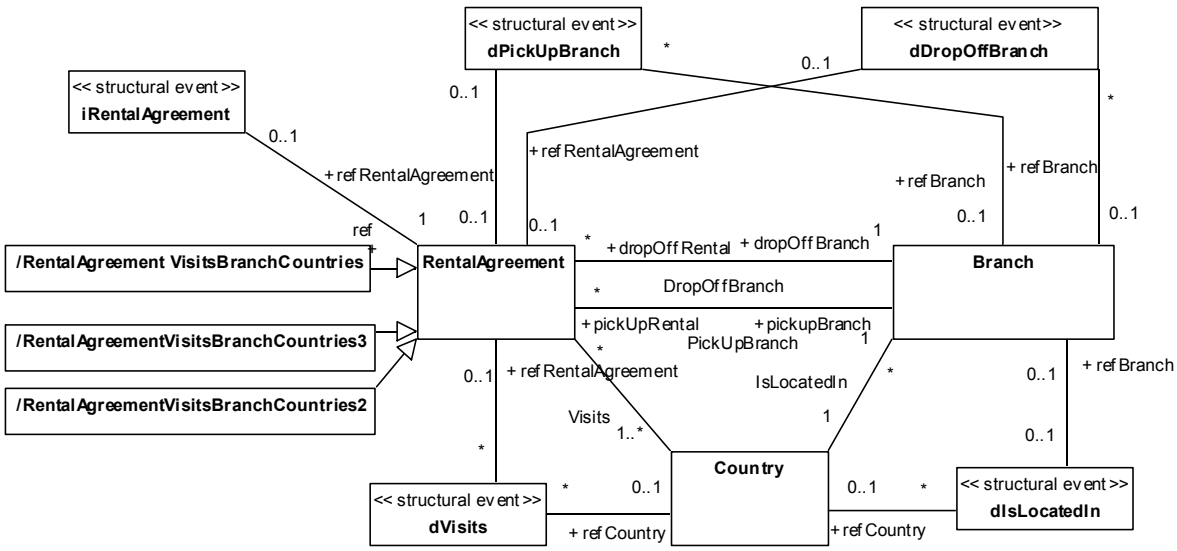
context ClosedRentalCorrectInterval$_3$ **inv** CorrectInterval$_3$: self.actualReturn.value>self.beginning.value

context DateTimeCorrectInterval$_4$ **inv** CorrectInterval$_4$: self.rentalBeg->select(r| r.oclIsKindOf(ClosedRental)) -> forAll(r| r.beginning.value< r.initEnding.value and r.oclAsType(ClosedRental).actualReturn.value>r.beginning.value)

context DateTime **inv** CorrectInterval$_5$: self.rentalIni->select(r| r.oclIsKindOf(ClosedRental)) -> forAll(r| r.beginning.value< r.initEnding.value)

context DateTime **inv** CorrectInterval$_6$: self.closedRental-> forAll(r| r.actualReturn.value>r.beginning.value)

Figure A.4. Schema modification for *CorrectInterval, CorrectInterval$_2$, CorrectInterval$_3$, CorrectInterval$_4$, CorrectInterval$_5$* and *CorrectInterval$_6$* integrity constraints. Note that the subexpression *select(r| r.oclIsKindOf(ClosedRental)* is added to the computation of the relevant instances for some of the events to ensure that only *ClosedRental* instances are taken into account.



-- The derivation rules

context RentalAgreementVisitsBranchCountries::allInstances() : Set(RentalAgreement) **body:**
dVisits.allInstances().refRentalAgreement->union(dIsLocatedIn.allInstances().refBranch.pickUpRental->union(dIsLocatedIn.allInstances().refBranch.dropOffRental->union(iRentalAgreement.allInstances().ref)))-> asSet()

context RentalAgreementVisitsBranchCountries$_2$::allInstances() : Set(RentalAgreement) **body:**
dDropOffBranch.allInstances().refRentalAgreement

context RentalAgreementVisitsBranchCountries$_3$::allInstances() : Set(RentalAgreement) **body:**
dPickUpBranch.allInstances().refRentalAgreement

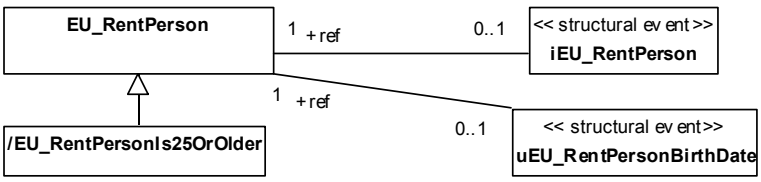
-- The redefined constraints

context RentalAgreementVisitsBranchCountries **inv** VisitsBranchCountries:
(self.country->count(self.pickupBranch.country)>0) and (self.country-> count(self.dropOffBranch.country)) > 0

context RentalAgreementVisitsBranchCountries$_2$ **inv** VisitsBranchCountries$_2$:
(self.country-> count(self.dropOffBranch.country)) > 0

context RentalAgreementVisitsBranchCountries$_3$ **inv** VisitsBranchCountries$_3$:
(self.country->count(self.pickupBranch.country)>0)

Figure A.5. Schema modification for *VisitsBranchCountries, VisitsBranchCountries$_2$,* and *VisitsBranchCountries$_3$* integrity constraints. Note that, after DeleteRT(IsLocatedIn) events, we must consider both the rentals related with the affected branch as *pickUpRentals* and as a *dropOffRentals*, since this PSE appears in both subexpressions of the constraint



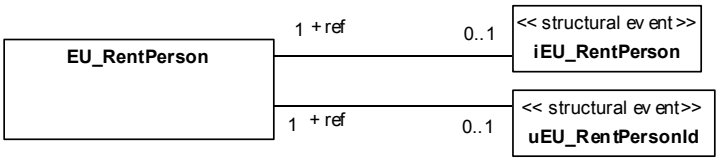
-- The derivation rules

context EU_RentPersonIs25OrOlder::allInstances() : Set(EU_RentPerson) **body:** iEU_RentPerson.allInstaces().ref->union(uEU_RentPersonBirthDate.allInstances().ref)->asSet()

-- The redefined constraints

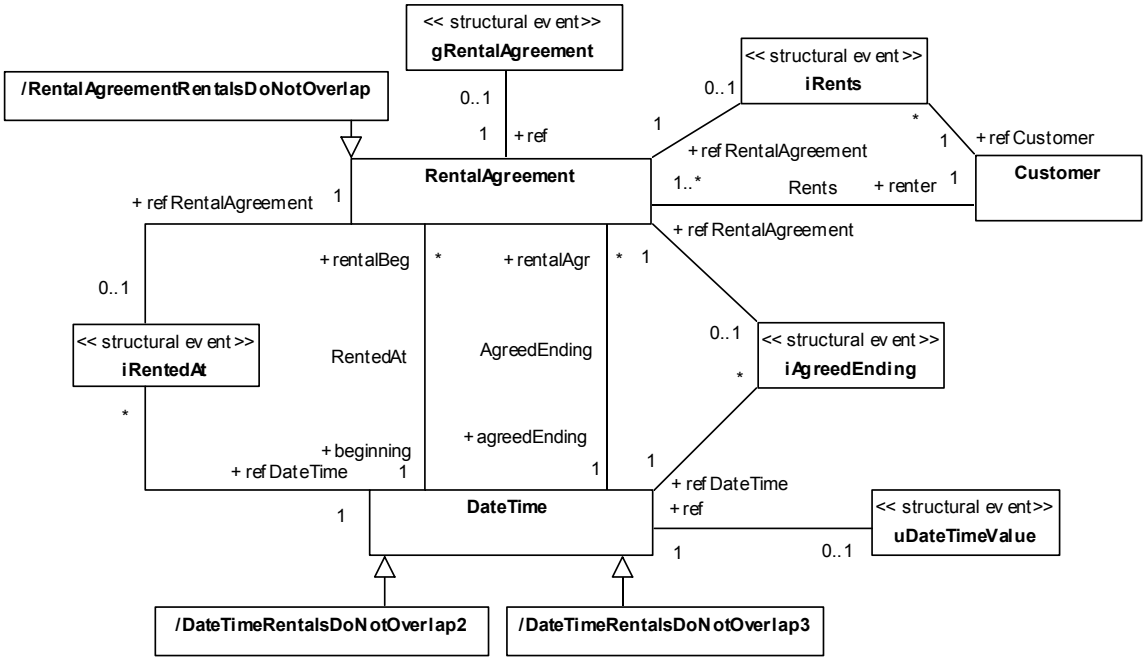
context EU RentPersonIs25OrOlder **inv** Is25OrOlder: (DateTime::now()-self.birthDate) >=25*365

Figure A.6. Schema modification for *Is25OrOlder* integrity constraint



-- The redefined constraints
context EU_RentPerson **inv** IdIsKey:
if iEU_RentPerson.allInstances->notEmpty() or uEU_RentPersonId.allInstances->notEmpty()
then EU_RentPerson::allInstances()->select(p|p.id=self.id) ->size()<2 endif

Figure A.7. Schema modification for *IdIsKey* integrity constraint. This constraint is not handled as partial instance constraint because all the PSEs attached to the subexpression beginning with the *self* variable also appear in the subexpression that starts with the *allInstances* operation



-- The derivation rules

context RentalAgreementRentalsDoNotOverlap::allInstances() : Set(RentalAgreement) **body:**
iRents.allInstances().refRentalAgreement->union(gRentalAgreement.allInstaces().ref-
>union(iAgreedEnding.allInstances().refRentalAgreement->union(iRentedAt.allInstances().refRentalAgreement)))->asSet()

context DateTimeRentalsDoNotOverlap$_2$::allInstances() : Set(RentalAgreement) **body:** uDateTimeValue.allInstances().ref

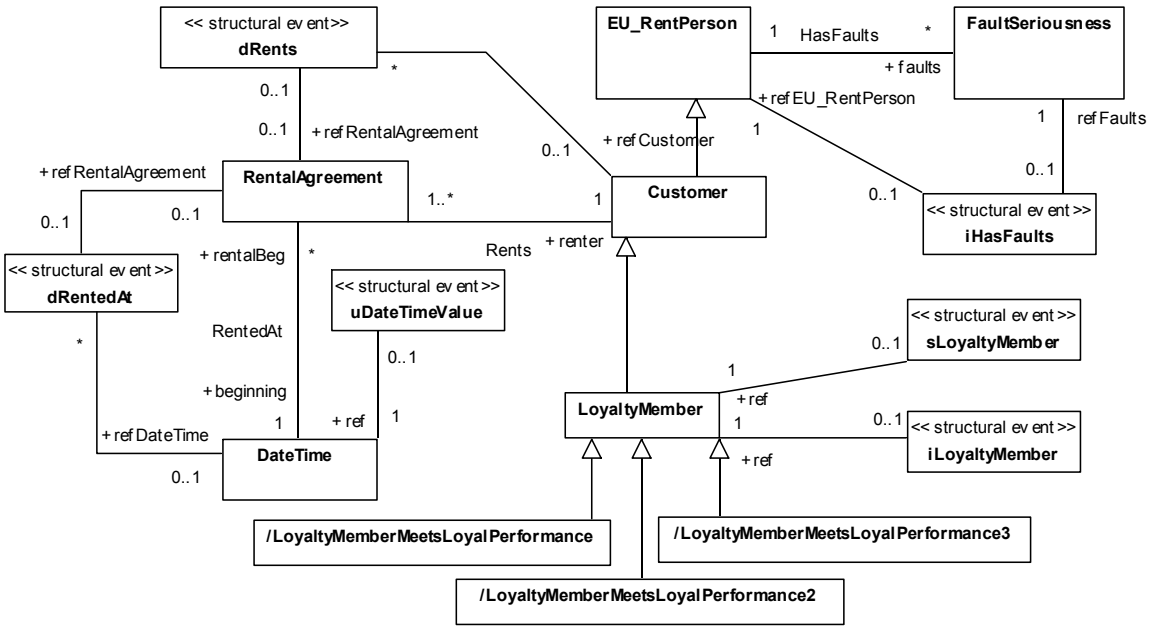
context DateTimeRentalsDoNotOverlap$_3$::allInstances() : Set(RentalAgreement) **body:** uDateTimeValue.allInstances().ref

-- The redefined constraints
context RentalAgreementRentalsDoNotOverlap **inv** RentalsDoNotOverlap:
self.customer.rentalAgreement->forAll(rAOther| self.oclIsKindOf(CanceledReservation) or ((rAOther.beginning.value <=
self.beginning.value) or (rAOther.beginning.value > self.agreedEnding.value)))

context DateTimeRentalsDoNotOverlap$_2$ **inv** RentalsDoNotOverlap$_2$:
self.rentalBeg->forAll(r: RentalAgreement | r.customer.rentalAgreement->forAll(rAOther| r.oclIsKindOf(CanceledReservation) or
((rAOther.beginning.value <= r.beginning.value) or (rAOther.beginning.value > r.agreedEnding.value))))

context DateTime **inv** RentalsDoNotOverlap$_3$:
self.rentalAgr->forAll(r: RentalAgreement | r.customer.rentalAgreement->forAll(rAOther| rA.oclIsKindOf(CanceledReservation)
or ((rAOther.beginning.value <= r.beginning.value) or (rAOther.beginning.value > r.agreedEnding.value))))

Figure A.8. Schema modification for *RentalsDoNotOverlap, RentalsDoNotOverlap$_2$,* and *RentalsDoNotOverlap$_3$* integrity constraints



-- The derivation rules

context LoyaltyMemberMeetsLoyalPerformance::allInstances() : Set(LoyaltyMember) **body:** iLoyaltyMember.allInstances().ref -> union (sLoyaltyMember.allInstaces().ref) ->asSet()

context LoyaltyMemberMeetsLoyalPerformance$_2$::allInstances() : Set(LoyaltyMember) **body:** dRents.allInstances().refCustomer->select(c| c.oclIsKindOf(LoyaltyMember))-> union (dRentedAt.allInstances().refRentalAgreement.renter->select(c| c.oclIsKindOf(LoyaltyMember))->union(uDateTimeValue.allInstances().ref.rentalBeg.renter->select(c| c.oclIsKindOf(LoyaltyMember))))->asSet()

context LoyaltyMemberMeetsLoyalPerformance$_3$::allInstances() : Set(LoyaltyMember) **body:** iHasFaults.allInstances().refEU_RentPerson->select(c|c.oclIsKindOf(LoyaltyMember))

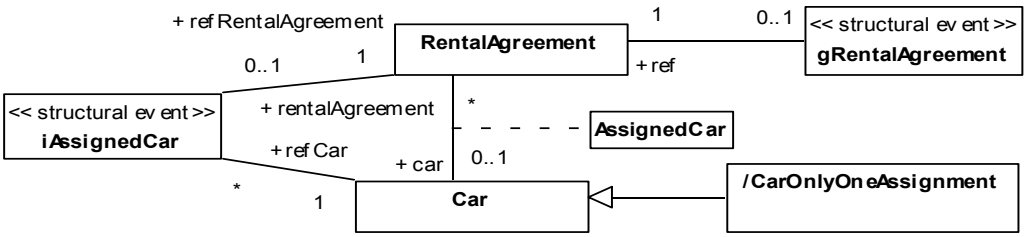
-- The redefined constraints

context LoyaltyMemberMeetsLoyalPerformance **inv** meetsLoyalPerformance: (self.rentalAgreement.beginning->select(dT: DateTime | dT.value > ((DateTime::now()) - 365))->size()) > 0) and ((self.faults->size()) = 0

context LoyaltyMemberMeetsLoyalPerformance$_2$ **inv** meetsLoyalPerformance$_2$: self.rentalAgreement.beginning->select(dT: DateTime | dT.value > ((DateTime::now()) - 365))->size()) > 0

context LoyaltyMemberMeetsLoyalPerformance$_3$ **inv** meetsLoyalPerformance$_3$: self.faults->size() = 0

Figure A.9. Schema modification for *MeetsLoyalPerformance, , MeetsLoyalPerformance$_2$* and *MeetsLoyalPerformance$_3$* integrity constraints



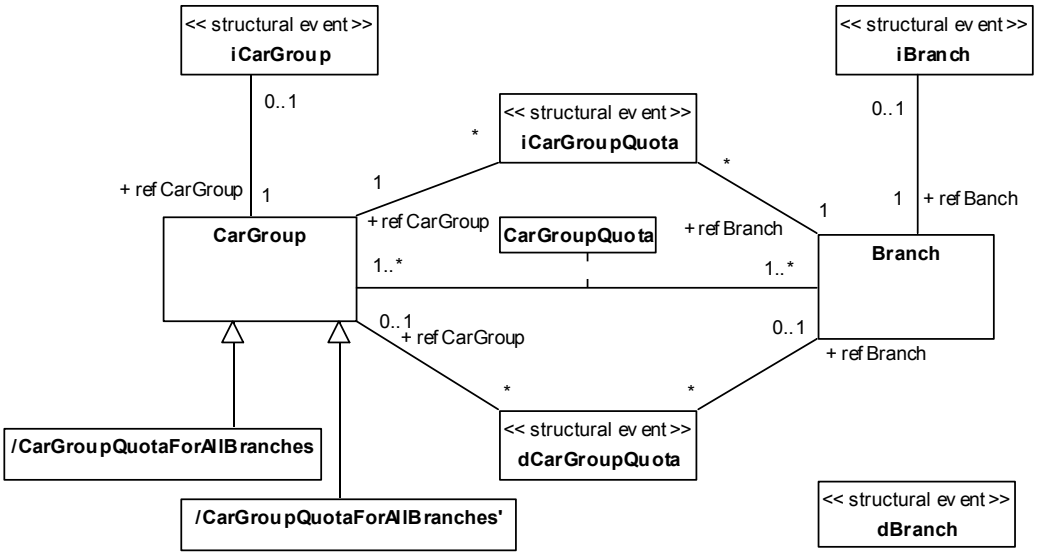
-- The derivation rule

context CarOnlyOneAssignment::allInstances() : Set(Car) **body:** iAssignedCar.allInstaces().refCar->union(gRentalAgreement.allInstances().ref.car)->asSet()

-- The redefined constraint

context CarOnlyOneAssignment **inv** onlyOneAssignment: self.rentalAgreement->select(rA |not(rA.oclIsTypeOf(CanceledReservation)) and not(rA.oclIsTypeOf(Closedrental)))->size()<=1

Figure A.10. Schema modification for *OnlyOneAssignment* integrity constraint



-- The derivation rules

context CarGroupQuotaForAllBranches::allInstances() : Set(CarGroup) **body:**

if CarGroupQuotaForAllBranches'.allInstances()->isEmpty() then iCarGroupQuota.allInstances().refCarGroup->union(dCarGroupQuota.allInstances().refCarGroup->union(iCarGroup.allInstances().ref))) ->asSet() endif

context CarGroupQuotaForAllBranches'::allInstances() : Set(CarGroup) **body:**

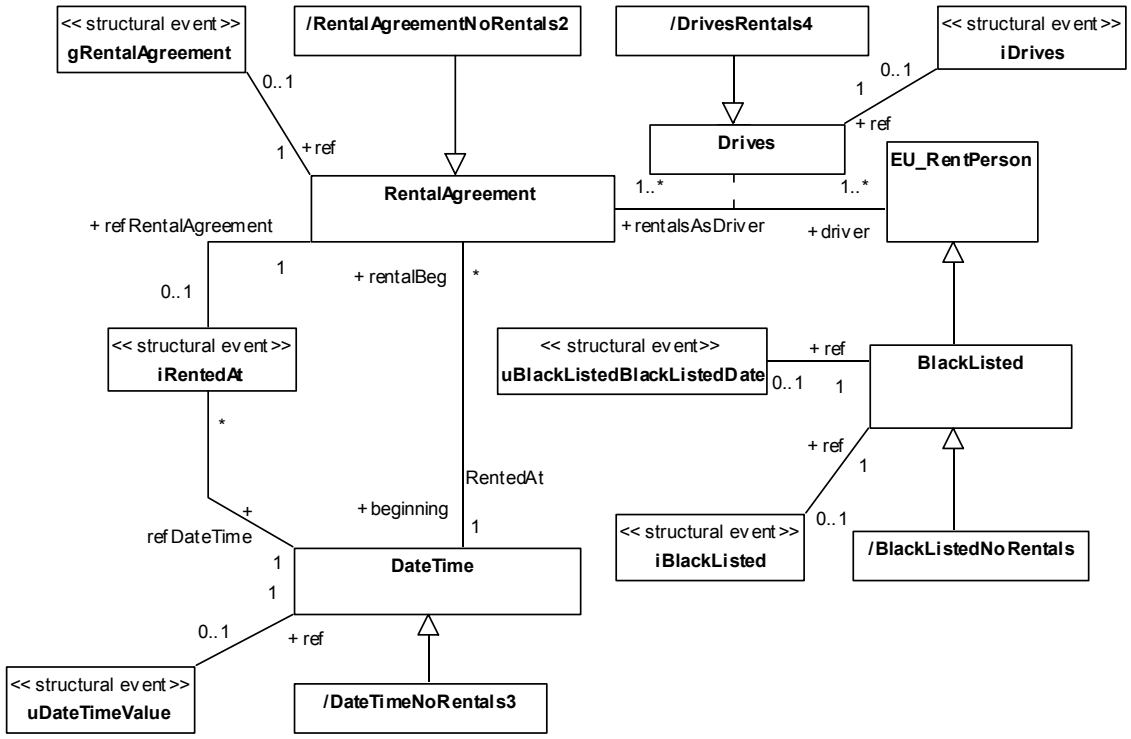
if (dBranch.allInstances()->union(iBranch.allInstances())->notEmpty() then CarGroup.allInstances() endif

-- The redefined constraints

context CarGroupQuotaForAllBranches **inv** quotaForAllBranches:
self.CarGroupQuota->size()=Branch::allInstances()->size()

context CarGroupQuotaForAllBranches' **inv** quotaForAllBranches':
self.CarGroupQuota->size()=Branch::allInstances()->size()

Figure A.11. Schema modification for *QuotaForAllBranches* integrity constraint. Even though we have a single alternative for this constraint we split it in two derived subtypes and two redefined constraints because it is a partial instance constraint (the PSEs *DeleteRT(Branch)* and *InsertET(Branch)* are class PSEs, and thus, after their issue we must check all existing *CarGroup* instances)



-- The derivation rules

context BlackListedNoRentals::allInstances() : Set(BlackListed) **body:** sBlackListed.allInstaces().ref->union(uBlackListedBlackListedDate.allInstances().ref)->asSet()

context RentalAgreementNoRentals$_2$::allInstances() : Set(RentalAgreement) **body:** iRentedAt.allInstances().refRentalAgreement->union(gRentalAgreement.allInstances().ref)->asSet()

context DateTimeNoRentals$_3$::allInstances() : Set(DateTime) **body:** uDateTimeValue.allInstances().ref

context DrivesNoRentals$_4$::allInstances() : Set(Drives) **body:** iDrives.allInstances().ref

-- The redefined constraints

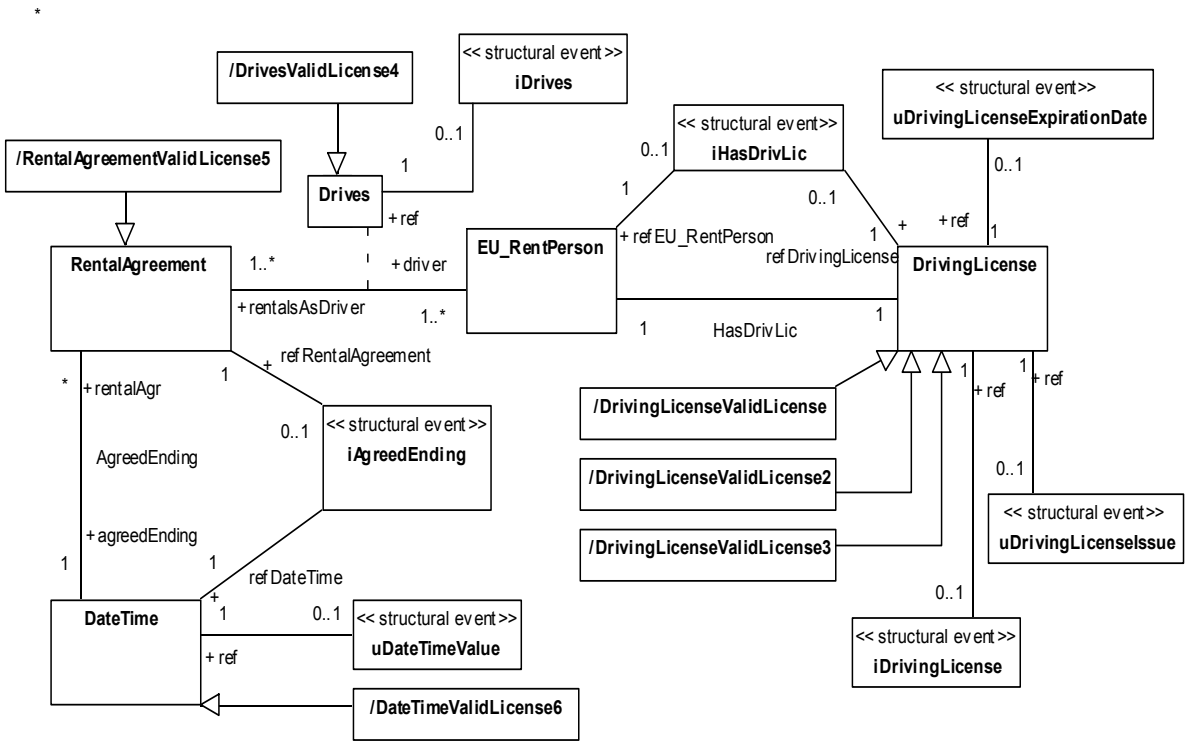
context BlackListed NoRentals **inv** NoRentals:
self.rentalsAsDriver->forAll(rA2: RentalAgreement | (rA2.beginning.value <= self.blackListedDate) or rA2.oclIsTypeOf(CanceledReservation))

context RentalAgreementNoRentals$_2$ **inv** NoRentals$_2$:
self.driver->select(d| d.oclIsKindOf(BlackListed))-> forAll(d| self.beginning.value <= d.oclAsType(BlackListed).blackListedDate or self.oclIsTypeOf(CanceledReservation))

context DateTimeNoRentals$_3$ **inv** NoRentals$_3$:
self.rentalBeg->forAll(r| r.driver->select(d| d.oclIsKindOf(BlackListed))-> forAll(d| r.beginning.value<=d.oclAsType(BlackListed).blackListedDate or r.oclIsTypeOf(CanceledReservation)))

context DrivesNoRentals$_4$ **inv** NoRentals$_4$:
(self.rentalsAsDriver.beginning.value <= self.driver->any(d| d.oclIsKindOf(BlackListed)).oclAsType(BlackListed).blackListedDate or (self.rentalsAsDriver.oclIsTypeOf(CanceledReservation))

Figure A.12. Schema modification for *NoRentals, NoRentals $_2$*, *NoRentals $_3$* and *NoRentals $_4$* integrity constraints. Note that the relationship type *Drives* has been reified since it is the context type of one of the alternative constraints. This implies that the *InsertRT(Drives)* event type is treated as an *InsertET(Drives)* event.



-- The derivation rules

context DrivingLicenseValidLicense::allInstances() : Set(DrivingLicense) **body:** iDrivingLicense.allInstaces().ref

context DrivingLicenseValidLicense$_2$::allInstances() : Set(DrivingLicense) **body:** uDrivingLicenseIssue.allInstances().ref

context DrivingLicenseValidLicense$_3$::allInstances() : Set(DrivingLicense) **body:** uDrivingLicenseExpirationDate.allInstances().ref->union(iHasDrivLic.allInstances().refDrivingLicense)->asSet()

context DrivesValidLicense $_4$::allInstances() : Set(Drives) **body:** iDrives.allInstances().ref

context RentalAgreementValidLicense $_5$::allInstances() : Set(RentalAgreement) **body:** iAgreedEnding.allInstances().refRentalAgreement

context DateTimeValidLicense $_6$::allInstances() : Set(DateTime) **body:** uDateTimeValue.allInstances().ref

-- The redefined constraints

context DrivingLicenseValidLicense **inv** ValidLicense: (DateTime::now()-self.issue)> 365 and self.eu_RentPerson.rentalsAsDriver.agreedEnding-> forAll(d|d.value<self.expirationDate)

context DrivingLicenseValidLicense$_2$ **inv** ValidLicense$_2$: (DateTime::now()-self.issue)> 365

context DrivingLicenseValidLicense$_3$ **inv** ValidLicense$_3$: self.eu_RentPerson.rentalsAsDriver.agreedEnding-> forAll(d|d.value<self.expirationDate)

context DrivesValidLicense$_4$ **inv** ValidLicense$_4$: self.rentalsAsDriver.agreedEnding.value<self.driver.drivingLicense.expirationDate

context RentalAgreementValidLicense$_5$ **inv** ValidLicense$_5$: self.driver.drivingLicense->forAll(d| self.agreedEnding.value<d.expirationDate)

context DateTimeValidLicense$_6$ **inv** ValidLicense$_6$: self.rentalAgr->forAll(r| r.driver.drivingLicense->forAll(d| r.agreedEnding.value<d.expirationDate))

Figure A.13. Schema modification for *ValidLicense, ValidLicense$_2$, ValidLicense$_3$, ValidLicense$_4$, ValidLicense$_5$* and *ValidLicense$_6$* integrity constraints.

A.5 Efficiency of the processed CS

Once we have completed the processing of the original CS we have obtained a new conceptual schema where all constraints have been redefined in order to get their incremental evaluation after arbitrary modifications of the IB.

Defining the cost of checking a constraint as the number of entities that must be taken into account during its evaluation, Tables A.11-A.20 compare, for each one of the original constraints, the cost of a direct checking of the constraint with the cost obtained when checking the new version. Although not explicited in the tables, an additional efficiency gain of the processed schema is that when the modification of the IB does not include any of the PSEs for a constraint *c*, *c* is not verified (i.e. the cost is zero). This is not restricted in the original schema.

In all tables, the column *PSE* shows the PSEs of the original constraint. Column *Incr Constraint* refers to the name of the specialized constraint generated for that PSE in the processed CS (note that even if the name of the new constraint coincides with that of the original one, in the processed schema the new constraint has been redefined to be evaluated over the relevant instances, and thus, their cost may be different). Column *cost* refers to the cost of evaluating the processed constraints after the issue of an event of the event type appearing in the first column. The cost of evaluating the original constraint is always the same regardless the PSE applied over the IB.

In the cells, P_x stands for the population of the type *X* (for instance, $P_{country}$ represents the number of instances of *Country*). $N_{x\text{-}y}$ stands for the (average) number of entities of *X* related with an entity of *Y* (for instance, $N_{rental\text{-}country}$ represents the average number of country visited by a rental agreement). *Y* may be either the name of a role or the name of the destination type (when no ambiguity exists). We usually abbreviate the names of the types in the formulas by using the first two letters of the type name.

Table A.11. Constraint *CorrectInterval*. $Cost_{CorrectInterval} = P_{Cl} + P_{Cl} \times 3$ (i.e. we access the whole population of *ClosedRental* plus the related dates of each rental instance). Note that when updating a date, we need to add the cost of the three generated alternatives affected by this event.

PSE	Incr Constraint	Cost
InsertRT(RentedAt)	CorrectInterval	4
SpecializeET(ClosedRental)	CorrectInterval	4
InsertRT(InitialEnding)	$CorrectInterval_2$	3
InsertRT(ReturnedAt)	$CorrectInterval_3$	3
UpdateAttribute(value, DateTime)	$CorrectInterval_4$, $CorrectInterval_5$, $CorrectInterval_6$	$(1+N_{da\text{-}rentalBeg} + N_{da\text{-}rentalBeg} \times 3) + (1+N_{da\text{-}rentalIni} + N_{da\text{-}rentalIni} \times 2) + (1+N_{da\text{-}Cl} + N_{da\text{-}Cl} \times 2)$

Table A.12. Constraint *VisitsBranchCountires*. $Cost_{VisitsBranchCountries} = P_{Re} + P_{Re}$ x $(N_{ra\text{-}co} + 4)$ (i.e. we access the whole population of *RentalAgreement* plus, for each rental instance, the visited countries, the related pickup and dropoff branches and the countries where the branches are located in).

PSE	Incr Constraint	Cost
DeleteRT(Visits)	VisitsBranchCountries	$1+ (N_{ra\text{-}co} + 4)$
DeleteRT(IsLocatedIn)	VisitsBranchCountries	$1+ (N_{Br\text{-}pi}+ N_{Br\text{-}dr}) + (N_{Br\text{-}pi}+ N_{Br\text{-}dr})$ x $(N_{ra\text{-}co} +4)$
InsertET(RentalAgreement)	VisitsBranchCountries	$1+ (N_{ra\text{-}co} + 4)$
DeleteRT(DropOffBranch)	$VisitsBranchCountries_2$	$1+ (N_{ra\text{-}co} +2)$
DeleteRT(PickUpBranch)	$VisitsBranchCountries_3$	$1+ (N_{ra\text{-}co} + 2)$

Table A.13. Constraint *Is25OrOlder*. $Cost_{Is25OrOlder} = P_{EU}$.

PSE	Incr Constraint	Cost
InsertET(EU_RentPerson)	Is25OrOlder	1
UpdateAttribute(birthDate)		

Table A.14. Constraint *IdIsKey*. $Cost_{IdIsKey} = P_{EU}xP_{EU}$ (we compare each person with all other existing people).

PSE	Incr Constraint	Cost
InsertET(EU_RentPerson)	IsIsKey	$P_{EU}xP_{EU}$
UpdateAttribute(id)		

Table A.15. Constraint *RentalsDoNotOverlap*. $Cost_{RentalsDoNotOverlap} = P_{Cu} + P_{Cu}$ x $(N_{cu\text{-}ra}\text{^}2 + N_{cu\text{-}ra}\text{^}2$ x 3) (i.e. we access the whole population of *Customer* plus, for each rental combination, the related dates).

PSE	Incr Constraint	Cost
InsertRT(Rents)	RentalsDoNotOverlap	$4+ 2xN_{cu\text{-}ra}$
GeneralizeET(RentalAgreement)		
InsertRT(AgreedEnding)		
InsertRT(RentedAt)		
UpdateAttribute(value,DateTime)	$RentalsDoNotOverlap_2$, $RentalsDoNotOverlap_3$	$1+ N_{da\text{-}rentalBeg}+ N_{da\text{-}rentalBeg}$ x $(3+2xN_{cu\text{-}ra}) + N_{da\text{-}rentalAgr}+ N_{da\text{-}rentalAgr}$ x $(3+2xN_{cu\text{-}ra})$

Table A.16. Constraint *MeetsLoyalPerformance*. $Cost_{MeetsLoyalPerformance} = P_{Lo} + P_{Lo} \times (N_{lo\text{-}ra} + N_{lo\text{-}ra} \times 1 + N_{lo\text{-}fa})$ (i.e. we access the whole population of *LoyaltyMember* plus, for each loyal member, his/her rental agreements and their dates, and the related faults).

PSE	Incr Constraint	Cost
InsertET(LoyaltyMember)	MeetsLoyalPerformance	$1 + 2xN_{lo\text{-}ra} + N_{lo\text{-}fa}$
SpecializeET(LoyaltyMember)		
DeleteRT(Rents)	$MeetsLoyalPerformance_2$	$1 + 2x\ N_{lo\text{-}ra}$
DeleteRT(RentedAt)	$MeetsLoyalPerformance_2$	$1 + N_{ra\text{-}cu} + N_{ra\text{-}cu} \times (2xN_{lo\text{-}ra})$
UpdateAttribute(value,DateTime)	$MeetsLoyalPerformance_2$	$1 + N_{da\text{-}rentalBeg} + N_{da\text{-}rentalBeg} \times N_{ra\text{-}cu} + N_{da\text{-}rentalBeg} \times N_{ra\text{-}cu} \times (2xN_{lo\text{-}ra})$
InsertRT(HasFaults)	$MeetsLoyalPerformance_3$	$1 + N_{lo\text{-}fa}$

Table A.17. Constraint *OnlyOneAssignment*. $Cost_{OnlyOneAssignment} = P_{Ca} + P_{Ca} \times N_{ca\text{-}ra}$ (i.e. we access the whole population of *Car* plus, for each car, its rental agreements).

PSE	Incr Constraint	Cost
InsertRT(AssignedCar)	OnlyOneAssignment	$1 + N_{ca\text{-}ra}$
GeneralizeET(RentalAgreeement)		

Table A.18. Constraint *QuotaForAllBranches*. $Cost_{QuotaForAllBranches} = P_{Ca} + P_{Ca} \times (N_{ca\text{-}carGroupQuota} + P_{Br})$ (i.e. we access the whole population of *CarGroup* plus, for each car group, its quotes and all existing branches, as required by the *allInstances* operation).

PSE	Incr Constraint	Cost
InsertRT(CarGroupQuota)	QuotaForAllBranches	$1 + N_{ca\text{-}carGroupQuota} + P_{Br}$
DeleteRT(CarGroupQuota)		
InsertET(CarGroup)		
InsertET(Branch)	QuotaForAllBranches	$P_{Ca} + P_{Ca} \times (N_{ca\text{-}carGroupQuota} + P_{Br})$
DeleteET(Branch)		

Table A.19. Constraint *NoRentals*. $Cost_{NoRentals} = P_{Bl} + P_{Bl} \times (N_{Bl\text{-}ra} + N_{Bl\text{-}ra} x1)$ (i.e. we access the whole population of *BlackListed* plus, for each black listed person, his/her rental agreements and their dates).

PSE	Incr Constraint	Cost
SpecializeET(BlackListed)	NoRentals	$1 + 2xN_{Bl\text{-}ra}$
UpdateAttribute(BlackListedDate)		
InsertRT(RentedAt)	$NoRentals_2$	$2 + N_{ra}.N_{eu}$
GeneralizeET(RentalAgreement)		
UpdateAttribute(value,DateTime)	$NoRentals_3$	$1 + 2xN_{da\text{-}ra} + N_{da\text{-}ra} \times N_{ra}.N_{eu}$
InsertRT(Drives)	$NoRentals_4$	4

Table A.20. Constraint *ValidLicense*. $Cost_{ValidLicense} = P_{Dr} + P_{Dr} \times (1 + N_{eu—ra} + N_{eu—ra} x1)$ (i.e. we access the whole population of *DrivingLicense* plus, for each license, the corresponding eu_rentPerson as well as his/her rental agreements and their dates).

PSE	Incr Constraint	Cost
InsertET(DrivingLicense)	ValidLicense	$2 + 2xN_{eu\text{-}ra}$
UpdateAttribute(issue)	$ValidLicense_2$	1
UpdateAttribute(expirationDate)	$ValidLicense_3$	$2 + 2x\ N_{eu\text{--}ra}$
InsertRT(HasDrivLic)		
InsertRT(Drives)	$ValidLicense_4$	5
InsertRT(AgreedEnding)	$ValidLicense_5$	$2 + 2xN_{ra\text{-}eu}$
UpdateAttribute(value,DateTime)	$ValidLicense_6$	$1 + 2xN_{da\text{-}rentalAgr} + 2xN_{da\text{-}rentalAgr} \times N_{ra\text{-}eu}$