JORDI CABOT SAGRERA

INTEGRATION OF EFFICIENT INTEGRITY CHECKING TECHNIQUES IN CODE-GENERATION TOOLS FOR WEB APPLICATIONS

BARCELONA

2007

A report presented by Jordi Cabot Sagrera in partial fulfillment of the requirements for the travel grant BE 2006 (id 00062) of the Generalitat de Catalunya

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 [37]. Recently, a number of new alternatives ([24], [35], [12], among many others) and standards [32] 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 [27]. [27] 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 [27] 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 [22]. 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. The development of such a method was the main goal of my PhD Thesis [8]. The method presented therein provides an automatic and incremental evaluation of all integrity constraints in a CS, in particular, for CSs specified in UML (Unified Modeling Language [31]) with constraints specified in OCL (Object Constraint Language [30]; constraints in OCL are defined in the context of a specific type, the *context type*, and must be satisfied by all instances of that type). Since the 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 say that the method is an *incremental* method, since, given the basic assumption that the IB is in a consistent state prior to its modification, it exploits 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). Structural events can be defined as elementary changes over the population of entity and relationship 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 work done during my stay at the Politecnico di Milano consisted on: 1 – validating the method of the thesis with additional examples, 2- optimizing it for the specific case of the web applications and 3 – extending it to deal with workflow-based applications.

Regarding the method optimization for web applications, we proposed several parameters that permit to tune the method implementation depending on the specific performance requirements of each web application. With respect to the workflows, we studied the set of constraints a workflow specification implies (activity ordering polices, access control policies,...). Once we have extracted all constraints from the workflow we can just apply the method over them to to generate efficient workflow-based applications as well.

The rest of this document is structured as follows. Section 2 briefly reviews the main parts of the method presented in my PhD Thesis. Section 3 studies the relation of this method (and in general, of integrity checking methods) with the specific field of web applications. Section 4 shows how the method can deal with workflow applications. Finally, Section 5 summarizes the main results of the research stay.

2. Method Overview

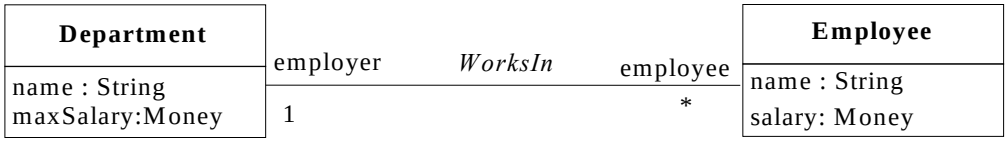
Given an initial conceptual schema *CS*, the result of my method for integrity checking is another conceptual schema *CS'* that, when executed or directly implemented in a particular technology platform, is able to check all constraints incrementally.

All CASE tools can benefit from our method if, once the designer has defined the conceptual schema *CS*, the tool uses my 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.

The method consists on three main steps. Each step partially improves the efficiency of the integrity checking process. Note that a direct integrity checking of an OCL constraint would imply 1 - checking the constraint after each modification of the IB and 2 - when checking the constraint, to evaluate the constraint body over all instances of its context type. As an example, a direct verification of the *MaxSalary* constraint (see the CS of Figure 2.2; the constraint states that the salary of an employee must be lower than the max salary defined in his/her department) would involve checking, after any kind of modifications, that for each department, all its employees still have a correct salary. Obviously this is really inefficient since this direct checking involves a lot of irrelevant verifications. In what follows we briefly describe, by means of this example, how the different steps of the method improve the efficiency of the *MaxSalary* integrity checking. Refer to [8] for a full description of the method.



context Department **inv** MaxSalary: self.employee->forAll(e| e.salary<=self.maxSalary)

Figure 2.2. Conceptual schema used as a running example

1 - Determining the structural events that may violate a constraint

Not all kinds of events may induce a violation of a given constraint. For instance, only the update of *salary* and *maxSalary* attributes and the insertion of a new relationship (i.e. link) in *WorksIn* may violate the previous *MaxSalary* constraint.

Other events, as the update of the name of a department, the removal of an employee or even the insertion of an employee (when the employee is not assigned to any department) does not violate the constraint. Hence, we may improve efficiency of integrity checking by discarding the verification of *MaxSalary* after executing operations that do not include any of these events.

2 - Computing the incremental expression to verify a constraint after executing a given structural event

To minimize the number of entities examined when checking a constraint *c* after an event *ev*, the method computes an OCL expression *exp* that can be used instead of *c* to verify that the state of the IB is still consistent with respect to *c* after the execution of *ev*. The state is consistent iff *exp* evaluates to true. As an example, the expression *exp* required to verify *MaxSalary* after a salary update over an employee *e* is *e.salary<=e.employer.maxSalary*. Note that, after this event, we just need to check the relationship between the updated employee (represented by the *self* variable) and his/her department. Therefore, we avoid verifying all departments (we just access the department where the modified employee works in) and for that department we just compare its salary with the one of the updated employee, thus, discarding the verification of the other employees working in the same department.

3 - Automatic generation of an efficient CS

With the previous feature alone the designer would be in charge of generating an implementation of the CS that benefits from *exp* to efficiently verify the constraint. However, the method is also able to modify an initial CS to ensure that all its constraints are efficiently verified.

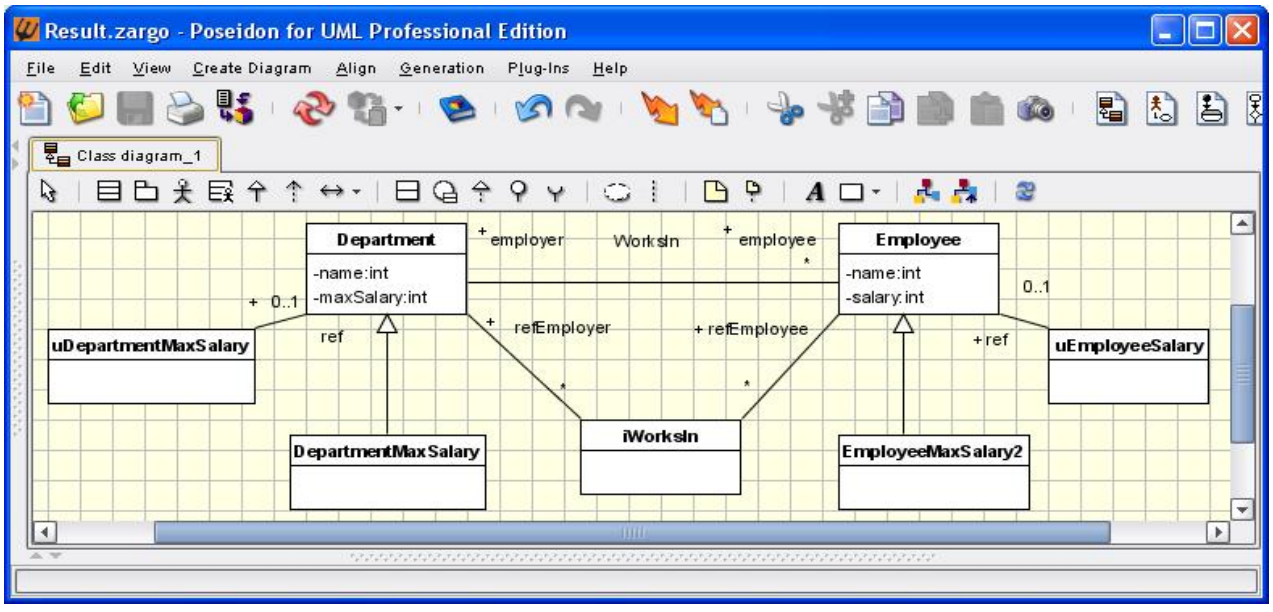
First, the method determines which are the events that may violate each constraint (with the techniques used in step 1). Then, for each event, it computes an appropriate alternative representation (though semantically-equivalent) of the constraint to be used when verifying the IB after executing events of that kind. Searching for this alternative representation may imply expressing the constraint using a different type of the CS as a context type.

Then, we process the CS in the following way: 1 - The CS is extended with a set of new types that record the events issued during the operation execution, 2 - Each constraint (either one of the original ones or one of the generated alternatives) is redefined to be evaluated only over the instances affected by the events (computed with the expressions obtained using the techniques of step 2) for which that constraint is an appropriate alternative. The redefinition process consists in changing its context type of the constraint to a new derived subtype of the

previous context type. These derived subtypes are designed (by means of their derivation rules) to hold only the affected instances. Therefore, after the redefinition, the constraint body is evaluated only over the instances of the derived subtype, i.e. over the affected instances.

Figure 2.3 shows the generated schema for the CS of Figure 2.2 (Poseidon for UML is used to display the processed schema as generated automatically by our prototype tool implementing the method) together with the redefined ICs and derivation rules. Note that during the process we have generated a new alternative for *MaxSalary* (*MaxSalary2*, defined using employee as a context type and with the body *self.salary <= self.employer.maxSalary*). This alternative is used to check the consistency of the data in the IB after the events *update of the salary of an employee* and *insert a new link between an employee and a department*. The original constraint is only used to check the IB after updates of the *maxSalary* attribute.

Both alternatives have been redefined over the new derived subtypes *EmployeeMaxSalary2* and *DepartmentMaxSalary*. According to their derivation rules, *DepartmentMaxSalary* contains those departments where the maxSalary attribute has been modified during the update of the IB while *EmployeeMaxSalary2* contains those employees that have been assigned to a department or that have changed the value of their salary attribute. The information about the executed events is recorded in the new types *uDepartmentMaxSalary*, *uEmployeeSalary* and *iWorksIn*.



context DepartmentMaxSalary inv MaxSalary:
self.employee->forAll(e: Employee | e.salary <= self.maxSalary)
context DepartmentMaxSalary::allInstances() : Set(Department) body:
uDepartmentMaxSalary::allInstances().ref->asSet()
context EmployeeMaxSalary2 inv MaxSalary2: self.salary <= self.employer.maxSalary
context EmployeeMaxSalary2::allInstances() : Set(Employee) body:
uEmployeeSalary::allInstances().ref->union(iWorksIn::allInstances().refEmployee)->asSet()

Figure 2.3 Generated schema

3. Constraint Tuning and Management for Web Applications[1]

3.1 Introduction

Despite the importance of integrity constraints (ICs) in conceptual modeling, current web development methods pay little attention to the problem of integrity checking. Among the conceptual modeling languages for the Web, some of them (e.g., W2000 [3]) provide constraint checking only at the metamodel level, thus constraining the way models should be designed, but not enforcing consistent states at runtime; others (e.g., WebML [11], [12]) use ECA rules (which is one of the typical IC implementation techniques), but for different purposes, such as exception handling and adaptivity rules definition. Most of the approaches (including the cited ones, OOH [19], OO-HDM [36], OO-Method [18], Strudel [17] and others) exploit rule-based approaches for personalization and dynamic page rendering. However, none of them currently address generic IC management for the information base, and in particular no efficient and flexible IC management implementation techniques are provided.

On the other hand, well-known efficient integrity checking methods in the deductive and relational database field exist since the beginning of the nineties (see [20] for a survey). These methods (also known as incremental methods) exploit available information about the structural events applied over the information base to consider as few entities of the information base as possible during the verification of ICs. A structural event is an elementary change in the population of an entity type (i.e. a class) or relationship type (i.e. an association) such as: create object, update attribute, create relationship link, etc. Structural events are a way to define the effect of the operations appearing in the conceptual schemas.

These methods could be integrated with web development approaches to guarantee and efficient integrity checking of the ICs defined in the conceptual schema. In particular, code-generation methods of web development approaches could be enhanced with incremental techniques (as the ones included in my method, see section 2) so that the generated implementation of the ICs is able to verify efficiently the consistency of the information base.

Although the wide variety and the complexity of requirements of web applications and of the scenarios where they are used ask for such integration, we believe that these complexity factor themselves slow down the adoption of IC management; indeed, they make impossible for a single incremental strategy to fit all kinds of requirements of any web applications. For instance, some web applications may be interested in checking the ICs after each single event issued by the user, while others may prefer to provide a more flexible user experience and to defer the integrity checking until the end of the transaction. Furthermore, some web

[1] Research done in collaboration with Marco Brambilla from the Politecnico di Milano

applications require the quickest response time to the user requests, even if, afterwards, it takes longer to verify the ICs at commit time, while others may accept a little overhead in the processing of single events in order to minimize the total transaction time. Finally, some applications may desire to postpone the integrity checking until certain periods of the day when they estimate a more reduced load on their servers.

Moreover, existing incremental methods do not take into account the dynamic models (i.e., hypertext applications in the web context) of applications. When processing the ICs for an efficient integrity checking, they always assume the worst scenario, which is that all kinds of structural events can apply over the information base. On the contrary, often, web applications offer limited data-management possibilities (i.e. not all data in the information base can be modified through the web application). In such a case, some of the structures created in the information base to ensure an efficient verification may become useless and could be removed to improve the run-time efficiency of the applications. Obviously, there is a strong assumption: the optimization approach must be aware of all the applications that are going to work on the considered information base, and optimize the constraint management considering the conceptual models of all these applications. In the following discussion, we assume to have one single web application working on the information base, although the generalization can be easily inferred.

The main goal of the proposal described in this section is to present a general framework to facilitate the integration of efficient integrity checking methods in web applications. This framework can be parameterized with the characteristics of a specific web application, at the purpose of offering the optimal set of techniques for implementing the verification of the imposed ICs.

Figure 3.1 shows the general architecture of the framework. The designer specifies the conceptual schema and the ICs, possibly using a CASE tool for web application design. The framework consists in three steps:

1. Given the set of ICs, the framework first analyses each IC to determine which kind of changes (i.e. which kind of structural events) to the information base may violate the IC [9]. We call this set of events the Potentially-violating Structural Events (PSEs) for the IC.

2. Given the conceptual model of the hypertext, the framework removes all PSEs related to events not appearing in the hypertext model/s, thus applying a pruning on the set of PSEs extracted by step 1.

3. In the third step, the framework takes into account the requirements and necessities of the web application (which are provided by the analyst as parameters) to recommend an implementation technique for each IC. Different ICs may require different implementation techniques. These recommendations may be sent back to the web design tool to guide the generation process of the code-generation module, or may be provided directly to the developer, as a roadmap for IC implementation.

Notice that the framework exploits two techniques (first the pruning of the PSEs; second the recommendation of the optimal implementation technique for each IC), which can be applied independently, each providing some optimization. Even the application of just one of the two can bring some efficiency advantages: it is possible to prune the PSEs and implement them in a standard way; or it is possible to implement the whole set of PSEs, but choosing the best implementation for each of them.

The rest of the section is structured as follows: we will use the running example introduced in Section 3.2 to better explain the approach throughout the section; Section 3.3 describes steps 1 and 2 of the framework, consisting in PSEs extraction and pruning; Section 3.4 describes the algorithm for computation of the best solution with respect to the specific needs of the application; Section 3.5 sketches some details about the framework implementation; finally, we present a summary and point out future work on this topic in Section 3.6.

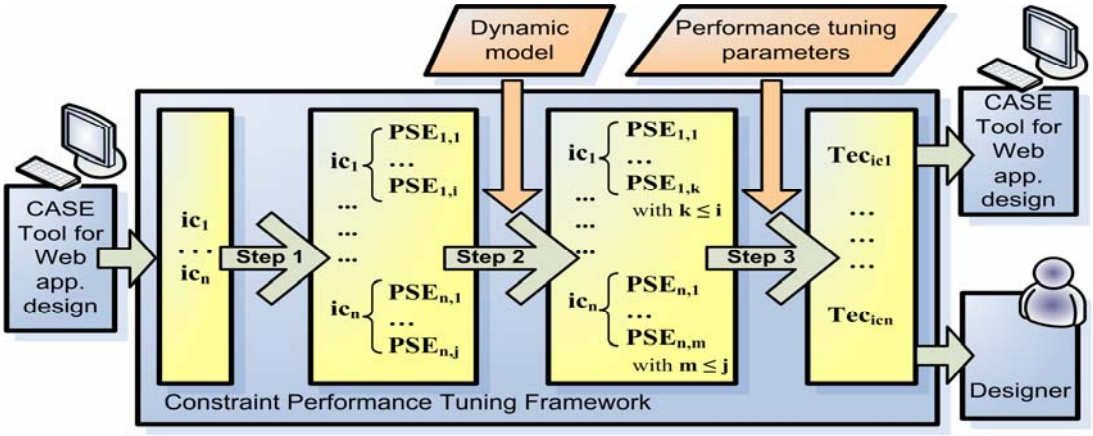


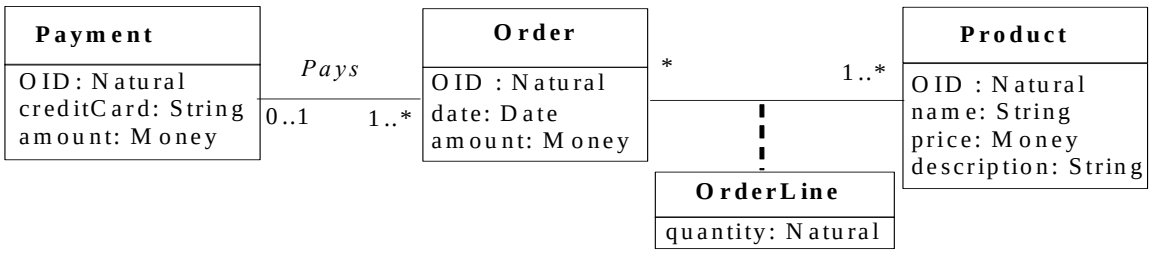
Figure 3.1 – Framework architecture

3.2 Running example

To illustrate the different techniques of our framework we will use as a running example the information base described by the conceptual schema of Figure 3.2, meant to (partially) model a simple e-commerce application.

The conceptual schema is specified using a UML class diagram compatible notation [31]. It contains information about the orders (*Order* entity type) and the products (*Product* entity type) they contain. The relationship type *OrderLine* registers the quantity of products of the same type included in a given order. Instances of the *Payment* entity type register the information of paid orders (a single payment may cover several orders).

Additionally, the conceptual schema includes three textual ICs expressed in OCL [30]. The first one (*CorrectProduct*) states that the product price must be a positive value. *ValidPayment* forces the amount of a payment to cover, at least, the sum of amounts of related orders. *MaxPendingOrders* ensures that the information system never holds more than a thousand unpaid orders.




```
context Product inv CorrectProduct: self.price>0
context Payment inv ValidPayment: self.amount>=self.order.amount->sum()
context Order inv MaxPendingOrders:
   Order.allInstances()->select(o| o.payment->isEmpty())->size() <= 1000
```

Figure 3.2 - Conceptual schema used as running example

Note that ICs in OCL are defined in the context of a specific type, the *context type,* and must be satisfied by all instances of that type. For instance, in the constraint *CorrectProduct*, *Product* is the context type, the variable *self* refers to an instance of *Product*, and the IC must hold for all possible different values of *self* (i.e. all instances of the *Product* entity type). Some OCL ICs (as *MaxPendingOrders*) require the use of the operator *allInstances*. *AllInstances* is a predefined feature on entity types that results in the set of all instances of the type in existence at the specific time when the expression is evaluated.

3.3 Role of dynamic models in constraint simplification

All efficient methods for integrity checking (see next section) need to create some additional data structures (e.g., triggers, temporary tables, views) to manage each event included in the list of PSEs for an IC. Obviously, if we can ensure that some of those PSEs are never going to be applied over the database, the run-time efficiency of our web application may be improved by discarding the generation of their additional data structures during the code-generation phase. This is the rationale that justifies the first technique of our framework: once we have extracted all the PSEs (step 1), we can prune some of them by considering which events will actually happen during the execution (step 2).

For web applications, we can decide if a certain PSE *p* needs to be considered, depending on whether *p* is included in any of the hypertext models of the web application. Otherwise, *p* will never happen during the run-time application execution. As already mentioned, if other applications modify the same information base, we should verify that *p* do not appear in their hypertext models (or dynamic models, in general) either, or ensure that these other applications always leave the database in a consistent state.

Let set_{PSE} be the set of PSEs of a constraint *ic* and set_{ev} be the set of structural events appearing in the hypertext models of the application. Then, the final set of PSEs set'_{PSE} we need to take into account when generating the IC is defined as:

$$set'_{PSE} = set_{ev} \cap set_{PSE}$$

Set'$_{PSE}$ could result in the empty set (meaning the ICs cannot possibly be violated by the application execution). Then, we can discard the whole IC management during the code generation phase. Although this situation may appear unnatural, it is not so uncommon, since typically the IC definition includes all the business constraints on the data, but then the actual application may implement a small subset of the management function of the information base, thus not implying any risk of violation of the ICs. For sake of transparency, the framework will be able to provide the total number of PSEs and the number of remaining PSEs after the pruning, thus making the designer aware of the situation.

3.3.1 Determining set_{PSE}

To precisely determine the set of PSEs for an IC (set_{PSE} in the previous formula), as required by step 1 of the method, we use the technique proposed in [9], which distinguishes the following kinds of structural events:

- InsertET(*ET*): insertion over the entity type *ET*;
- UpdateAttribute(*Attr*,*ET*): update of the value of attribute *Attr*.
- DeleteET(*ET*): deletion of an entity of the entity type *ET*;
- SpecializeET(*ET*): specialization of an entity of a supertype of *ET* to *ET*;
- GeneralizeET(*ET*): generalization of an entity of a subtype of *ET* to *ET*;
- InsertRT(*RT*): creation of a new relationship in the relationship type *RT*;
- DeleteRT(*RT*): deletion of a relationship of *RT*.

According to this method, the PSEs (set_{PSE}) for the example ICs (Figure 3.2) are:

- *CorrectProduct*: UpdateAttribute(Price,Product), InsertET(Product);
- *ValidPayment*: UpdateAttribute(Amount, Payment), InsertRT(Pays), UpdateAttribute(Amount, Order);
- *MaxPendingOrders*: InsertET(Order), DeleteRT(Pays).

Note that *ValidPayment* can be violated by: *(i)* changes in the amounts of the payment (event UpdateAttribute(amount, payment)); *(ii)* changes in the amount of the order (event UpdateAttribute (amount, order)); or *(iii)* by assigning new orders to existing payments (event InsertRT(Pays)); but no violation will apply after inserting new orders. Indeed, until new orders are not assigned to a payment, they cannot violate the IC.

3.3.2 Determining set_{ev}

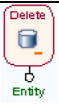
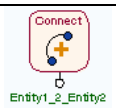
The computation of set_{ev} depends on the kind of dynamic model used to specify the application behavior. In this section we will use as a dynamic model the WebML hypertext models [12]. However, this is not a restriction imposed by our method. We can use any kind of dynamic model as long as we provide the correspondence between the operations appearing in the model and our predefined set of events. For instance, in [9] we presented the correspondence between our supported event types and the actions defined in the UML Actions package.

The specification of a WebML application consists of a data schema (as the one in Figure 3.2), and of one or more hypertext models (e.g., for different types of users or for different publishing devices), called site views, expressing the Web interface used to publish and manipulate this data. A site view is a graph of pages, consisting of connected units, representing at a conceptual level the atomic pieces of homogeneous information to be published: the content displayed by a unit is extracted from an entity type, possibly filtered. Units are connected by links carrying data from a unit to another, to allow computation of the hypertext and definition of navigational paths for users.

Hypertext models also include content-management units, used to specify update operations on the data underlying the site. Table 3.1 shows the basic update operations and the kind of event thrown by each operation. Note that WebML does not distinguish between the insertion of a completely new object (*InsertET* event) and the insertion in a subtype of an object already existing in a supertype (*SpecializeET* event). In both situations, the create unit is used and then, at run-time and depending on the unit parameters, the run-time environment decides the appropriate event. Likewise, *Delete* units can throw either *DeleteET* or *GeneralizeET* events.

Figure 3.3 shows an example of a WebML hypertext model. The hypertext allows the creation of new orders with their set of products (related to the orders through the *order lines*), and the creation of new payments along with the association to the orders the payment covers. Once the payment is submitted, the user can review the payment and possibly update the paid Amount. Notice that WebML normalizes relationship types in the conceptual model; for instance, *OrderLine* is represented as an entity type with two relationship types, one connecting to *Product* and the other to *Order*.

Table 3.1. Summary of WebML operations and corresponding events.

Symbol	Operation description	Thrown event
	*Create unit*: creation of a new instance of the *Entity* entity type	InsertET SpecializeET
	*Modify unit*: updates the value of one or more attributes.	UpdateAttribute
	*Delete unit*: deletes an instance of an entity type	DeleteET GeneralizeET
	*Connect unit*: creates a new relationship between two entities	InsertRT

	*Disconnect unit*: deletes a relationship between two entities	DeleteRT

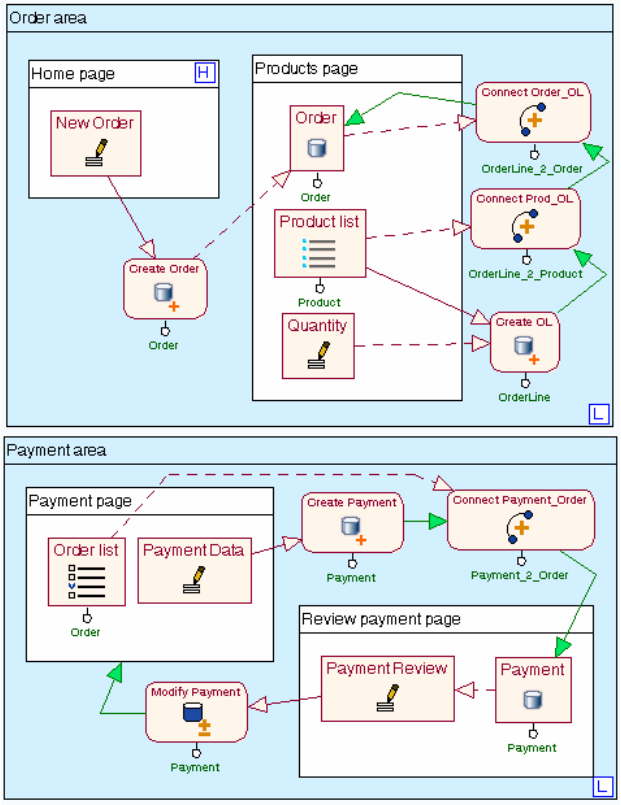


Figure 3.3 – Example of a WebML hypertext model

From this hypertext model we extract the following structural events (set_{ev}):

- *InsertET(Order)* due to the *Create Order* unit;
- *InsertET(Payment)* because of the *CreatePayment* unit;
- An *UpdateAttribute* event for each *Payment* attribute because of the *ModifyPayment* unit;
- *InsertRT(Pays)* due to the *Connect Payment_2_Order* unit;
- *InsertRT(OrderLine)* as a consequence of the *Create OL*, *Connect Prod_OL* and *Connect Order_OL* units. This is necessary because of WebML normalization: the creation of a new order line requires a create unit for the entity type, and two connect units for the two relationship types.

Note that events *SpecializeET(Order)* and *SpecializeET(Payment)* are not included since the *Order* and *Payment* types do not belong to any taxonomy, and thus, the corresponding create units always generate the corresponding *InsertET* event.

Following this example, when considering this hypertext model to simplify the ICs of Figure 3.2, the final set of PSEs for each IC, defined as $set'_{PSE} = set_{ev} \cap set_{PSE}$, results in:

- *CorrectProduct*: ∅ (we remove all the events since none of the PSEs for this IC appear in the hypertext model);
- *ValidPayment*: InsertRT(Pays) and UpdateAttribute(Amount, Payment) (we remove the Order Amount update event, since the hypertext model does not allow update of existing orders);
- *MaxPendingOrders*: InsertET(Order) (we remove the delete event since the deletion of existing payments is not permitted).

These are the only PSEs we need to consider when implementing the ICs, according to the PSEs pruning step.

3.4 Deciding the optimal technique

The aim of this section is to describe the algorithms for step 2 of our approach, which is to recommend for each IC the optimal implementation technique, given the set of ICs with their respective set of PSEs (possibly a pruned set as explained in Section 3.3). We define a technique as optimal when it is the one providing the best run-time efficiency of the application wrt the other ones compatible with the application requirements. Notice that sometimes more than one optimal solution is possible.

We also need to take the complexity of the IC into account when deciding the optimal technique. We distinguish three basic complexity levels (adapted from [38]):

- Intra-instance ICs: constraints restricting the value of the attributes of a single instance. Example: *CorrectProduct*;
- Inter-instance ICs: constraints restricting the relationships between an entity and other entities, instances of different entity types. Example: *ValidPayment*. Within this category, it is worth to distinguish the subcategory of ICs containing aggregator operators (like *sum*, *count*, *size*…);
- Type-level ICs: constraints restricting a set of objects of the same entity type. Example: *MaxPendingOrders*.

We classify OCL ICs in one of the above categories applying the following rules over their syntactic definition:

1. ICs defined using the *self* variable and not referencing any relationship type are considered Intra-instance ICs;
2. ICs defined using the *self* variable and not satisfying rule number 1 are considered Inter-instance ICs;
3. ICs not satisfying rules 1 and 2 are considered type-level ICs. These ICs require the use of the operator *allInstances*.

In what follows we present the different parameters that the designer can define (subsection 3.4.1), the possible implementation techniques for the ICs (subsection 3.4.2) and the discussion about the recommended techniques for each parameter combination (subsection 3.4.3).

3.4.1 Performance Tuning Parameters

For each IC, the designer may provide values for the following four performance tuning parameters: *checking time*, *run-time efficiency*, *population volumes*, and *usage of Database Management System (DBMS) support* if available. The purpose of step 3 is to identify the best mix of techniques for each IC, considering the parameter values. The admitted values for each parameter are the following:

1. Checking time: it refers to the moment when the IC must be verified:

a. *Immediate*: immediately after the execution of the structural event that may induce a constraint violation;

b. *Deferred*: at the end of the transaction including the event;

c. *Postponed*: at some future time point after the end of the transaction. This implies that for a period of time the database may be in an inconsistent state.

2. Run-time efficiency: in general, the designer is interested in minimizing the overall time devoted to the integrity checking of the ICs. However, in order to be efficient during the integrity checking, efficient IC techniques need to gather certain information about the events issued during the transaction. This implies a little overhead in the processing of those events. For some applications or environments even this little overhead may not be acceptable, and thus, the designer may prefer a lower efficiency in the integrity checking in exchange for a faster event processing. The possible values of this parameter are:

a. *Individual efficiency*: each single event over the information base must be processed as quickly as possible even if, afterwards, it takes longer to verify the ICs;

b. *Overall efficiency*: the total time of the transaction (time of processing each single event plus the time required for the integrity checking) must be as quick as possible, even if we lose some efficiency when processing each single event;

c. *Average efficiency*: a small delay for individual events is acceptable to gain a significant improvement in the efficiency during the integrity checking.

3. Volume of table population: refers to the expected number of tuples in the tables:

a. *High*: the more data is estimated in the tables the more important is the use of efficient implementation techniques

b. *Low*: the benefits of implementing efficient techniques it is not so clear. It may be faster to verify the whole table population (for instance, if the whole table can be loaded into main memory) than computing which tuples need to be verified and checking only those tuples

4. Usage of DBMS support: to use the DBMS support to handle the ICs when possible: *Yes*/*No* value.

Surely, it may be difficult to decide the right combination of parameter values at design time, since, when testing the recommended technique in the production environment, the situation may differ from the expectations. E.g., it may happen that the performance of the DBMS is better (or worse) than expected; the number of simultaneous users is bigger (or lower); the

volume of data in the tables higher (or lower); and so forth. In these cases, the designer should reapply the framework, adapting the value of some of the parameters to obtain a more tuned recommendation.

3.4.2 Alternative Techniques for implementing ICs in a Relational Database

During the code-generation process, the ICs defined in the CS must be transformed in a combination of code-fragments and/or data structures that ensure the consistency of the information base after changes produced by the applied structural events. Even though, in principle, several technologies could serve to implement the information base, in the following discussion of the example we assume that as information base we use a relational database.

In this work we adapt the strategies introduced by existing methods to provide a set of different techniques that may be used in the transformation process of ICs, ranging from a direct (inefficient) implementation of ICs to the most efficient techniques. The possible recommendations of the framework will be combinations and/or variations of the techniques presented here. It is worth to note that all techniques are able to provide a detailed user feedback regarding the exact instances that violate the IC.

The techniques are illustrated with examples over the *Payment* and *Order* tables (see schema definition in Figure 3.4.1), obtained from the types of our running example applying the usual mappings [12].

```
CREATE TABLE Payment( OID Integer primary key,
amount real, creditCard char(16));
CREATE TABLE Order(OID Integer primary key,
amount real, dateOrder Date,
payment Integer references payment(oid));
```

Figure 3.4.1. *Payment* and *Order* tables

3.4.2.1 Direct Implementation of ICs

The SQL standard defines the *assertion* mechanism to define general ICs in a database. However, current DBMS do not support this mechanism [38]. They only support the definition of some particular ICs as *primary* and *foreign keys*, *unique* constraints and *checks* over attribute values. This rather limited support not suffices to implement the general ICs defined in the conceptual schemas (from the running example of Figure 3.2 only *CorrectProduct* could be defined using the DBMS support).

Therefore during the code-generation phase we need to generate also our own data structures to implement and control the ICs. The simplest strategy when generating the ICs is to generate them in the form of inconsistency predicates. For each IC we create a view that returns a non-empty result if and only if the IC has been violated during the transaction. In particular, the tuples retrieved querying the view are the ones violating the IC. Before committing the transaction, we must query all generated views to check if all ICs still hold. If

any view instance is found, a violation occurred and the transaction must be rolled back (or a repair action must be triggered).

The SELECT clause of the view is derived from the constraint definition, in denial form, i.e. we want to select the data *not* verifying the constraint condition. It can be automatically obtained using already existing OCL-SQL transformation patterns [12],[14]. As an example, Figure 3.4.2 shows the view corresponding to the *ValidPayment* IC.

```
CREATE VIEW ValidPayment AS
SELECT * FROM Payment p WHERE not
  (p.amount >=
     (SELECT nvl(sum(amount),0)
      FROM order o WHERE o.payment=p.oid));
```

Figure 3.4.2. View for the *ValidPayment* IC

Note that the integrity checking process using these views is highly inefficient since to verify each IC we examine all tuples of the corresponding table and not only the ones that may violate the IC. For instance, after executing a transaction that simply modifies the amount of payment p_1, the *ValidPayment* view examines all payments to verify the IC, instead of checking just p_1. Besides, since we do not register the events executed during the transaction, we do not have any way to know at the end of the transaction which ICs have to be verified, and thus, we are forced to verify all of them.

3.2.4.2 Trigger Mechanism

ECA-Rules (implemented as triggers in a database) were initially proposed for integrity checking in [10]. The basic idea is to create a trigger for each event appearing in the list of PSEs for an IC. The body of the trigger is in charge of verifying that the modified tuple does not violate the IC. During a transaction, the DBMS fires the appropriate trigger immediately after the PSEs is issued. Since these triggers do not modify the state of the database, neither termination nor confluence problems occur.

Figure 3.4.3 shows the trigger verifying *ValidPayment* after the event *UpdateAttribute(amount,Payment)*. Note that the trigger just checks the updated payment and not all payments (as in the previous case). However, if we want to completely cover the *ValidPayment* IC, we should create additional triggers for its other PSEs.

```
CREATE TRIGGER tuPayment
AFTER UPDATE OF Amount ON Payment FOR each row
DECLARE
  v_Sum NUMBER; EInvalidPayment Exception;
BEGIN
  SELECT sum(o.amount) into v_Sum
  FROM Order o WHERE o.payment=:new.oid;
  IF not (:new.amount>=v_Sum) THEN
      raise EInvalidPayment; END IF;
END;
```

Figure 3.4.3. Trigger for *Payment* update events

3.2.4.3 Efficient Views

The rationale of most of the efficient (incremental) methods developed for deductive databases (see [20] for a survey) is to transform each ICs in a set of database views (one for each PSE) over the tuples modified during the transaction (instead of defining the views over the whole table population as in section 3.4.2.1). We refer to these views as *efficient views* to distinguish them from the *inefficient views* of section 3.4.2.1.

Implementation of these methods in current DBMS implies the creation of auxiliary tables to reflect the changes done by the structural events over the database tables. We call this tables *event tables*. Insertion and removal of tuples in event tables should be done automatically and transparently to the user. In relational databases this automatic update can be addressed by means of triggers monitoring the changes over the database tables. Since these triggers only modify the state of the event tables and no triggers are defined on them, no termination nor confluence problems appear. Moreover, event tables must be empty at the beginning of each transaction. We obtain automatically this behavior if we define them as temporary tables (part of the SQL:1999 standard [25]).

As an example, Figure 3.4.4 shows the efficient view in charge of verifying the *ValidPayment* IC after the update of payments. The table *uPaymentAmount* represents the event table for the *UpdateAttribute(amount,Payment)* event. The trigger *tuPayment* updates the *uPaymentAmount* table. Note that the SELECT expression of the view *uValidPayment* is exactly the same as in Figure 3.4.2 except for the table appearing in the *from* clause (now it is the *uPaymentAmount* table). This ensures the view only verifies the payments updated during the transaction. Note that for transactions not including any update event, the *uPaymentAmount* table is empty, and thus, the *uValidPayment* view does not check any payment. To completely verify *ValidPayment* we should create additional views (and event tables and triggers) for the other PSEs.

These views can become even more efficient if the triggers updating the event tables take into account the *net effect* of the transaction [10]. As an example, assume a transaction updating twice the amount of the same payment. Then, the previous trigger would insert a different tuple in *uPaymentAmount* for each update event, and thus, the same payment will be checked twice when querying the view *uValidPayment*. When the IC is complex (for instance, it includes aggregator operators) it may be interesting to avoid this situation by requiring the triggers to check that the tuple (i.e. the payment) is not already registered in the event table before inserting the new tuple.

```
CREATE GLOBAL TEMPORARY TABLE uPaymentAmount
(OID Integer, amount Integer) ON COMMIT DELETE ROWS;

CREATE TRIGGER tuPayment
AFTER UPDATE OF amount ON Payment FOR EACH ROW
BEGIN
 INSERT INTO uPaymentAmount
    VALUES (:new.oid,:new.amount);
END;

CREATE VIEW uValidPayment AS
SELECT * FROM uPaymentAmount p WHERE not
  (p.amount >=
     (SELECT nvl(sum(amount),0)
      FROM order o WHERE o.payment=p.oid));
```

Figure 3.4.4. Efficient view for the ValidPayment IC

3.4.2.4 Semi-efficient approach

We propose this technique as an intermediate technique between the direct verification and the efficient view techniques. The aim of this technique is to register enough information to avoid the verification of ICs not affected by the events executed during the transaction but, for those ICs that need to be verified, the technique examines all the table population.

With this approach we create a statement-level trigger for each PSE. In contrast with the previous row-level triggers, statement-level triggers fire only one time per statement regardless the number of tuples modified. These triggers simply register in the event table the execution of the event and *not* the modified instance (see Figure 3.4.5). Now, the event table can be as simple as a table with a single attribute. The existence of at least one tuple in the event table means that the corresponding PSE has been issued, and thus, it is necessary to check the IC. Then, the IC can be checked using the views of section 3.4.2.1. The difference is that now we know the exact views we need to query, and thus, we avoid verifying all ICs.

```
CREATE GLOBAL TEMPORARY TABLE uPaymentAmount
   (Event Char(1)) ON COMMIT DELETE ROWS;

CREATE TRIGGER tuPayment
AFTER UPDATE OF amount ON Payment FOR EACH STATEMENT
BEGIN
  INSERT INTO uPaymentAmount VALUES ('1');
END;
```

Figure 3.4.5. Trigger example for the *ValidPayment* IC in the semi-efficient approach

3.4.3 Recommended Techniques

Given the web application performance tuning parameters (Section 3.4.1) and the complexity of each IC (beginning of Section 3.4) we recommend in this section the optimal technique/s (among the ones presented in Section 3.4.2) to implement the IC. We organize the discussion taking as a basis the value of the *checking time* parameter since is the one influencing the most the technique to select. This value is combined with the possible values of the other parameters to get the optimal technique. A simplified version of this discussion is summarized

in Table 3.2 (for the sake of simplicity assuming a high populated database and the use of the constraint management features of the DBMS when possible).

Checking time: Immediate
For immediate checking, the best alternative is always the trigger technique (section 3.4.2.2). It makes no sense to register the event in an event table and then immediately query the corresponding view to check if the event violates the IC; the efficiency would be reduced. The value of the *run-time efficiency* parameter is irrelevant since we are forced to verify the IC just after the event execution.

If the designer wants to use the DBMS capabilities for constraint definition, Intra-instance ICs should be defined as *checks*. Likewise with unique constraints. In fact, we recommend this option. Although we could use the triggers to manage this kind of constraints efficiently, it is reasonable to assume that the DBMS will be even more efficient in managing them.

For type-level ICs, due to their complexity, an immediate verification impairs the run-time efficiency of the application. If possible, their verification should be deferred until the end of the transaction. As an example, imagine that we want to verify *ValidPayment* after each event. A transaction updating the amount of several orders would verify the constraint after each update, even if before finishing the transaction the user also updates the corresponding payment to reflect the changes in the order amounts.

Checking time: Deferred
The best technique depends on the *run-time efficiency* parameter:

(i) Individual efficiency: no overhead in the processing of the events is permitted. Thus, the only possible solution is a direct verification of the ICs using inefficient views (Section 3.4.2.1). When reusing the capabilities of the DBMS, the checks and unique constraints must be defined as deferred to avoid its verification until the end of the transaction. Nevertheless, the *individual* parameter value is only recommended for low populated tables, otherwise the total transaction time may become unacceptable.

(ii) Overall efficiency: the optimal technique is the efficient views technique (Section 3.4.2.3) for intra and Inter-instance constraints. For these kinds of ICs, knowing the relevant instances (the ones modified by the structural events) heavily reduces the time required for the integrity checking. For type-level constraints, since we need to examine all the tuples of the table restricted by the constraint, it is a waste of time to register the exact instances modified during the transaction. However, it is still a key issue to know whether the IC must be verified. Therefore, the most appropriate technique for these ICs is the semi-efficient approach (Section 3.4.2.4). Depending on the value of the *use of DBMS support* parameter, Intra-instances ICs can be defined as checks with the deferred clause. Again, we recommend this option. For low populated tables, the semi-efficient approach technique can also be an acceptable solution for all ICs.

(iii) Average efficiency: the recommended technique is the semi-efficient approach for all kinds of ICs. This way we get the best trade-off between the low overhead for each single event (the triggers are statement-level and do not compute the net effect of the transaction) and a semi-efficient verification at the end of the transaction (we know which views to query). Intra-instance constraints can be defined as checks with the deferred clause. For highly populated databases, we should consider also the application of the *efficient view* technique, if necessary without applying the *net effect* improvement to fasten the trigger execution.

Checking time: Postponed

The main characteristic of this alternative is that ICs are not verified during the transaction but at some point after the transaction has committed. Unfortunately, DBMSs do not support this feature at all. Therefore, we cannot count on any support from the DBMS. The only possibility is to disable the constraints and enable them just before starting the integrity checking. However in such a case the user (or application) starting the process must connect to the database with administrator privileges. Besides, the verification would be inefficient because when enabling the ICs, all tuples would be verified.

Additionally, when proposing the optimal techniques for this alternative, we must have in mind that the number of unverified structural events (events belonging to transactions committed after the last integrity checking) may be huge.

Depending on the *run-time efficiency* parameter, the proposed techniques are the following:

(i) Individual efficiency: again, the only possible technique is the use of inefficient views. However, in this case, the alternative is acceptable even in not so low-populated tables since the (probably) huge number of modified (and unverified) tuples reduces the difference between an efficient verification and a direct verification.

(ii) Overall efficiency: the recommended technique for high populated databases is still the efficient view technique. Despite the number of tuples unverified, for high populated databases the difference between an efficient and a direct verification is still relevant, especially for Inter-instance ICs including aggregator operators. Otherwise, the overhead on the single events may not be worthwhile and we could move to the semi-efficient approach. On the other hand, the event tables registering the modified tuples cannot be temporary since we need to keep their information beyond commit time (in particular, until the integrity checking takes place). Afterwards, their data must be truncated, but with this approach, the DBMS cannot do this process automatically and the application is in charge of it. For type-level constraints we still can apply the semi-efficient approach even though, with so many unverified structural events, most ICs will need to be verified.

(iii) Average efficiency: in the postponed context, this alternative is practically useless and the designer should opt for one of the previous two. As we have just commented, when starting the integrity checking there exist many pending transactions to be verified. Considering that each transaction may be composed by several structural events, it is quite probable that each IC finds that at least one of these events appear in its list of PSEs, and thus, that it needs to be

verified. Therefore, it is difficult to justify the advantage of registering the events to decide which ICs need verification. It depends on the number of committed transactions pending to verify, on their number of events and on the diversity of those events (when most of the transactions are composed by a repetitive set of events, we may still find that several ICs should not be verified, and thus, it is relevant to know which ones).

The situation where this alternative becomes worthwhile is when some ICs are extremely complex to verify. For those ICs a small overhead for registering their PSEs is acceptable, at the purpose of being absolutely sure that they are not checked unnecessarily.

Table 3.2. Summary of the recommended techniques. Cells show the optimal technique for each parameter combination. N/R means that the combination is not recommended.

Checking time	**IC complexity**	**Run-time efficiency**		
		Individual	Overall	Average
Immediate	Intra	*checks*	*Checks*	*checks*
	Inter	*triggers*	*Triggers*	*triggers*
	Type	*N/R*	*N/R*	*N/R*
Deferred	Intra	*checks (deferred)*	*checks (deferred)*	*checks (deferred)*
	Inter	*inefficient views*	*efficient views*	*semi-efficient*
	Type	*inefficient views*	*semi-efficient*	*semi-efficient*
Postponed	Intra	*inefficient views*	*efficient views*	*N/R*
	Inter	*inefficient views*	*efficient views*	*N/R*
	Type	*inefficient views*	*semi-efficient*	*semi-efficient*

3.5 Implementation

A prototype implementation of the concepts presented in this section has been developed. Given the XMI file [29] representing the conceptual schema and the set of ICs in a textual form (parsed using Dresden OCL Toolkit), the prototype computes the set of PSEs for each IC (step 1). Optionally, the user can provide a XML file containing a WebML project for specifying the hypertext dynamic model of the application, from which the prototype extracts the events to simplify the previous set of PSEs (step 2). Finally, given the performance tuning parameters provided by the designer, the tool returns the set of optimal techniques for each IC (step 3), as a set of implementation recommendations.

The prototype does not actually generate the SQL expressions to implement the recommended techniques, since already existing tools can be used with this purpose. For instance, WebRatio or OCL2SQL (from the Dresden toolkit) are able to generate the required tables and views. As a further work we plan a complete integration of the framework within the WebRatio tool.

3.6 Summary

In this section we have presented an approach for efficient integrity constraint management in Web application. The approach is based on the widely adopted incremental methods, which work by extraction of violating events and definition of appropriate support structures for efficient implementation. Our main contribution consists in a method (and an experimental tool) for allowing the designer of Web applications to easily integrate ICs management within the implementation of the application. As a final comment, we remark that query efficiency is not damaged by the extra data structures needed for efficient implementation of the constraints, therefore applications mostly based on data query (e.g., content publishing web applications) can benefit from this approach.

Future and ongoing work on this topic would include: quantitative comparison between the various implementation techniques when applied on real industrial applications, and also implementation of a more sophisticated GUI for the experimental tool, thus facilitating usage by designers and integration within the WebRatio tool.

4. Automatic Generation of Workflow-extended Conceptual Schemas[2]

4.1. Introduction

Specification of complex business applications usually requires the definition of a workflow model to express logical precedence and constraints among the different business activities (i.e. the units of work).

Workflow models are usually implemented with the help of dedicated workflow management systems (e.g., [21], [34]) which are heavy-weight applications focused on the control aspects of the workflow enactment. Alternatively, some approaches propose a direct implementation of the workflow model in the final technology platform, generally in the form of triggers in a relational database [2], or as constraints coded in the final application to force the user to perform tasks in the correct order. The latter solution is typically adopted in the new generation of Web applications and e-solutions, which require the management of collaborative applications and workflows, spanning multiple individuals and organizations. In such applications the business process constraints are implemented as hypetextual links and buttons placed properly in Web pages, thus restricting the user navigation depending on the workflow precedence constraints [4].

In this proposal we adopt a different approach and advocate for the integration of the workflow model with the domain *conceptual schema (CS)*. Starting from the workflow model it is possible to define a full fledged conceptual schema enriched with the entity and relationship types needed to record the required workflow information (mainly the activities of the workflow and the enactment of these activities in the different workflow executions) and with a set of process constraints over such types to control the correct workflow execution. We refer to this resulting conceptual schema as the *workflow-extended CS*. We will represent it using UML class diagrams [31] and will use OCL [30] to specify the process constraints.

The main characteristic of a workflow-extended CS is that it automatically ensures a consistent behavior of all enterprise applications with respect to the business process specification. As long as the applications properly update the workflow information in the CS, the process constraints defined in the schema enforce that the different tasks are done according to the initial workflow model. Another advantage of a workflow-extended CS is its technological-independence. Indeed, any method and tool designed for managing a generic CS can benefit from a workflow-extended CS, no matter the target technology platform or the purpose of the tool, spawning from direct application execution, to automatic generation of the implementation (including the integrity constraints by means of the method presented in

[2] Research done in collaboration with Marco Brambilla and Sara Comai from the Politecnico di Milano

section 2), to property analysis, and to metrics measurement. Those methods do not need to be extended to cope with the generation of workflow-extended CS, because the workflow and the domain parts of the CS have a homogeneous representation. Moreover, workflow-extended CS enables the definition of more expressive constraints, including timing conditions [13] or involving both workflow and domain information. These constraints are generally not allowed in workflow definition languages since they require using a general-purpose (textual) sublanguage [16], and on the other hand may be quite complex and tedious to specify using a general-purpose CS alone.

As far as we know, ours is the first proposal where both workflow information and process constraints are automatically derived from a workflow model and integrated within the conceptual schema. In literature, workflow metadata and OCL constraints have only been used in [15] to manually specify workflow access control constraints and derive authorization rules, in [1] to express constraints with respect to the distribution of work to teams, and in ArgoUML [23] to check for well-formedness in the design of process models. In our previous works we treated the automatic generation of workflow-based applications in the Web context from a high-level language (namely, WebML) [4, 7] and used temporal logic rules for the static verification of the generated applications [6]: however, these works were not technologically-independent nor did cover all expressivity tackled in our proposal.

The rest of the section is structured as follows: in subsection 2 the basic workflow concepts and our case study are illustrated. In subsections 3 and 4 we provide the definition of the workflow-extended conceptual schema and of the OCL process constraints, respectively. Subsection 5 presents possible approaches for the implementation of the workflow-extended CS and in subsection 6 we sum up our proposal and present future work.

4.2. Basic workflow concepts

Several visual notations, languages, and methodologies to specify workflow models have been proposed, with different expressive power, syntax and semantics. In our work we have adopted the Workflow Management Coalition terminology [39] and the BPMN [33] notation.

The workflow model is hence based on the concepts of *Process* (the description of the business process), *Case* (a process instance), *Activity* (the elementary unit of work composing a process), *Activity instance* (an instantiation of an activity within a case), *Actor* (a user role intervening in the process), *Event* (some punctual situation that happens in a case), and *Constraint* (logical precedence among activities and rules enabling activities execution). Processes can be internally structured using a variety of constructs: sequences of activities; gateways implementing AND, OR, XOR splits, respectively realizing splits into independent, alternative and exclusive threads; gateways implementing joins, i.e., convergence point of two or more activity flows; conditional flows between two activities; loops among activities or repetitions of single activities. Each construct may involve several constraints over the activities.

Our approach covers a large subset of the full expressive power of BPMN, with the following exceptions: we do not provide support to the concepts of transactions (i.e., one or more activities to be executed with transactional properties), nested subprocesses, and a few combinations of primitives, such as the direct concatenation of several gateways (i.e., the outgoing flow of a gateway cannot be the input flow of another gateway. However, the desired effect can be obtained by introducing fake activities between the gateways).

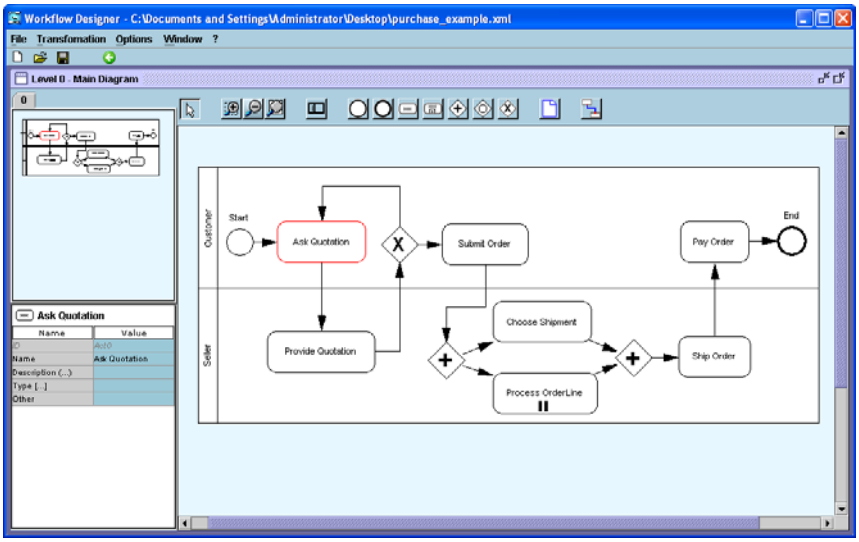


Figure 4.2.1 - Example of a workflow schema

In the sequel, we will exemplify the proposed approach on a case study consisting of a workflow implementing a simplified purchase process, as illustrated in Figure 4.2.1. According to the BPMN semantics, the depicted diagram specifies a process involving two actors (represented by the two swimlanes): a customer and a seller. The customer starts the workflow by asking for a quotation about a set of products (*Ask Quotation* activity). The seller provides the quotation and the customer may decide (exclusive choice) to modify his request or to accept it; in the former case, the quotation request and response cycle is repeated. In the latter case the customer submits the order and the seller takes care of it. The order management requires two parallel activities to be performed: the choice of the shipment options and the internal management of each order line. The management of the order lines is represented by the multi-instance activity called *Process OrderLine*: a different instance of the activity is started for each order line included in the submitted order. Once all order lines have been processed and the shipment has been decided (i.e., after the synchronization of the two branches by means of the AND merge), the order is shipped and the customer pays the corresponding amount.

4.3. Extending conceptual schemas with workflow information

Assuming that the application is initially specified by a conceptual schema describing the domain information, the conceptual schema of the workflow-based application can be obtained by extending the domain schema with some additional elements. In our case study, the domain schema (see the bottom part of Figure 4.3.1) includes entity types *Product*,

Quotation, *QuotationLine,* and *Order*. A *Quotation* is defined as a set of *QuotationLines*, each of them referring to a *Product*. When accepted by the customer, a *Quotation* generates an *Order*. Then, the quotation lines of the quotation associated to the submitted order are referred to as order lines.

The extensions to the schema include: *(i) user*-related information, *(ii) workflow*-related information, *(iii)* a set of possible relationships between the domain conceptual schema, the workflow information and the user information, and *(iv)* a set of process constraints guaranteeing a consistent state of the whole conceptual schema with respect to the workflow definition. This extended model can be (semi-) automatically derived from the workflow model.

We define a workflow-extended conceptual schema as follows. Given an initial conceptual schema CS with entity types $E=\{e_1,\ldots,e_n\}$, representing the knowledge about the domain, and a workflow model w with activities $A=\{a_1,\ldots,a_m\}$, the workflow-extended CS is obtained in the following way:

i) *User subschema:* User-related information is added to the CS by means of two entity types (see the top-left part of Figure 4.3.1):
   - Entity type *User* represents workflow actors.
   - Entity type *Group* represents groups of users, having access to the same set of tasks. A user may belong to different groups, and this is specified by a relationship type between *User* and *Group*.

ii) *Workflow subschema:* Workflow-related information (see the top-right part of Figure 4.3.1) includes the following entity types and their relationship types:
   - Entity type *Process* represents the supported workflow and is characterized by a name and a description.
   - Entity type *Case* denotes an instance of a process, which has a name, a start time, an end time, and a status, which can be: ready, active, cancelled, aborted, or completed.
   - Entity type *ActivityType* represents the classes of activities that compose a process. Activity types are assigned to groups of users, which are responsible of managing them.
   - Entity type *ActivityInstance* denotes the occurrence of an activity within a case, described by the start time, the end time, and the current status, which can be: ready, active, cancelled, aborted, or completed. Only one user can execute a particular activity instance, and this is recorded by the relationship type Performs. We define a Precedes relationship between activities to keep track of their execution order.
   - Entity type *EventType* represents the events that may affect the sequence or timing of activities of a process (e.g., temporal events, messages etc.). There are three different kinds of events (*eventKind* attribute): start, intermediate, and end event. For start and intermediate events we may define the triggering mechanism (*eventTrigger*). For end events, we may define how they affect the case execution (*eventResult*).
   - Entity type *EventInstance* denotes the occurrence of an event.
   - For each activity $a \in A$, a new subtype s_a is added to the entity type *ActivityInstance*. The name of the subtype is the name of a (e.g., in Figure 4.3.1 we introduced *ProcessOrderLine*, *AskQuotation*, *ShipOrder,* and so on).

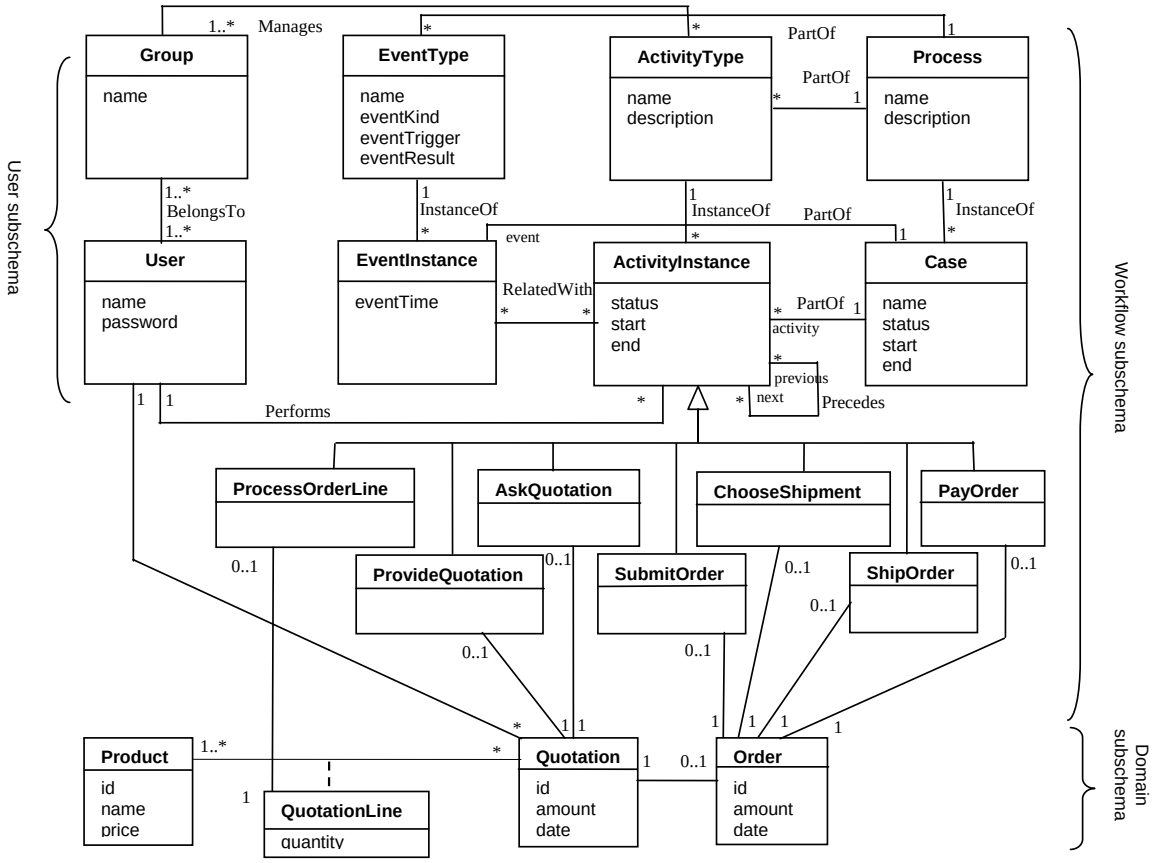


Figure 4.3.1 - Example of a workflow-extended conceptual schema

iii) *Relationship types between workflow subschema and domain subschema*: each subtype s_a is related with a (possibly empty) set of entity types $E_a \subseteq E$. These new relationship types are required to evaluate conditions appearing in the process constraints or to record the entities modified during the execution of a certain activity. In the case study (see relationships between the workflow and domain information in Figure 4.3.1), a set of relationship types are established: *Quotations* are associated to the activities *Ask Quotation* and *Provide Quotation*; *QuotationLines* are associated to the *ProcessOrderLine* activity; and *Orders* are associated to the activities *Submit Order*, *Choose Shipment*, *Process OrderLine*, *Ship Order*, and *Pay Order*. This means that the domain information is managed by the respectively connected activities.

The associations between the domain and the workflow subschemata must be manually specified by the designer, since the workflow model does not contain enough information to automatically relate both subschemata, if we do not consider auxiliary primitives for describing the data flow between activities. All the other components of the workflow-extended CS can be instead automatically generated.

This workflow-extended CS contains all types required to record the process information. It can be possibly enriched with further relationship types and/or attributes, according to the requirements of the specific workflow application. Next section complements the CS with a set of constraints, deduced from the workflow model, to ensure a consistent state of the CS regarding the workflow definition.

4.4. Translation patterns for process constraints

The structure of the workflow model implies a set of constraints regarding the execution order of the different activities, the number of possible instances of each activity in a given case, the conditions that must be satisfied in order to start a new activity, and so forth. These constraints are usually referred to as *process constraints*.

The behavior of all enterprise applications must always satisfy these constraints. Therefore, the generation of the workflow-extended CS must take all process constraints into account. Process constraints are translated as constraints over the population of the $s_{a1},\ldots,s_{am}$ activity instance entity subtypes. The translation of process constraints guarantees that any update event over the population of one of these subtypes (for instance, the creation of a new activity instance or the modification of its status) will be consistent with the process constraints defined in the workflow model.

We specify process constraints by means of invariants written in the OCL language. Invariants in OCL are defined in the context of a specific type, the *context type*. The actual OCL expression stating the constraint condition is called the *body* of the constraint. The *body* 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. The body expression may refer to attributes and relationships of the entity type. For instance, a constraint like: *context A inv: condition,* implies that all instances of *A* must verify *condition*.

Next subsections define a set of patterns for the generation of the process constraints corresponding to the different constructs appearing in workflow models (sequences, split gateways, merge gateways, conditions, loops…). The patterns can be combined to produce the full translation of the workflow model. As an example, we provide in Section 4.4.7 the translation of the workflow model of Figure 4.2.1.

Note that some constructs admit several graphical representations equivalent to the ones used in this section (see [33] for details). Moreover, the workflow language defines some complex constructs that can be derived from the basic ones, such as complex gateways and event-based gateways, not addressed here due to lack of space.

4.1 Sequences of activities

A sequence flow between two activities (Figure 4.4.1) indicates that the first activity must be completed before starting the second one. Moreover, if the first activity is completed within a given case, the second one must be eventually started[3] before ending the case.

[3] We do not require the second activity to be completed, since, for instance, it could be interrupted by the trigger of an intermediate exception event.

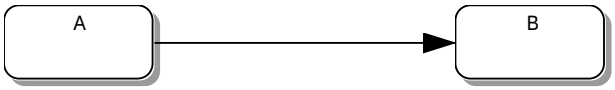


Figure 4.4.1 – Sequence flow

This behavior can be enforced in the workflow-extended CS by means of the following set of constraints:

- A constraint seq_1 over the entity type corresponding to the destination activity (*B* in the example) stating that for all activity instances of type *B* the preceding activity instance must be of type *A* and that it must have been already completed

  context B inv seq_1*: previous->size()=1 and previous->exists(a| a.oclIsTypeOf(A) and a.status='completed')*

- A constraint seq_2 over the second activity to prevent the creation of two different *B* instances related with the same *A* activity instance

  context B inv seq_2*: B.allInstances()->isUnique(previous)*

- A constraint seq_3 over the *Case* entity type verifying that when the case is completed there exists a *B* activity instance for each completed *A* activity instance. This *B* instance must be the only instance immediately following the *A* activity instance.

  context Case inv seq_3*: status='completed' implies self.activity-> select(a| a.oclIsTypeOf(A) and a.status='completed')-> forAll(a|a.next-> exists(b|b.oclIsTypeOf(B)) and a.next->size()=1)*

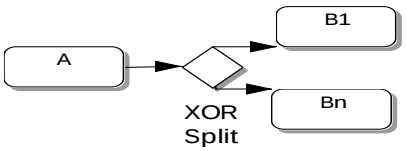
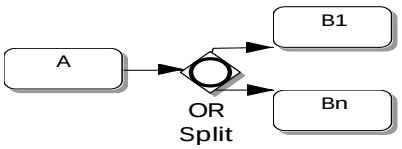
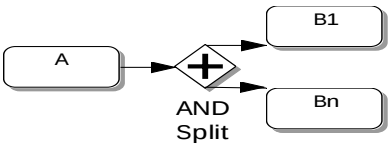
4.4.2 Split gateways

A split gateway is a location within a workflow where the sequence flow can take two or more alternative paths. The different split gateways differ on the number of possible paths that can be taken during the execution of the workflow. For *XOR-split* gateways only a single path can be selected. In *OR-splits* several of the outgoing flows may be chosen. For *AND-splits* all outgoing flows must be followed.

Table 4.1 shows for each kind of BPMN split gateway (first column) the process constraints required to enforce the corresponding behavior (second column).

Beside the process constraints appearing in the table, we must also add to all the activities $B_1 \ldots B_n$ the previous constraints seq_1 and seq_2 to verify that the preceding activity *A* has been completed and that no two activity instances of the same activity B_i are related with the same preceding activity *A*. We also require that, for all split gateways, the activity instance/s following *A* is of type B_1 or … or B_n.

Table 4.1 Process constraints for split gateways

Split gateway	Process constraints
	- Only one of the $B_1..B_n$ activities may be started *context A inv: next->select(a| a.oclIsTypeOf(B_1) or ... or a.oclIsTypeOf(B_n))->size()<=1* - If *A* is completed, at least one of the $B_1..B_n$ activities must be created before ending the case *context Case inv: status='completed' implies activities-> select(a|a.oclIsTypeOf(A) and a.status='completed')-> forAll (a|a.next->exists(b|b.oclIsTypeOf(B_1)or..or b.oclIsTypeOf(B_n)))*
	- Since several $B_1..B_n$ activities may be started, we just need to verify that if *A* is completed, at least one of the $B_1..B_n$ activities is created before ending the case (like in the XOR gateway above)
	- If *A* is completed, all $B_1..B_n$ activities must be eventually started *context Case inv:status='completed' implies activites->select(a| a.oclIsTypeOf(A) and a.status='completed')->forAll(a| a.next->exists(b| b.oclIsTypeOf(B_1)) and ... and a.next->exists(b|b.oclIsTypeOf(B_n)))*

4.4.3 Merge gateways

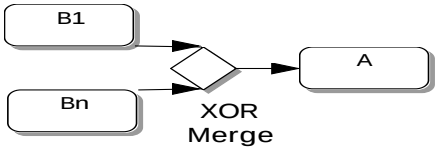
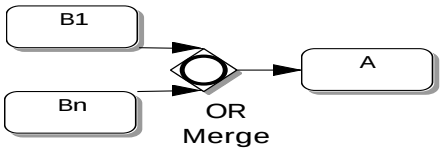
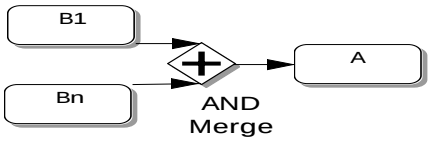
Merge gateways are useful to join or synchronize alternative sequence flows. Depending on the kind of merge gateway, the outgoing activity may start every time a single incoming flow is completed (*XOR-Merge*) or must wait until all incoming flows have finished in order to synchronize them (*AND-Merge* gateways).

The semantics of the *OR-Merge* gateways is not so clear. According to the BPMN standard, the system should wait for all sequence flows that have been started upstream. It does not require that all incoming sequence flows finish, but that all the flows that have actually started finish. Unfortunately, except for very simple workflows models it is not possible, even at run-time, to compute the exact set of incoming flows we should wait for. In this section, we propose a simpler but also more feasible solution. We always wait for at least an incoming flow and, depending on the workflow model, we also determine at design time (by examining the structure of the workflow model) the activities the *OR-Merge* must surely wait for. Given a set of incoming activities $B_1..B_n$, the *OR-merge* must wait for an activity B_i if there is a path in the workflow from the start event till the activity B_i that does not include any *OR-split* or *XOR-split* gateways. Notice that, depending on the workflow structure, this set could be empty. In this case the *OR-merge* waits for the first incoming flow.

Table 4.2 presents the different translation patterns required for each kind of merge gateway. Beside the constraints included in the table, a constraint over *A* should be added for all

gateways to verify that no two *A* instances are created for the same incoming set of activities (i.e. the intersection between the *previous* instance/s of all *A* instances must be empty).

Table 4.2 Process constraints for merge gateways

Merge gateway	Process constraints
	- All *A* activity instances have as a previous activity instance a completed activity instance of type B_1 or B_2 or … or B_n *context A inv: previous->size()=1 and previous-> exists(b| (b.oclIsTypeOf(B1) or ... or b.oclIsTypeOf(Bn)) and b.status='completed')* - Each $B_1..B_n$ activity instance is followed by an *A* activity *context Case inv: status='completed' implies activity-> select (b|b.oclIsTypeOf(B1) or ... or b.oclIsTypeOf(Bn))-> forAll(b|b.next->exists(a| a.oclIsTypeOf(A)))* - An *A* activity instance must wait for at least an incoming flow *context A inv: previous->select(b| (b.oclIsTypeOf(B_1) or ... or b.oclIsTypeOf(B_n)) and b.status='completed')->size()>=1*
	- An *A* activitiy instance should also wait for a complete instance of all the activities B_i .. B_k, that the merge must surely wait for because of the workflow topology (see above). *context A inv: previous->forAll(b| (b.oclIsTypeOf(B_i) or ... or b.oclIsTypeOf(B_k)) and b.status='completed')* - An activity instance of type *A* must wait for a set of activities $B_1..B_n$ to be completed *context A inv: previous->exists(b| b.oclIsTypeOf(B1) and b.status='completed') and ... and previous->exists(b| b.oclIsTypeOf(Bn) and b.status='completed')*
	- Each set of completed $B_1..B_n$ activity instances must be related with an *A* activity instance. *context Case inv: status='completed' implies not (activity->exists(b| b.oclIsTypeOf(B1) and b.status='completed' and not b.next->exists(a| a.oclIsTypeOf(A)) and ... and activity->exists(b| b.oclIsTypeOf(Bn) and b.status='completed' and not b.next->exists(a| a.oclIsTypeOf(A)))*

4.4.4 Condition constraints

The sequence flow and the OR-split and XOR-split gateways may contain condition expressions to control the flow execution at run-time. As an example, Figure 4.4.2 shows a conditional sequence flow. In the example, the activity *B* cannot start until *A* is completed and the condition *cond* is satisfied. The condition expression may require accessing the entity types of the domain subschema related to *B* in the CS. Through the Precedes relationship type, we can also define conditions involving the previous *A* activity instance and/or its related domain information.

To handle these condition expressions we must add, for each condition defined in a sequence flow or in an outgoing link of OR and XOR gateways, a new constraint over the destination activity. The constraint ensures that the preceding activity satisfies the specified condition. The constraint follows the following pattern:

context B inv: previous->forAll(a| a.cond)

Note that these additional constraints only need to hold when the destination activity is created, and thus, they must be defined as *creation-time constraints* [28].

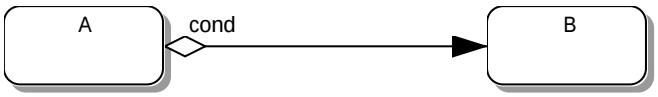


Figure 4.4.2 – A conditional sequence flow

4.4.5. Loops

A workflow may contain loops among a group of different activities or within a single activity. In this latter case we distinguish between *standard* loops (where the activity is executed as long as the loop condition holds) and *multi-instance* loops (where the activity is executed a predefined number of times). Every time a loop is iterated a new instance of the activity is created. Figure 4.4.3 shows an example of each loop type.

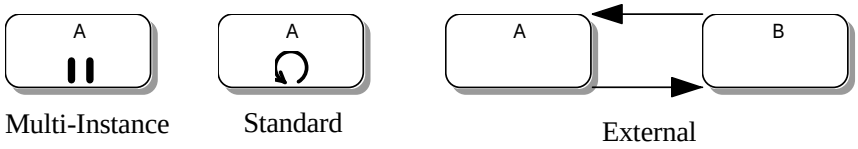


Figure 4.4.3 – Loop examples within a single activity and among different activities

Management of external loops does not require new constraints but requires the addition of a temporal condition in all constraints stating a condition like “an instance of type *Y* must be eventually created if an instance of type *X* is completed”. In those constraints we must ensure that the *Y* instance is created *after* the *X* instance is completed (earlier *Y* instances may exists due to previous loop iterations).

Standard loops may be regarded as an alternative representation for conditional sequence flows having the same activity as a source and destination. Therefore, the constraints needed to handle standard loop activities are similar to those required for conditional sequence flows:

- A constraint checking that the previous loop instance has finished
  context A inv: previous->forAll(status=‘completed’)

- A constraint stating that the loop condition is still true when starting the new iteration. Again, this is a creation-time constraint.
  context A inv: loopCondition=true

  where the *loopCondition* is taken from the properties of the activity as defined in the workflow model.

Moreover, we need also to check that the activity/ies at the end of the outcoming flows of the loop activity are not started until the loop condition becomes false. To prevent this wrong behavior we should treat all outgoing flows from the loop activity as conditional flows with

the condition *'not loopCondition'*. Then, the constraints generated to control the conditional flow will prevent next activity/ies to start until the condition *'not loopCondition'* becomes true.

Multi-instance loop activities are repeated a fixed number of times, as defined by the loop condition, which now is evaluated only once during the execution of the case and evaluates to a natural value instead of a boolean value. At the end of the case, the number of instances of the multi-instance activity must be an exact multiple of this value. Assuming that the multi-instance activity is called *A*, the OCL formalization of this constraint would be:

context Case inv: (activity->select(a| a.oclIsTypeOf(A))->size()) mod (loopCondition) = 0

For multi-instance loops the different instances may be created sequentially or in parallel. Besides, we can define when the workflow shall continue. It can be either after each single activity instance is executed (as in a normal sequence flow), after all iterations have been completed (similar to the *AND-merge* gateways), or as soon as a single iteration is completed (similar to the basic *OR-merge* gateway, where the system waits for the first incoming flow to arrive, but the other iterations do not generate additional sequence flows). Depending on each case, the constraints over the next activity are generated accordingly.

4.4.6. Event management

An event is something that "happens" during the course of the workflow execution. There are three main types of events: *Start*, *Intermediate* and *End*. Figure 4.4.4 shows an example of each type of event. A workflow schema may contain several start, intermediate, and end events.

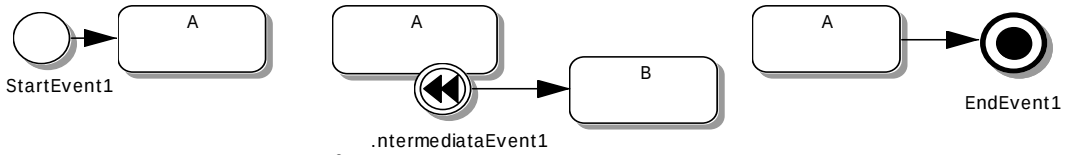


Figure 4.4.4 – Examples of events

Start events initiate a new flow, while end events indicate the termination of a flow. Intermediate events are instead used to change the normal flow. In particular, they can be used to show where messages are expected or sent within the process, to disrupt the normal flow through exception handling, or to show extra work required for compensating certain activities. Intermediate events can be attached to an activity (the triggering of the event aborts the activity execution) or can be placed in the middle of a sequence flow between two activities (the flow does not continue until the event is issued).

When a start event is issued, an instance of each activity connected to the event has to start afterwards. Reversely, no activity instance is created in a case before the occurrence of at least a start event. In particular, activity instances for activities connected only to flows coming

from one or more start events (as activity *A* in the previous figure) cannot be created until one of those start events is issued. The formalization of these constraints is the following:

- *context EventInstance inv: eventType.name='StartEvent1' and case.status='completed' implies case.activity->select(a|a.oclIsTypeOf(A) and a.eventInstance=self)->size()=1*
- *context Case inv: activity->notEmpty() implies event->exists(e| e.eventType.eventKind='StartEvent')*
- *context A inv: self.eventInstance->exists(ev| ev.eventType.name='StartEvent1')*

For end events defined as *terminate* end events [33] we must add a new constraint stating that no activity instances can be created in the case after the event has been issued. Assuming that *EndEvent1* (Figure 4.4.4) is defined as a terminate event, the following constraint must be added to the CS:

context EventInstance inv: eventType.name='EndEvent1' implies
case.activity->forAll (a| a.start< eventTime)

For intermediate events, the target activity of the event must be executed after the triggering of the event (and it cannot be executed otherwise). Depending on the kind of intermediate event, the interrupted activity will change its status to cancelled or aborted (which, for instance, may prevent the next activity in the normal sequence flow to be started). Given the previous *IntermediateEvent1* example, the following process constraints would be generated:

- *context EventInstance inv: eventType.name='IntermediateEvent1' and case.status='completed' implies case.activity->exists(a|a.oclIsTypeOf(B))*
- *context Case inv: activity->exists(a|a.oclIsTypeOf(B)) implies event-> exists(e| e.eventType.name='IntermediateEvent1')*

Obviously, this last constraint is true as long as *B* has no other incoming flows. Otherwise, all incoming flows form an implicit XOR-Merge over *B* and we should generate the constraints according to the pattern for XOR-Merge gateways.

4.4.7. Applying the translation patterns

As an example, Table 4.3 summarizes the result of applying the pattern translation over the workflow schema of Figure 4.2.1. For each activity (first column) we comment the generated process constraints. For sake of brevity, we do not specify here the set of constraints that describe the whole workflow. We provide in Table 4.4 the full definition of the constraints involved in the specification of one step of the workflow, namely the *Provide Quotation* activity. The rest of the specification can be found in the extended version of this proposal at [5].

Table 4.3 Process constraints for the workflow running example

Activity	Constraints
Ask Quotation	- When the activity instance comes after a *Provide Quotation*, the latter must have been completed (a single new ask quotation activity can be generated). Otherwise, it must have been created in response to the occurrence of a start event (due to the implicit XOR merge

	gateway corresponding to the two incoming arrows).
Provide Quotation	- A quotation cannot be provided until the *Ask Quotation* activity has finished. Moreover, if an instance of *Ask Quotation* is completed, a single *Provide Quotation* instance must eventually be created - After providing a quotation we can either ask for a new quotation or submit an order, but not both. At least one of them must be executed.
Submit Order	- The previous *Provide Quotation* activity must be completed. Besides, only a single *Submit Order* instance must be created for the same *Provided Quotation* instance - After submitting an order, both the *Choose Shipment* and the *Process OrderLine* activities must be executed
Choose Shipment	- The preceding *Submit Order* activity instance must be completed. Besides, a single *Choose Shipment* activity must be executed for each *Submit Order* activity instance
Process OrderLine	- The preceding *Submit Order* activity must be completed - The system must execute exactly as many *Process OrderLine* activity instances as the number of order (quotation) lines for the related order
Ship Order	- The order cannot be shipped until the shipment has been chosen and all order lines have been processed. In such a case, a *Ship Order* activity instance must be executed before ending the case
Pay Order	- An order cannot be paid until it has been shipped. Moreover, a single *pay order* activity shall be created in response to each order shipment

The *Provide Quotation* activity involves a set of constraints due to the sequence constraint with *Ask Quotation* activity, and a set due to the subsequent *XOR* split. These constraints are shown in Table 4.4.

Table 4.4 Constraint definition for the *Provide Quotation* activity

Constraints due to the sequence with *Ask Quotation*	The preceding activity must be of type *Ask Quotation* and must be completed
	context ProvideQuotation inv: previous->size()=1 and previous-> exists(a| a.oclIsTypeOf(AskQuotation) and a.status='completed')
	No two instances may be related with the same *Ask Quotation* instance
	context ProvideQuotation inv: ProvideQuotation.allInstances()-> isUnique(previous)
	A Provide Quotation instance must exist for each completed *Ask Quotation*
	context Case inv: status='completed' implies activity-> select(a| a.oclIsTypeOf(AskQuotation) and a.status='completed')-> forAll(a|a.next->exists(b|b.oclIsTypeOf(ProvideQuotation) and a.end<=b.start) and a.next->size()=1)
Constraints due to the *XOR split*	The next activity must be either another *Ask Quotation* instance or a *Submit Order* instance, but not both
	context ProvideQuotation inv: next->select(a| a.oclIsTypeOf(AskQuotation) or

	a.oclIsTypeOf(ProvideQuotation))->size()<=1
	If the *Provide Quotation* instance is completed, an *Ask Quotation* or a *Submit Order* must be created before ending the case.
	context Case inv: status='completed' implies activity-> select(a| a.oclIsTypeOf(ProvideQuotation) and a.status='completed')-> forAll (a| a.next-> exists(b| b.oclIsTypeOf(AskQuotation) or b.oclIsTypeOf(SubmitOrder)))
	Only *Ask Quotation* activity instances or *Submit Order* instances may follow a *Provide Quotation* instance
	context ProvideQuotation inv: next->forAll(b| b.oclIsTypeOf(AskQuotation) or b.oclIsTypeOf(SubmitOrder)

4.5. Implementation of the workflow-extended CS

A workflow-extended CS is a completely standard CS. No new modeling primitives have been created to express the extension of the original CS with the required workflow information. Therefore, any method or tool able to provide an automatic implementation of the initial CS can also cope with the automatic generation of our workflow-extended CS in any final technology platform.

The application of my method for OCL constraints incremental integrity checking (see section 2) can be applied to these constraints to generate their efficient implementation.

```
create trigger AskProvideSeqConstraint
before insert on AskProvideRelationship
for each row
Declare v_Status Varchar(10);
        EInvalidActivity Exception;
Begin
   SELECT status into v_Status
   FROM AskQuotation a
    WHERE a.id = :new.askActivity_id;
    If (v_Status<>'completed')
       then raise EInvalidActivity; end if;
End;
```

(i)

```
void AssignPreviousActivity(AskQuotation a)
throws Exception
{
   if (! a.status.equals("completed"))
        throw new Exception("Invalid Activity");
   else previous.add(a);
}
```

(ii)

Figure 4.5.1. Examples of a sequence constraint implemented in particular technologies

As an example, Figure 4.5.1 shows a possible implementation of the sequence constraint between activities *Ask quotation* and *Process quotation* when the CS is implemented in *(i)* a relational database and *(ii)* with an object-oriented technology. In the former case, the constraint may be implemented as a trigger over the table *AskProvideRelationship* representing the *Precedes* relationship type between both activities. In the latter case, the

constraint is verified as part of the method *AssignPreviousActivity* redefined in the *ProvideQuotation* class, which represents the *ProvideQuotation* entity type defined in the CS. The translation from OCL into SQL or Java has already been addressed in several works ([12],[14], [26]).

For Web applications, an interesting alternative is to derive an initial hypertext model from the workflow-extended CS so that the hypertext structure enforces some of the process constraints among activities assigned to the same user (or group of users) by means of driving the user navigation through the Web site. This can be done by designing in the proper way the set of pages and links that can be browsed. For instance, assuming a sequence constraint between activities *A* and *B*, performed by the same user, Figure 4.5.2 shows a hypertext model that from the home page forces the user to go through the Web pages implementing *A* before starting *B*. The hypertext model is defined in WebML [12], a conceptual language for the specification of Web applications, already extended with workflow-specific primitives [4]. The operation units *StartActivity* and *EndActivity* are in charge of recording the information about the activities' progress in the corresponding entity types of the conceptual schema. More complicated constraints appearing in the workflow-extended CS can be enforced by means of appropriate branching and task assignment primitives.

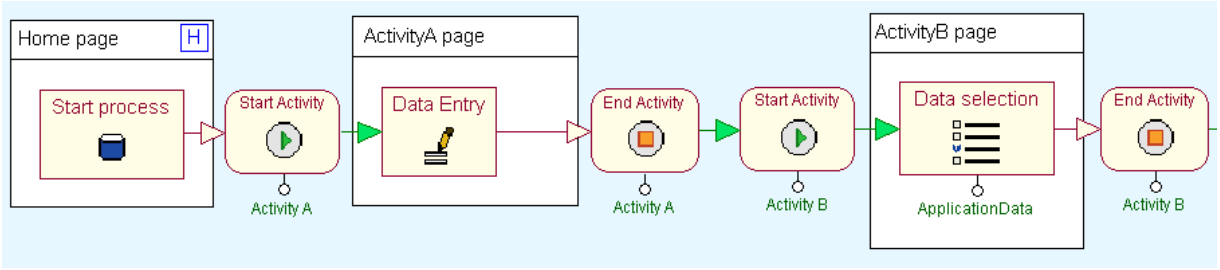


Figure 4.5.2. Example of a sequence constraint implemented
within the hypertext model of a Web application

4.6. Summary

In this section we have presented an automatic approach to integrate the semantics of business process specifications within conceptual schemas.

The main contribution of this work is the demonstration of the applicability of the conceptual modelling approach to the specification (and implementation) of workflow-based applications. We described how to build a workflow-extended conceptual schema by means of extending the domain conceptual schema with *(i)* the definition of a set of new entity and relationship types for workflow status tracking; and *(ii)* the rules for generating the integrity constraints on such types, needed for enforcing the business process specification. Once the extended conceptual schema is generated, it is possible to apply the usual model-driven development methods to generate an implementation of the system.

To make the proposed approach viable, we have developed a visual editor prototype (a sample screenshot can be seen in Figure 4.2.1) that allows to design BPMN diagrams and to automatically generate the corresponding workflow subschema, according to the guidelines presented in this section. The actual automatic generation is performed by means of XSLT transformations that apply to an XML representation of the workflow model. The result is, then, added to the XMI file [29] representing the domain subschema to finally generate the workflow-extended conceptual schema.

Future work will include the extension of our translation patterns to cover the full expressivity of the BPMN notation and the study and comparison of different implementation options for the workflow-extended conceptual schemas depending on the application requirements. This is especially true for process constraints, which may benefit from alternative implementation techniques.

5. Results of the research stay

As a (partial) consequence of my research stay I would like to remark the following achievements:

1 – Ph.D. in Computer Science for the Universitat Politècnica de Catalunya, obtained in November 2006 (cum laude by unanimity) with European mention (a research stay was a mandatory requirement to get this mention).

2 – Published paper: Constraint tuning and management for web applications (coauthor: Marco Brambilla from the Politecnico di Milano) presented at the 6th International Conference on Web Engineering.

3 – Funded project: Spanish-Italian Integrated action. Principal Investigator: Ernest Teniente. Funded by the Spanish Ministry of Science and Technology and the Italian Government. This project will allow me to continue my joint research with the group from Milan during the years 2007 and 2008. The funding covers travel expenses and meeting organizations among the Catalan and the Italian group. The proposal for this integrated action was submitted during my research stay and based on the preliminary results obtained so far.

References

1. Aalst, W. M. P. v. d., Kumar, A.: A reference model for team-enabled workflow management systems. Data & Knowledge Engineering 38 (2001) 335-363
2. Bae, J., Bae, H., Kang, S.-H., Kim, Y.: Automatic Control of Workflow Processes Using ECA Rules. IEEE Transactions on Knowledge and Data Engineering 16 (2004) 1010-1023
3. Baresi, L., Colazzo, S., Mainetti, L.: First Experiences on Constraining Consistency and Adaptivity of W2000 Models. In: Proc. SAC, (2005)
4. Brambilla, M.: Extending Hypertext Conceptual Models with Process-Oriented Primitives. In: Proc. 22nd Int. Conf. on Conceptual Modeling (ER'03), LNCS, 2813 (2003) 246-262
5. Brambilla, M., Cabot, J., Comai, S. Automatic Generation of Worfklow-extended Conceptual Schemas (extended version).[Online]. Available: http://www.lsi.upc.edu/~jcabot/research/workflow/
6. Brambilla, M., Deutsch, A., Sui, L., Vianu, V.: The Role of Visual Tools in a Web Application Design and Verification Framework: a Visual Notation for LTL Formulae. In: Proc. 5th Int. Conf. in Web Engineering (ICWE'05), LNCS, 3579 (2005) 557-568
7. Brambilla, M., Ceri, S., Fraternali, P., Manolescu, I.: Process Modeling in Web Applications. ACM Transactions on Software Engineering and Methodology in print (2006)
8. Cabot, J.: Incremental Integrity Checking in UML/OCL Conceptual Schemas. PhD Thesis. LSI Department. Technical University of Catalonia (2006)
9. 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
10. Ceri, S., Widom, J.: Deriving Production Rules for Constraint Maintenance. In: Proc. 16th Int. Conf. on Very Large Databases (VLDB'90), (1990) 566-577
11. Ceri, S., Daniel, F., Demaldé, V., Facca, F. M.: An Approach to User-Behavior-Aware Web Applications. In: Proc. ICWE, LNCS, 3579 (2005) 417-428
12. Ceri, S., Fraternali, P., Bongio, A., Brambilla, M., Comai, S., Matera, M.: Designing Data-Intensive Web Applications. Morgan Kaufmann (2002)
13. Combi, C., Pozzi, G.: Temporal Conceptual Modelling of Workflows. In: Proc. 22nd Int. Conference on Conceptual Modeling (ER'03), LNCS, 2813 (2003) 59-76
14. 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 2001), LNCS, 2185 (2001) 104-117
15. Domingos, D., Rito-Silva, A., Veiga, P.: Workflow Access Control from a Business Perspective. In: Proc. ICEIS, vol. 3 (2004) 18-25
16. Embley, D. W., Barry, D. K., Woodfield, S.: Object-Oriented Systems Analysis. A Model-Driven Approach. Yourdon Press Computing Series. Yourdon (1992)
17. Fernandez, M. F., Florescu, D., Levy, A. Y., Suciu, D.: Declarative Specification of Web Sites with Strudel. VLDB Journal 9 (2000) 38-55
18. 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
19. Garrigós, I., Gómez, J., Cachero, C.: Modelling Dynamic Personalization in Web Applications. In: Proc. ICWE, (2003) 472-475

20. 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
21. IBM: MQSeries Workflow.
22. ISO/TC97/SC5/WG3: Concepts and Terminology for the Conceptual Schema and Information Base. ISO, (1982)
23. Knapp, A., Koch, N., Zhang, G., Hassler, H.: Modeling Business Processes in Web Applications with ArgoUWE. In: Proc. UML 2004, LNCS, 3273 (2004), 69-83
24. Mellor, S. J., Balcer, M. J.: Executable UML. Object Technology Series. Addison-Wesley (2002)
25. Melton, J., Simon, A. R.: SQL:1999, Understanding Relational Language Components. Morgan Kaufmann (2002)
26. Objecten, K.: Octopus: OCL Tool for Precise Uml Specifications. (2005)
27. 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
28. 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)
29. OMG: XML Metadata Interchange (XMI) Specification. OMG Adopted Specification (formal/03-05-02) (2002)
30. OMG: UML 2.0 OCL Specification. OMG Adopted Specification (ptc/03-10-14) (2003)
31. OMG: UML 2.0 Superstructure Specification. OMG Adopted Specification (ptc/03-08-02) (2003)
32. OMG: MDA Guide Version 1.0.1. (2003)
33. OMG/BPMI: Business Process Management Notation v.1. OMG Adopted Specification (2006)
34. Oracle: Workflow 11i.
35. 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
36. Schwabe, D., Rossi, G.: The Object-Oriented Hypermedia Design Model. Communications of the ACM 38 (1995) 45-46
37. Teichroew, D., Sayani, H.: Automation of System Building. Datamation 17 (1971) 25-30
38. Türker, C., Gertz, M.: Semantic integrity support in SQL:1999 and commercial (object-)relational database management systems. The VLDB Journal 10 (2001) 241-269
39. WfMC. Workflow Management Coalition.[Online]. Available: www.wfmc.org