UNIVERSITAT POLITÈCNICA DE CATALUNYA
DEPARTAMENT DE LLENGUATGES I SISTEMES INFORMÀTICS

ANNA QUERALT CALAFAT

VALIDATION OF UML CONCEPTUAL SCHEMAS WITH OCL CONSTRAINTS AND OPERATIONS

PhD. THESIS
ADVISED BY DR. ERNEST TENIENTE i LÓPEZ

BARCELONA
2009

A thesis presented by Anna Queralt Calafat
in partial fulfillment of the requirements for the degree of
Doctor en Informàtica per la Universitat Politècnica de Catalunya

A l’Òscar i la Júlia

Acknowledgements

I would like to thank Ernest Teniente for his exigency and rigor, and for the confidence placed in me. I am very grateful for all the things I have learnt from him.

Thanks to Antoni Olivé and my colleagues in the *Grup de Modelització Conceptual* and the *Secció de Sistemes d'Informació* for their friendship and support.

I would also like to thank the examiners of the thesis board: Prof. Antoni Olivé, Dr. Dolors Costal, Prof. Bernhard Thalheim, Prof. Stefano Ceri and Prof. Oscar Pastor. It is an honor for me that they accepted to be members of this panel.

I am grateful to Guillem Lubary for his effort in implementing the method presented in this thesis. I would also like to thank Albert Tort, Elena Planas and Antonio Villegas for being always ready to lend us a hand.

Finally, I thank my family for their support and for their hope of seeing this work completed. Specially, I thank Òscar for everything.

This work has been partly supported by Spain's Ministerio de Ciencia y Tecnología under projects TIN2005-06053 and TIN2008-00444.

Abstract

To ensure the quality of an information system, it is essential that the conceptual schema that represents the knowledge about its domain and the functions it has to perform is semantically correct.

The correctness of a conceptual schema can be seen from two different perspectives. On the one hand, from the point of view of its definition, determining the correctness of a conceptual schema consists in answering to the question "*Is the conceptual schema right?*". This can be achieved by determining whether the schema fulfills certain properties, such as satisfiability, non-redundancy or operation executability.

On the other hand, from the perspective of the requirements that the information system should satisfy, not only the conceptual schema *must be right*, but it also *must be the right one*. To ensure this, the designer must be provided with some kind of help and guidance during the validation process, so that he is able to understand the exact meaning of the schema and see whether it corresponds to the requirements to be formalized.

In this thesis we provide an approach which improves the results of previous proposals that address the validation of a UML conceptual schema, with its constraints and operations formalized in OCL. Our approach allows to validate the conceptual schema both from the point of view of its definition and of its correspondence to the requirements.

The validation is performed by means of a set of tests that are applied to the schema, including automatically generated tests and ad-hoc tests defined by the designer. All the validation tests are formalized in such a way that they can be treated uniformly, regardless the specific property they allow to test.

Our approach can be either applied to a complete conceptual schema or only to its structural part. In case that only the structural part is validated, we provide a set of conditions to determine whether any validation test performed on the schema will terminate. For those cases in which these conditions of termination are satisfied, we also provide a reasoning procedure that takes advantage of this situation and works more efficiently than in the general case. This approach allows the validation of very expressive schemas and ensures completeness and decidability at the same time.

To show the feasibility of our approach, we have implemented the complete validation process for the structural part of a conceptual schema. Additionally, for the validation of a conceptual schema with a behavioral part, the reasoning procedure has been implemented as an extension of an existing method.

Contents

1 Introduction

An information system performs three main functions (Boman, Bubenko et al. 1997):

- Memory: To maintain a consistent representation of the state of a domain.
- Informative: To provide information about the state of a domain.
- Active: To perform actions that change the state of a domain.

In order to perform its functions, an information system needs general knowledge about its domain, as well as knowledge about the functions it has to perform. In the information systems field, this knowledge is called conceptual schema (Olivé 2007).

The purpose of conceptual modeling is to determine and formally define the conceptual schema of an information system, which must include all relevant static and dynamic aspects of its domain (ISO/TC97/SC5/WG3 1982). The part of a conceptual schema that deals with static aspects is called the structural schema, and the part that deals with dynamic aspects is called the behavioral schema.

The structural schema defines the state of the domain that must be represented in order that the system performs the memory function. A structural schema consists of a taxonomy of entity types together with their attributes; a taxonomy of relationship types among entity types; and a set of integrity constraints over the state of the domain, which define conditions that each state of the information base must satisfy.

An information system maintains a representation of the state of a domain in its information base (IB). The state of the IB is the set of instances of the entity and relationship types defined in the conceptual schema. The integrity constraints of the structural schema guarantee the consistency of the IB.

The content of the IB changes due to the execution of operations. A behavioral schema contains a set of operations and the definition of their effect on the IB, that is, the changes they make on the IB when they are executed. This knowledge is usually defined by the preconditions and postconditions of the operations. A precondition expresses a condition that must be satisfied when the call to the operation is done. A postcondition expresses a condition that the new state of the information base must satisfy. The execution of an operation results in a set of one or more structural events to be applied to the IB. Structural events are elementary changes on the content of the information base, that is, insertions or deletions of instances.

1.1 Conceptual Modeling in UML

In the last years, the UML (Unified Modeling Language) (OMG 2007) has become a de facto standard in conceptual modeling.

This language provides several diagrams for the different stages of information systems development, and only some of them are useful to define a conceptual schema. In particular, the analysis class diagram is used to define the entity types and relationship types of the structural schema (called classes and associations in UML, respectively), together with some integrity constraints that can be expressed graphically.

A complete conceptual schema must include the definition of all relevant integrity constraints (ISO/TC97/SC5/WG3 1982). Then, those constraints that cannot be expressed graphically, must also be expressed in UML by means of any general-purpose language. According to (Warmer and Kleppe 2003), we assume they are specified in the OCL (Object Constraint Language) (OMG 2006).

Operation contracts are the basic component of the behavioral schema, since they are the ones that specify which are the effects of operations, and the conditions needed so that the operation can be executed.

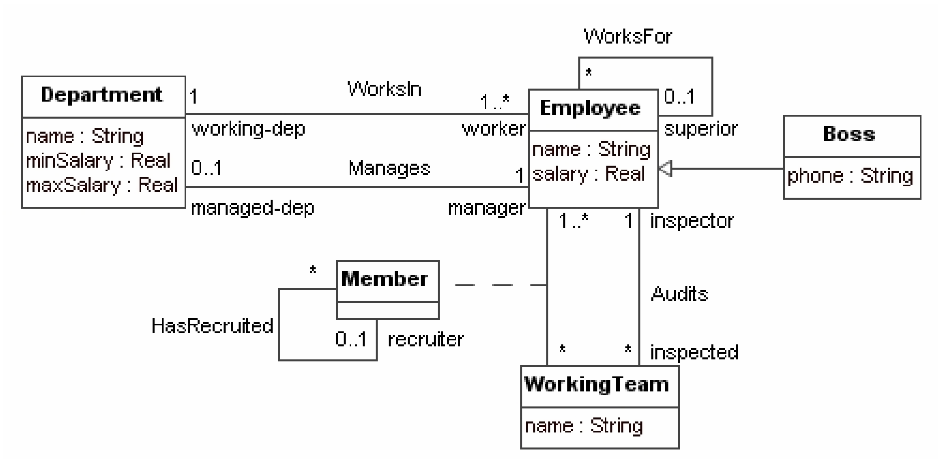


Fig. 1. A structural schema for the domain of Employees and their assignment to Departments

```
Integrity constraints

1.    context Department inv UniqueDep:
      Department.allInstances()->isUnique(name)

2.    context Employee inv UniqueEmp:
      Employee.allInstances()->isUnique(name)

3.    context WorkingTeam inv UniqueTeam:
      WorkingTeam.allInstances()->isUnique(name)

4.    context Department inv MinimumSalary:
      self.minSalary > 1000

5.    context Department inv CorrectSalaries:
      self.minSalary < self.maxSalary

6.    context Department inv ManagerIsWorker:
      self.worker->includes(self.manager)

7.    context Department inv ManagerHasNoSuperior:
      self.manager.superior->isEmpty()

8.    context Boss inv BossIsManager:
      self.managed-dep->notEmpty()

9.    context Boss inv BossHasNoSuperior:
      self.superior->isEmpty()

10.   context Boss inv SuperiorOfAllWorkers:
      self.employee->includesAll(self.managed-dep.worker)

11.   context WorkingTeam inv InspectorNotMember:
      self.employee->excludes(self.inspector)

12.   context Member inv NotSelfRecruited:
      self.recruiter<>self

13.   context WorkingTeam inv OneRecruited:
      self.member->exists(m|m.recruiter.workingTeam=self)
```

Fig. 2. Textual integrity constraints of the schema in Figure 1, expressed in OCL.

Operation contracts consist of a precondition and a postcondition, that can be expressed in OCL. OCL is a declarative language and, as such, it expresses conditions that some state of the IB must satisfy. In particular, preconditions specify the conditions that the IB must fulfill so that the operation can be executed, and postconditions describe the state of the IB after the effect of the operation is applied.

Figure 1, 2 and 3 show a complete UML conceptual schema, which we will use throughout this document. Textual integrity constraints and operation contracts are specified in OCL.

In Figure 1 we show its structural part. It consists of a UML class diagram with five classes (*Department, Employee,* its subclass *Boss, WorkingTeam* and the association class *Member*), together with their attributes, and five associations (*WorksIn*, *Manages, WorksFor, Audits* and, again, *Member*). Moreover, there are some graphical cardinality constraints specified (for instance, the cardinality 1 of *Department* states that an employee must work exactly in one department).

Operation that creates a new department with name *d-name, minSal* and *maxSal*

```
Op:     newDept(d-name: String, minSal, maxSal: Real, managerName:
        String, managerSal: Real)
Pre:
Post:   Department.allInstances()-> exists(d | d.oclIsNew() and
        d.name=d-name and minSalary=minSal and maxSalary=maxSal
        and Employee.allInstances()->exists(e | e.oclIsNew() and
        e.name=managerName and e.salary=managerSal and
        d.manager=e))
```

Operation that removes the department *dep*

```
Op:     removeDept(dep: Department)
Pre:
Post:   Department.allInstances()->excludes(dep)
```

Operation that creates a new employee and assigns him/her to the department *dep*

```
Op:     hire(e-name: String, sal: Real, dep: Department)
Pre:
Post:   Employee.allInstances()-> exists(e | e.oclIsNew() and
        e.name=e-name and e.salary=sal and e.working-dep=dep)
```

Operation that classifies the employee *emp* as an instance of *Boss,* in case he is not a manager

```
Op:     promote(emp: Employee, phone: String)
Pre:    emp.managed-dep->isEmpty()
Post:   emp.oclIsTypeOf(Boss) and emp.oclAsType(Boss).phone=phone
```

Operation that removes the employee *emp,* as long as he is not assigned to a department

```
Op:     fire(emp: Employee)
Pre:    emp.working-dep->isEmpty()
Post:   Employee.allInstances()->excludes(emp)
```

Operation that, if some department exists, creates a new working team and associates it to the inspector *insp*

```
Op:     newTeam(t-name: String, insp: Employee)
Pre:    Department.allInstances()->notEmpty()
Post:   WorkingTeam.allInstances()-> exists(t | t.oclIsNew()
        and t.name = t-name and t.inspector = insp)
```

Operation that makes the employee *emp,* optionally recruited by the member *rec,* become a member of the working team *team*

```
Op:     newMember(emp:Employee,team:WorkingTeam,rec:Member)
Pre:
Post:   Member.allInstances()->exists(m|m.oclIsNew() and
        m.employee=emp and m.workingTeam=team and m.recruiter=rec)
```

Fig. 3. The behavioral schema corresponding to the structural schema of Figures 1 and 2

The OCL constraints in Figure 2 provide the class diagram with additional semantics. There are three key constraints (*UniqueDep, UniqueEmp* and *UniqueTeam*), one for each class. The constraint *MinimumSalary* states that the minimum salary to be paid to all the employees working in the department must be greater than 1000, and *CorrectSalaries* guarantees that the minimum salary of a department is lower than its maximum salary. The constraint *ManagerIsWorker* states that the manager of a department must be one of its workers. Note that this constraint could be expressed graphically by means of a *subset* (see chapter 2). Constraint *ManagerHasNoSuperior* guarantees that the manager of a department does not work for any other employee. The constraint *BossIsManager* guarantees that a boss is the manager of some department. The next constraint, *BossHasNoSuperior*, states that a boss does not work for any other employee. The constraint *SuperiorOfAllWorkers* states that the workers of a department managed by a boss must work for that boss. The next one, *InspectorNotMember* ensures that an employee cannot audit a working team if he is one of its members. The constraint *NotSelfRecruited* prevents a member of a working team from recruiting himself and, finally, *OneRecruited* guarantees that each working team has at least one member recruited by another member of the same team.

The behavioral part of this conceptual schema can be found in Figure 3. It includes seven operation contracts that specify the only changes that can be performed on the information base. As can be drawn from its name, the operation *newDept* creates an instance of Department with its corresponding manager, and *removeDept* deletes the indicated instance of *Department*. The operation *hire* creates an employee and assigns it to a department, the operation *promote* classifies an employee as an instance of *Boss*, and the operation *fire* removes an employee. There are also two operations about working teams, *newTeam,* that creates an instance of *WorkingTeam,* with its corresponding *inspector*, and *newMember*, which assigns an employee to a working team.

1.2 The Need for Validation

The conceptual schema shown in Figures 1, 2 and 3 is syntactically correct according to the UML and OCL metamodels. However, this does not ensure that it can be successfully populated, let alone that it correctly represents the intended domain. In order to guarantee the semantic correctness of a schema and, thus, to ensure the quality of the final application, it is necessary to provide the designer with some kind of help to answer questions like: can all the classes and associations of the schema be populated? are there any redundant constraints? are any constraints missing? is it possible to execute all the operations defined? is the schema actually representing the information needed? does it include all the operations required?

Quality problems of information systems have mostly been addressed in the later phases of the development process. In particular, several validation and verification techniques such as testing, debugging and program verification have been proposed.

However, while it is true that quality can be understood in terms of usability, reliability or efficiency, and these properties can mostly be determined by means of the final product, there are other quality factors that do not only depend on the implementation. In fact, the quality of an information system is largely determined early in the development cycle, i.e. during requirements specification and conceptual modeling. Moreover, errors introduced at these stages are usually much more expensive to correct than errors introduced during design or implementation. Thus, it is desirable to prevent, detect and correct errors as early as possible in the development process, by assessing the correctness of the conceptual schemas built. In fact, this goal has been included in the research agenda to achieve conceptual-schema centric development, a grand challenge of information systems research (Olivé 2005). In particular, it should be possible to test and verify conceptual schemas at least to the same extent that has been achieved in software. This is especially important nowadays, given that the process of transforming a conceptual schema into a design or implementation can be, at least partially, automated.

The correctness of a conceptual schema can be seen from two different points of view. From an internal point of view, determining the correctness of a conceptual schema corresponds to answering to the question "Is the conceptual schema right?" There are some typical properties that can be automatically tested to determine this kind of correctness, such as satisfiability of the schema, liveliness of its classes and associations, non-redundancy of its constraints or executability of its operations.

On the other hand, from an external point of view, correctness refers to the accuracy of the conceptual schema regarding the user requirements (Adrion, Branstad et al. 1982), and it corresponds to determining whether "Is it the right conceptual schema?". Testing whether a schema is correct in this sense may not be completely automated, since it necessarily requires the intervention of the designer. Nevertheless, it is desirable to provide the designer with a set of tools that assist him during the validation process (Bubenko 1986).

Due to the high expressiveness of the combination of the UML and OCL languages, checking the (internal or external) correctness of a UML conceptual schema manually becomes a very difficult task, especially when the set of textual constraints is large. For this reason, it is desirable to support the designer in validating a conceptual schema.

Several proposals deal with the validation of the structural part of ER and UML conceptual schemas, determining for example their satisfiability (Brucker and Wolff 2006; Hartmann 2001; Lenzerini and Nobili 1987) or allowing the designer to check if a given state is consistent according to the constraints defined in the schema (Gogolla, Bohling et al. 2005). However there are a few proposals that take the behavioral schema into account in the validation process (Costal, Teniente et al. 1996; Díaz, Paton et al. 1998; Formica and Frank 2002; Leuschel and Butler 2008), none of them dealing with UML schemas with general OCL constraints and operations. Additionally, none of the existing methods is able to check both the internal and the external correctness of a schema.

The goal of this thesis is to validate a UML conceptual schema, with general constraints and operations formalized in OCL, from both points of view. In particular, we provide the designer both with a set of tests that are automatically drawn from the schema in order to check its internal correctness, and with the ability to perform general user-defined validation tests to assess its external correctness. Both kinds of correctness can be tested either on the structural part or taking into account also the behavioral part.

1.3 Reasoning on the Structural Schema

In this section we exemplify the kind of tests that can be performed on the structural part of a conceptual schema to determine its correctness. In section 1.3.1 we explain the kinds of reasoning that can be performed on the definition of the schema, while in section 1.3.2 we will see how reasoning can be used to validate that the schema satisfies the user requirements.

1.3.1 Is the Structural Schema Right?

Correctness of a conceptual schema can be understood from de point of view of its structure, regardless the user requirements. This kind of correctness can be determined by reasoning on the definition of the conceptual schema, without taking the domain or the requirements of the system into account. For this reason, the question of whether a schema is right, i.e. correctly defined, can be answered in a completely automatic way by performing several tests, which are, in turn, automatically drawn from the schema without the intervention of the designer. That is, the process of testing the internal correctness of the schema is completely automated, since the tests are automatically defined and executed, and the answers provided directly determine the correctness without having to be interpreted by the designer. We may note, however, that when some error is found, the designer must decide how it must be fixed, since several valid solutions may exist.

We will illustrate our explanations by means of sample states of the IB corresponding to the schema in Figures 1 and 2, denoted by sets of instances of the classes and associations of the schema. For example, an employee named Peter will be represented by *employee(peter)*. Similarly, the fact that Peter works in the sales department will be denoted by *worksIn(peter, sales)*, which represents an instance of the association *WorksIn*. For the sake of clarity, we omit the attributes in those tests that are not affected by them.

There are some typical properties that can be checked on a schema but, as will be seen in section 1.5.1, the one mostly approached has been satisfiability, also known as consistency in the literature.

A schema is satisfiable if there is a non-empty state of the IB in which all its integrity constraints are satisfied.

For instance, consider the structural schema of Figure 1. This schema can be populated with an instance of *Employee*, who necessarily has to work in a *Department* according to the cardinality constraint 1 in the role *working-dep* of the association *WorksIn*. Additionally, according to the constraint *ManagerIsWorker*, the department must have a manager that is one of its workers. There is no need to populate the association *WorksFor* or the class *WorkingTeam*, since they are not required in order to have a valid instance of *Employee.* Thus, a sample state proving satisfiability is one in which an employee works in a department and he is its manager at the same time: *{employee(john), worksIn(john,sales), department(sales), manages(john,sales)}*. Thus, this schema is satisfiable.

However, this does not mean that the schema is completely correct, since there may be some classes or associations that are never populated in any of the valid instances of the schema. This will be discussed in chapter 3.

1.3.2 Is It the Right Structural Schema?

From an external point of view, correctness refers to the correspondence of the knowledge represented in the schema with the user requirements. This can be determined by means of validation techniques, which aim at checking whether the schema properly reflects what the user needs from the application to be developed. Rather than using informal techniques, such as building a prototype which shows the behavior of the application, a better option is for example to animate the specification itself and see whether the results of the animation correspond to the ones expected.

As an example, although the schema of Figures 1 and 2 is satisfiable, it may not specify what the designer intended. For instance, the designer may be interested to know whether the schema ensures that the salary of all the employees is greater than the minimum salary of their corresponding departments. The following state, which is admitted by the previous schema, shows that this is not the case: {*employee(mary, 500), worksIn(mary, iT), manages(mary, iT), department(iT, 1000, 1500)*} .

The reason is that, although there are constraints that ensure the correctness of the values of the attributes *minSalary* and *maxSalary* of *Department*, a constraint that relates the attribute *salary* of *Employee* with those of *Department* is missing. The designer may not realize of this mistake by testing the internal correctness of the schema, since this flaw has to do with its correspondence to the domain.

This kind of tests are the ones that help the designer to guarantee that the schema really specifies what he or she intended, so the correctness of the schema cannot be automatically determined from the result obtained. The designer will determine it once he has seen the result of the tests, by checking whether they correspond with his knowledge about the domain.

1.4 Reasoning in the Presence of Operations

In this section we explain the reasoning tasks that can be performed in the presence of operations and how they affect the properties satisfied by the structural schema. We illustrate how the importance of taking the behavioral schema into account in the validation.

The *behavioral schema* contains a set of *system operations* and the definition of their effect on the IB. System operations specify the response of the system to the occurrence of some event in the domain, viewing the system as a black box, and they define the only changes that can be performed on the IB (Larman 2004).

An operation is defined by means of a *precondition*, which expresses a condition that must be satisfied when the call to the operation is done, and a *postcondition,* which expresses a condition that the new state of the IB must satisfy.

The operation contracts in Figure 3 belong to the behavioral schema corresponding to the structural schema in Figures 1 and 2. Each contract describes the changes that occur in the IB when the operation is invoked. According to the strict interpretation of operation contracts, which is explained in chapter 5, preconditions need not be responsible for guaranteeing the satisfaction of integrity constraints. In this sense, it is assumed that constraints are checked at the end of each execution and the operation is rejected in case some constraint is violated. This is why possible violations of integrity constraints are not prevented in the preconditions of our contracts.

Reasoning on the structural schema of Figure 1 alone, it can be determined that it is satisfiable, as we have seen in section 1.3.1. For instance, the following state of the IB in which an employee works in a department and he is its manager at the same time proves the satisfiability of the class diagram, with its graphical and OCL constraints: *{employee(john), worksIn(john,sales), department(sales), manages(john,sales)}*.

However, the fact that the structural part of a conceptual schema is correct does not necessarily imply that the whole conceptual schema also is. That is, when we take into account that the only changes admitted are those specified in the operations of the behavioral schema, it may happen that the properties fulfilled by the structural schema alone are no longer satisfied.

In our example, although it is possible to find instances of *Department* satisfying all the constraints as we have just seen, there is no operation that successfully populates this class. The operation *newDept* seems to have this purpose, but it never succeeds since it does not associate the new department with an *Employee* by means of the association *WorksIn*, which violates the minimum cardinality of the role *worker*. As a consequence, since an *Employee* must be assigned to one *Department* according to the cardinality constraint 1..* in the role *working-dep* of *WorksIn*, there can not exist any instance of *Employee* either. Then, we have that this schema can never be populated using the operations defined and, although the structural part of the schema is semantically correct, the complete conceptual schema is not.

Also, when dealing with operations, additional validation tests can be performed, namely applicability and executability of each operation (Costal, Teniente et al. 1996). To illustrate these properties, let us consider the operation *fire* of Figure 3, which deletes the indicated *Employee.* As can be seen, the precondition of this operation requires the existence of at least an employee that does not work in any department, which is not possible according to the cardinality constraint 1 of *working-dep.* This means that this operation is not applicable, that is, it can never be executed because it is impossible to satisfy its precondition. Thus, in our example, the designer should avoid this erroneous situation by, for instance, removing the precondition or changing the cardinality constraint of *working-dep* to 0..1.

Although an operation is applicable, it may never be successfully executed because it always leaves the IB in an inconsistent state. For instance, let us consider the following additional operation, which removes the assignment of an employee to his department:

```
Op:     deassign(emp: Employee)
Pre:
Post:   emp.working-dep.worker->excludes(emp)
```

This operation is applicable by definition, since its empty precondition is always satisfied. However, the postcondition removes the instance of the association *WorksIn* between the employee *emp* and the only department to which *emp* is assigned. Since this operation neither replaces the assignment of the employee by creating a new instance of *WorksIn* linking *emp* to another department, nor deletes the employee, the result is that, after the execution of this operation, there is always an instance of employee that is not assigned to any department, violating the cardinality constraint 1 of *working-dep.* This means that the execution of this operation will always be rejected because it is impossible to satisfy its postcondition and the integrity constraints at the same time. This is another undesirable situation that the designer should avoid by modifying either the structural schema (changing the cardinality constraint of *working-dep* to 0..1) or the behavioral one (linking the employee to another department or deleting the employee).

1.5 Previous Work

1.5.1 On Validating the Structural Schema

In this section we review how correctness has been addressed in conceptual modeling, regarding the validation of the structural part of a conceptual schema. We have classified the most relevant approaches that do not deal with the behavioral schema into two categories. The first one includes methods and tools to reason on conceptual schemas that are not specified in UML (ER schemas, deductive databases, logic), whereas the second group includes those methods and tools that approach correctness in UML conceptual schemas. As will be seen, none of the methods is able to determine external correctness, since all of them deal with typical reasoning tasks such as satisfiability. Additionally, only the last method commented considers general constraints, but it renounces to completeness.

Non-UML Approaches

Some reasoning tasks have been addressed in ER conceptual schemas, the most popular one being satisfiability of cardinality constraints.

Usually, methods for ER schemas report whether the structural part of the schema is satisfiable, and use a more restrictive notion of satisfiability. Classically, satisfiability of a schema admits that a schema is satisfiable if there exists an instance of the schema (possibly empty) satisfying all integrity constraints. However, this is not sufficient for cardinality constraints and, in particular, for databases, and introduce the notion of strong satisfiability, which ensures that there is at least one fully populated instance of the schema satisfying all the constraints. Strong satisfiability was introduced in (Lenzerini and Nobili 1987), and their approach to determining strong satisfiability consists in reducing the problem to solving a linear inequality system. This system is defined from the set of relationships and cardinality constraints of the schema. Then, a schema is strongly satisfiable if and only if there are solutions for its associated inequality system. An application of these results to UML schemas is shown in (Maraee and Balaban 2007).

The same problem is addressed in (Hartmann 1998), which determines strong satisfiability of a schema by means of a graph-theoretic approach. This time, the kind of constraints considered are int-cardinality constraints (Thalheim 2000), which are more general than traditional ones, since they allow gaps in the sets of permitted cardinalities.

The same method is used in (Hartmann 2001), but this time it serves more specific purposes. Given a cardinality constraint set S, the method can find superfluous entities, i.e. entities whose population is empty in every instance of the schema satisfying S, and determine which is the minimal subset of constraints that causes a schema not to be satisfiable.

Another problem is approached in (Bowers 2003), which is the detection of potentially redundant associations in an ER schema. The method is based this time in adjacency matrixes, but is expensive and incomplete, since some types of redundancy involving more than one relationship between two entities cannot be detected.

Finally, (Vigna 2004) presents an approach in the context of web-based database administration systems, where the only changes that can be applied to an IB are local modifications: insertions/deletions of an entity and all its relationships, or insertions/deletions of a single relationship. This work determines whether all the instances of a schema are mutually reachable using only the local modifications mentioned. In case they are all mutually unreachable, then it means that the schema is incorrect for his purposes.

In the context of Object-Oriented schemas, (Bekaert, Van Nuffelen et al. 2002) define how an EROOS schema can be automatically translated into ID-Logic, which is an integration of first-order logic and logic programming. Once the schema is translated, general integrity constraints can be manually added to the ID-logic schema, and a general solver is used to reason on the schema. In particular, the reasoning task performed is scheduling, which consists in trying to achieve a goal specified by the user in a populated IB. For instance, assume an instance of the schema of Figure 1 consisting of an employee *Mary* assigned to a department *Sales*, and a goal to achieve that is having *Mary* also assigned to another existing department *IT*. In this case, this goal cannot be satisfied due to the cardinality constraint of the association *WorksIn*.

In (Formica 2003), a decidable method for checking finite satisfiability of Object-Oriented database schemas, based on a graph-theoretic approach, is provided. Schemas are defined by means of the language TQL*, an object-oriented language aimed at modeling the structural aspects, including some kinds of constraints, of object-oriented database schemas. The method can only deal with cardinalities, as well as a specific class of constraints, defined by means of a navigation path, a comparison operator and a constant.

A common drawback of all these methods is that none of them takes general integrity constraints into account. Additionally, these approaches are aimed at checking only some specific properties regarding the internal correctness of the schema.

UML Approaches

Some popular approaches to reason on UML class diagrams are based on Description Logics. Description Logics (DLs) are a family of formalisms for knowledge representation, based on first-order logic. In the last years, DLs have gone beyond their traditional scope in Artificial Intelligence area to provide new alternatives and solutions to many topics in the database and conceptual modeling areas (Borgida 1995; Borgida and Brachman 2003; Borgida, Lenzerini et al. 2003; Calvanese, Lenzerini et al. 1998; Lenzerini 1999). In DLs, reasoning is emphasized as a central service: it allows to infer implicitly represented knowledge from the knowledge explicitly contained in the knowledge base.

The applications of DLs and their reasoning facilities to data modeling and management are expressing the structural part of the conceptual schema and expressing and evaluating queries. Additionally, its inference mechanism can be used to determine some properties of the structural schema, such as satisfiability, liveliness or class subsumption.

DLs are a knowledge representation formalism, and one usually assumes that a system should always answer the queries of a user in reasonable time, which is not guaranteed by first-order logic. Decidability and complexity of the inference problems depend on the expressive power of each specific DL. On the one hand, very expressive DLs are likely to have inference problems of high complexity, or may even be undecidable. On the other hand, DLs with efficient reasoning procedures may not be expressive enough to represent the knowledge required by a real application.

An approach to reasoning on UML class diagrams is to translate them to DLs and then use current standard DL-based reasoning systems on them. An essential condition to guarantee decidability is the disallowance of general OCL constraints, which indeed are very important in conceptual modeling. In addition to this important limitation, every particular DL has its own restrictions, and even the most expressive ones are far below UML as far as expressiveness is concerned.

An important approach in this direction is (Berardi, Calvanese et al. 2005), which deals with a limited form of cardinality constraints (only of attributes or binary associations with an association class), as well as disjointness and covering constraints in hierarchies. The approach followed in this work is to translate a UML structural schema into ALCQI, and then use one of the existing reasoners to check satisfiability, liveliness, class subsumption and implicit consequence. The complexity of this DL is exponential, but the reasoning tasks still remain decidable. However, its expressive power is not sufficient for data modeling since, it doesn't allow key constraints, functional dependencies or associations with more than two participants.

The schemas dealt in (Fillottrani, Franconi et al. 2006) are a bit more expressive, since they can contain key constraints and inclusion and exclusion dependencies, in addition to cardinality, disjointness and covering constraints as in the work previously commented. However, the reasoning tasks performed are only class or association subsumption and inference of cardinality constraints.

One of the most recent works in this area is (Cadoli, Calvanese et al. 2007). The approach consists in encoding such UML class diagrams as a constraint satisfaction problem, and then perform finite model reasoning, i.e. check whether a class is forced to have either zero or infinitely many instances. To solve the resulting constraint satisfaction problems the authors use existing off-the-shelf tools. In this case, the constraints handled are again cardinality, disjointness and covering constraints. However, the UML class diagrams considered do not contain attributes, association classes or n-ary associations which, in addition to the lack of general constraints, are very important limitations.

1.5.2 On Validating a Conceptual Schema with Operations

In this section we review the previous work on validating conceptual schemas with a behavioral part. As will be seen, most of the approaches are able to validate that the preconditions and postconditions are correct, in the sense that they guarantee the applicability and executability of the operations. However, they do not consider the operations in the reasoning, which means that the results of the tests may report as valid states of the IB that are impossible to reach according to the changes defined by the operations.

Non-UML Approaches

Although considering the behavioral schema is not very common in practice, an interesting example in the context of deductive databases is (Díaz, Paton et al. 1998), which shows how the system would behave according to its specification by executing the model. If an unexpected outcome arises, the system helps the designer with explanations such as the ones of (Olivé and Sancho 1996). However, some structural features, such as class hierarchies, integrity constraints and derivation rules, have not been addressed in this work.

Another idea for the animation of a conceptual schema can be found in (Oliver and Kent 1999), which consists in determining all the valid states resulting from the application of an operation to an initial state. Here, the operation is defined declaratively, by means of a precondition and a postcondition in OCL.

For instance, assume the structural schema of Figure 1, considering that the cardinality constraint that forces an employee to be assigned to a department has been eliminated. Assume also an information base in which there is an instance *e* of *Employee* assigned to an instance *d* of *Department*. Then, given the contract for the operation *fire* specified in Figure 2, the execution of *fire(e)* to such information base can result in two different states, both of them satisfying the postcondition, i.e. without *e* being assigned to *d*: one still having instance *e* of *Employee* and the other in which *e* has been deleted from the information base.

Then, this validation approach allows to detect ambiguous or erroneous specifications of operation contracts by means of animation, since the designer can simulate the results of executing an operation on a certain IB and see whether they are the expected. This approach does not allow to automatically determine the correctness of a schema, since it only works on a populated IB and is not able to search for new instances that satisfy the desired properties.

In (Formica and Frank 2002) the problem of determining the consistency of an object-oriented specification is addressed. The structural schema is expressed in a textual language similar to Description Logics, but with some differences: it includes typing of attributes and does not support inheritance (subsumption). The kind of behavioral schema considered is by means of statecharts, which specify intra-object behavior. This means that they specify the behavior of an entity, but not the operations that can be performed on the whole conceptual schema. The kind of consistency checked is related to the integrity constraints that affect that entity and the conditions on the transitions of its associated statechart. In this way, they can detect inconsistencies like the fact that a transition can never be performed because it is impossible that some object satisfies the necessary conditions in order to execute it. However, this work has important expressive limitations. For instance, as well as the lack of inheritance, the constraints that can be expressed can only compare attribute values with constants, and not with other attributes.

Despite being normally used to represent and reason only on the structural schema, the Description Logics concepts can also be used to express preconditions and postconditions, in combination with a fragment of the Situation Calculus (Baader, Lutz et al. 2005). This combination provides expressiveness in the specification of actions, while guaranteeing decidability. With this formalism, properties regarding the correctness of operations (their applicability and executability) can be checked.

The Alloy language and analyzer (MIT 2006) provide interesting validation capabilities for more expressive schemas by searching for examples of the tests specified by the designer. In this case, the expressiveness of the structural schema and of the constraints are not limited. Additionally, operations can be specified and their preconditions and postconditions can be checked manually, that is, the designer asks for examples satisfying the precondition before the execution of the operation or satisfying the postcondition once it has been executed, but does not consider them when performing the tests on the structural schema. This means that an unreachable state may be reported as valid when, in fact, it is not with the operations given. Additionally, since the search space must be limited by the user, failure to find an example does not necessarily mean that one does not exist.

As can be seen, from the approaches to reason on non-UML schemas, Alloy is the only one that is expressive enough for the conceptual schemas we consider. However, its incompleteness and the fact that it does not consider the meaning of operations during the validation process, are important limitations.

UML Approaches

One of the first approaches to check satisfiability of UML schemas with operations is (Dupuy, Ledru et al. 2000). The schemas considered cannot contain n-ary associations. General constraints are handled, but they must be expressed in Z instead of OCL, which is the language recommended by the UML to formalize constraints and operations. Besides checking satisfiability of the structural part of the schema, operations to insert, delete and update the instances of each class or association are automatically generated. These operations are guaranteed to be applicable and executable according to the Z constraints specified in the schema. However, operations are not taken into account when determining the satisfiability of the schema.

An interesting tool to validate UML/OCL conceptual schemas is USE (Gogolla, Büttner et al. 2007), which allows to test if a given instantiation is accepted by the schema taking into account the OCL constraints. Preconditions and postconditions can also be validated, but the execution of the operation has to be simulated manually, inserting and deleting instances of the model, and then asking the tool to test whether the instantiation satisfies the postcondition. The instantiations must be manually provided, and a consequence of this is incompleteness, since the tool only checks whether the given instantiations are accepted by the schema and the constraints, but does not invent new instances that could be accepted while the ones given alone are not. Moreover, it cannot validate that the schema accepts an IB containing a subset of information defined declaratively.

An approach to reason on UML/OCL schemas is HOL-OCL (Brucker and Wolff 2006). The method uses a theorem prover to determine some properties on the schema, such as equivalence of two integrity constraints, and applicability and executability of operations. The theorems to be proved are defined in terms of the meta-model and, thus, it is not possible to check whether a certain instantiation is accepted by a schema or which is the sequence of operations that leads to a certain state.

There is another approach to reason on UML conceptual schemas, this time annotated with OCL constraints, that is based on translating the structural schema into a constraint satisfaction problem (Cabot, Clarisó et al. 2008; 2009). With this approach, several properties of the schema can be tested in order to check its internal correctness. In order to do this, the domain of each attribute must be made finite, which leads to incompleteness: failure to find an IB satisfying a certain property does not mean that one does not exist with other values beyond the specified bounds. Regarding the behavioral schema, some properties regarding the correct definition of the operations, such as applicability, executability or precondition redundancy, can be tested.

We may note that all the UML approaches that consider the behavioral part have an important common drawback. None of them takes into account the definition of operations when determining whether a state is accepted or not by the schema. This means that they may report as valid a state satisfying all the constraints but that is impossible to construct using the operations defined in the schema. This also damages the results obtained when testing the correctness of operations themselves, since the states satisfying certain properties (for instance, a state satisfying the precondition when testing the applicability of an operation) may not be reached using the operations defined. Moreover, they cannot automatically construct the sequence of operations resulting in a certain state.

An exception is an approach that belongs to the Rodin project. It combines UML-B (Snook and Butler 2006) and ProB (Leuschel and Butler 2008), the former to represent the schema and translate it into the B language, and the latter to validate it by animation. One of its drawbacks is that UML-B only accepts a subset of the UML that is suitable for translation into B, which is defined through an ad-hoc profile. Moreover, constraints and operations must be directly expressed in B by the designer. In contrast, our models can be expressed in standard UML and OCL, which are the languages most commonly used in conceptual modeling. Regarding the animation process, the operations handled by ProB must incorporate the semantics of the constraints, which are checked after the simulated execution. On the contrary, our approach is able to deal with constraints as such, taking care of maintaining them while constructing the sample state. Additionally, ProB requires that the state space is made finite by enumerating the values to be used in the animation. Since the fact that a property does not hold for those values does not mean that it can never hold, completeness is not guaranteed by this approach.

1.6 Contributions of this Thesis

The goal of this thesis is to propose a method to validate both the structural and the behavioral parts of a UML conceptual schema. The structural part is defined by means of a UML class diagram and a set of constraints formalized in OCL, while the behavioral part consists of a set of operations, defined by means of operations contracts also formalized in OCL.

Two kinds of tests can be made on a conceptual schema using our method. On the one hand, our method can check the internal correctness of the schema by automatically verifying whether it satisfies a set of desirable properties. Some of these properties correspond to well-known reasoning tasks such as schema satisfiability, class liveliness or redundancy of integrity constraints, while others are an original contribution of this thesis. In short, these additional automatically generated tests allow to check whether the cardinalities of associations and the disjointness and covering constraints in hierarchies are correctly defined.

On the other hand, to check the external correctness of the schema, we provide the designer with a set of tests that can be automatically generated from the schema, as well as with the ability to ask any questions to see whether certain situations are accepted by the conceptual schema.

Roughly, our approach consists in automatically translating the UML schema with its OCL constraints and operations into a logical representation. Once this translation is done, we are able to perform the validation tests. We express all of them (the ones to check internal correctness and the ones to check the external correctness of the schema) in terms of checking the satisfiability of a derived predicate. In this way, for each validation test to be performed, a derived predicate (with its corresponding derivation rule) that formalizes the desired test is defined. With this input, together with the translated schema itself, any satisfiability checking method that is able to deal with negation of derived predicates can be used to validate the schema.

Our approach to reasoning is aimed at constructing an IB which shows that a certain property holds. That is, an IB where both the particular condition to be tested and all the integrity constraints in the schema are satisfied. In this way, our method can uniformly deal with all the validation tests.

It is well known that the problem of reasoning with integrity constraints in its full generality, as we want to do, is undecidable since general constraints can cause the schema to have an infinite number of models (or consistent IBs). Then, two different approaches can be followed, either based on decidable procedures for certain restricted kinds of constraints or limited domains, or based on semidecidable procedures for highly expressive constraints. Since we do not want to renounce to completeness, nor to a high expressiveness of constraints, we take a mixture of both directions. Once we have translated the UML schema with its OCL constraints into logic, we provide a set of conditions that, in case they are satisfied by the schema, guarantee that the schema does not have any infinite model, which implies that any reasoning task performed on it will terminate. The novelty of our approach lies in admitting a high expressiveness of the constraints, for which decidability is not guaranteed, and then determine whether termination can be ensured for each particular schema.

The conditions to determine decidability are not applicable to the behavioral part of the schema. In this case, the reasoning is performed after the translation of the schema into logic, without knowing a priori whether it will terminate.

In the following subsections we outline the different steps of our method, that is, the translation of the schema into logic, which depends on whether the behavioral schema is considered or not, the determination of the decidability of reasoning, which can only be checked on the structural schema, and the reasoning procedure, which is common for both cases.

1.6.1 Validating the Structural Part of the Schema

Before reasoning on the schema to check the desired properties, the schema must be translated into logic (Queralt and Teniente 2006a). When we want to validate the structural schema alone, we can also check whether the problem of reasoning on the particular schema to be validated is decidable (Queralt and Teniente 2008a). Once this has been done, the reasoning is performed as explained in section 1.6.3.

Translating the Schema into Logic

When validating the structural schema alone, classes, attributes and associations are represented in logic by means of basic predicates. Classes are translated into unary predicates, attributes into binary predicates and associations become predicates of their same arity. For instance, in the example of Figure 1, classes *Employee* and *Department* become *employee(E), employeeName(E,N)* and *department(D), departmentName(D,N),* and the binary association *WorksIn* is represented by *worksIn(E,D)*.

The OCL constraints of the schema are translated into formulas in denial form, which represent conditions that must not be satisfied by any state of the IB. For instance, the constraint *BossHasNoSuperior* is translated into:

$\leftarrow boss(E) \wedge worksFor(S,E)$

which means that it is not possible that an instance *e* of *Boss* exists such that there is an employee *s* that is his superior.

The graphical constraints of the schema, such as cardinality and taxonomic constraints (*disjoint* and *complete*), also need to be translated into this kind of conditions.

A class diagram also has a set of implicit constraints that need to be taken into account in the logical representation of the schema to preserve the semantics of the original one. For example, since UML is an object-oriented language, each instance has an internal object identifier (OID) which uniquely differentiates two instances, even though they are externally equivalent. Thus, additional constraints are needed in the logical representation to guarantee that two instances of the schema do not have the same OID. In the example of Figure 1 we need to specify the following constraint:

$\leftarrow employee(X) \wedge department(X)$

As well as OIDs, the implicit constraints we can find in a class diagram are:

- In class hierarchies, an instance of a subclass must also be an instance of the superclass.
- In associations or association classes, an instance of the association must link instances of the classes that define the association.
- In association classes, there cannot exist several instances linking exactly the same instances. Note that this is also true for associations without an association class, but an additional constraint is not needed in this case, since predicates representing n-ary (n>=2) associations have exactly n terms that can not be identical in two different instances of the predicate.

In section 3.1 the translation is explained in detail.

Determining Decidability

The logic representation of the schemas obtained from the translation of a UML schema with its OCL constraints has a specific structure. In particular, it has only one level of derivation, that is, derived predicates are never defined by other derived predicates. To satisfy this restriction, although OCL constraints can include the operations `includes`, `includesAll`, `notEmpty`, `exists` and `one`, these specific operations cannot be recursively combined in an OCL expression.

This mild syntactical restriction allows us to determine whether any reasoning task performed on the schema will terminate. To do this, we provide a set of conditions that guarantee that the schema does not have any infinite model, which means that reasoning on the schema will always terminate. Our approach consists in constructing a graph from the set of constraints of the schema that shows the existing dependencies between them, and then study the cycles in the graph to determine the absence of infinite models, as we will see in section 3.2.

1.6.2 Validating the Schema with its Behavioral Part

In order to validate a complete conceptual schema, we must take into account that the only changes that can be performed on the IB are those defined in the operations of the behavioral schema. For this reason, the translation into logic is different from the previous case (Queralt and Teniente 2008b). Now, since the resulting translation is more expressive than before, we are not able to determine the decidability, so the reasoning is performed immediately after the translation, as explained in section 1.6.3, without knowing whether it will always terminate.

Translating the Schema into Logic

Validation tests that consider the structural schema alone are aimed at checking that an instantiation fulfilling a certain property and satisfying the integrity constraints can exist. In this case, classes, attributes and associations can be translated into base predicates that can be instantiated as desired, as long as integrity constraints are satisfied, in order to find a state of the IB that proves a certain property (Queralt and Teniente 2006a).

However, when considering also the behavioral schema, the population of classes and associations is only determined by the events that have occurred. In other words, the state of the IB at a certain time *T* is just the result of all the operations that have been executed before *t,* since the instances of classes and associations cannot be created or deleted as desired. For instance, according to our schema in Figure 1 and the operations defined, *Mary* can only be an instance of *Employee* at a time *T* if the operation *hire* has created it at some time before *T* and the operation *fire* has not removed it between its creation and *T.*

For this reason, it must be guaranteed that the population of classes and associations at a certain time depends on the operations executed up to that moment. To do this, we propose that operations are the basic predicates of our logic formalization, since their instances are directly created by the user. Classes and associations will be represented by means of derived predicates instead of basic ones, and their derivation rules will ensure that their instances are precisely given by the operations executed. Although we assume a strict interpretation of operation contracts, our method could correctly handle also those operations specified under an extended interpretation, by defining an alternative translation of the schema into logic that incorporates this extended semantics (Queralt and Teniente 2006b).

The complete translation of a schema, with its operations specified in OCL, will be explained in section 4.2.

1.6.3 Reasoning on the Schema

For those cases in which the conditions of termination are satisfied, although any theorem prover or reasoning method can be used knowing that any reasoning task performed on the schema will terminate, we provide a reasoning procedure that always terminates and works more efficiently than in the general case. We outline it here, and will be explained in detail in section 3.3.

The procedure consists of two different steps: *goal satisfaction* and *integrity maintenance.*

As we mentioned, we formalize the satisfaction of each test in terms of checking the satisfiability of a derived predicate. This derived predicate is the *goal* to attain in the IB constructed by our procedure. One of the most difficult tasks is the assignment of concrete values to the variables appearing in the goal in order to construct the sample IB. Each possible choice defines a different alternative that satisfies the goal, i.e. a different sample IB.

We use *Variable Instantiation Patterns (VIPs)* (Farré, Teniente et al. 2005) for this purpose. These VIPs guarantee that the number of sample IBs to be considered is kept finite, by taking into account only those variable instantiations that are relevant for the schema, without losing completeness. That is, the VIPs guarantee that if a solution is not found by instantiating the variables in the goal using only the constants they provide, then no solution exists.

Once we have determined the set of facts that satisfies the goal to attain, the problem of reasoning on the schema can be reduced to that of *integrity maintenance* (Moerkotte and Lockemann 1991). Note that, in fact, we know that the property checked will be satisfied if the IB resulting from the previous step does not violate any constraint of the schema. If this is not the case, we must look for additional base facts (i.e. repairs) that make the sample IB being constructed fulfill all constraints.

Unfortunately, we may not rely on existing integrity maintenance methods to perform this activity. On the one hand, some methods like (Console, Sapino et al. 1995; Lobo and Trajcevski 1997) can only handle restricted types of integrity constraints which do not cover the kind of constraints we obtain as a result of the translation of the conceptual schema into logic. On the other hand, most methods do not provide an appropriate treatment to the existential variables that appear in the integrity constraint definition (Ceri, Fraternali et al. 1994; Decker, Teniente et al. 1996; Mayol and Teniente 2003; Schewe and Thalheim 1999). The general approach of these methods when instantiating an existential variable is either asking for a value from the user at run-time or assigning an arbitrarily chosen value of the corresponding data type. This is not suitable when using integrity maintenance for reasoning since only a few of the possible alternatives (just one in most cases) would be taken into account to repair a violated constraint. Therefore, this approach does not guarantee the completeness of the result since the impossibility to find a sample IB would not necessarily imply that the tested property does not hold. To our knowledge, the most appropriate method to perform the kind of integrity maintenance we require is the CQC-Method (Farré, Teniente et al. 2005). However, and in addition to the drawback that it is a semidecision procedure, the CQC-Method has important efficiency limitations. Thus, we need to build a new reasoning procedure, which can take advantage both of the dependency graph and the characterization of the logic formulas obtained from our schemas to work efficiently. Since the graph shows the interactions between the constraints, it provides the order in which they should be maintained.

There are some cases in which our reasoning procedure cannot be used, in particular, when the structural schema has some infinite model, or when the schema is more expressive than required to determine the absence of infinite models. This happens when the constraints contain certain combinations of OCL operations, as we have seen, or when we want to validate also the behavioral part. In these cases, any reasoning method can be used to test the correctness of the schema, but this time termination will not be ensured. Our approach consists in using the CQC-Method, which is the most suitable one to deal with the expressiveness of our constraints since, in these cases, its decidability and efficiency drawbacks cannot be overcome. However, it has been necessary to extend it so that it can correctly handle the operations.

After reviewing some basic concepts in chapter 2, we explain in chapter 3 our approach to validate the structural part of a conceptual schema. Here we formalize the translation from UML and OCL into logic, and provide a set of tests to check both the internal and the external correctness of the structural schema.

In chapter 4, the reasoning procedure for the structural schema is detailed. We explain our approach to determine whether reasoning on a given schema will always terminate, and then propose an algorithm to perform the validation in this case.

Chapter 5 is devoted to the validation of schemas including the behavioral part. We formalize the semantics of operations and our approach to validation, which is similar to the one proposed for the structural schema. The reasoning procedure and its implementation in this case are based on an existing method.

In Chapter 6 we show a prototype that implements the validation of a structural schema, from its translation into logic to the execution of the validation tests.

The previous work introduced in section 1.5 is analyzed in depth in Chapter 7. Here we provide a detailed comparison of previously existing approaches and show how this thesis contributes to the validation of conceptual schemas, with and without operations.

Finally, in Chapter 8 we expose our conclusions and point out further research directions based on this thesis.

The method presented in this thesis has been in part published in (Queralt and Teniente 2006a; 2006b; 2008a; 2008b).

2

Basic Concepts

In this chapter we provide definitions to the concepts we use throughout the document. It is divided into two sections, in section 2.1 we define those concepts regarding conceptual modeling of information systems, assuming that the reader is familiar with the syntax of UML class diagrams and OCL constraints. In section 2.2 we define the logic concepts underlying our method.

2.1 Basic Concepts on Conceptual Modeling

The *conceptual schema* of an information system must include all relevant static and dynamic aspects of its domain (ISO/TC97/SC5/WG3 1982). The part of a conceptual schema that deals with static aspects is called the structural schema and the part that deals with dynamic aspects is called the behavioral schema.

The *structural schema* consists of a taxonomy of entity types together with their attributes, a taxonomy of relationship types among entity types, and a set of integrity constraints. An information system maintains a representation of the state of a domain in its information base (IB). The state of the IB is the set of instances of the entity types and relationship types defined in the structural schema. The integrity constraints define conditions that each state of the IB must satisfy. Those constraints can have a graphical representation or can be defined through a particular language.

Without loss of generality, the only graphical constraints we consider as such are cardinalities and disjointness and covering constraints in hierarchies, due to their widespread use. Since we deal with general OCL, we assume that other graphical constraints (such as *subset* or *xor*) are expressed textually (Gogolla and Richters 2002).

We would like to remark that, as can be seen in chapters 3 and 5, our method is able to deal with n-ary associations, with and without association classes, disjointness and covering constraints in hierarchies and attribute cardinalities, despite they do not appear in the running example.

The *behavioral schema* contains a set of *operations*, which define the only changes that can be made on the IB. The effect of each operation on the IB is specified by an operation contract. An *operation contract* is defined by a *precondition*, which expresses a condition that must be satisfied when the call to the operation is made, and a *postcondition*, which expresses a condition that the new state of the IB must satisfy. The content of the IB changes as a result of the execution of the operations specified in the behavioral schema. Changes in the state of the IB are defined by a set of one or more structural events to be applied that are drawn from the preconditions and postconditions of the operation contracts.

2.2 Basic Concepts on the Logic Formalization

This section sets the formal background required for the technical development of the rest of this document. In particular, we recall some basic concepts and notation of deductive databases.

Throughout the document, a, b, c, a_1, b_1,… are constants. The symbols X, Y, Z, X_1, Y_1, … denote variables. Sets of constants are denoted by $\bar{a}$, $\bar{b}$, $\bar{c}$, $\bar{a}_1$, $\bar{b}_1$, ..., and $\bar{X}$, $\bar{Y}$, $\bar{Z}$, $\bar{X}_1$, … denote sets of variables. Predicate symbols are p, q, r, p_1, q_1, … A *term* is either a variable or a constant. If p is a n-ary predicate and T_1, …, T_n are terms, then $p(T_1, …, T_n)$ is an *atom*, which can also be written as $p(\bar{T})$ when n is known from the context. An atom is *ground* if every T_i is a constant. An *ordinary literal* is defined as either an atom or a negated atom, i.e. $\neg\, p(\bar{T})$. A *built-in literal* has the form of $A_1\ \omega\ A_2$, where A_1 and A_2 are terms. Operator ω is either $<$, $\leq$, $>$, $\geq$, $=$ or $\neq$.

A *normal clause* has the form

$A \leftarrow L_1 \wedge \ldots \wedge L_m$ with $m \geq 0$,

where A is an atom and each L_i is a literal, either ordinary or built-in. All the variables occurring in A, as well as in each L_i, are assumed to be universally quantified over the whole formula. A is often called the *head* and $L_1 \wedge \ldots \wedge L_m$ is the *body* of the clause.

A set of normal clauses is called a *normal program*. The *definition* of a predicate symbol r in a normal program P is the set of all clauses in P that have r in their head.

Terms, literals and the syntactic structures made of them, such as rule bodies, whole rules or facts, are *expressions*. If E is an expression, then *constants*(E) and *variables*(E) are the sets containing the constants and variables, respectively, occurring in E.

A *substitution* θ is a set of the form $\{X_1/T_1, …, X_n/T_n\}$, where each variable X_i is unique and each term T_i is different from X_i. The term T_i is called a *binding* for X_i. θ is called a *ground substitution* if each T_i is a ground term, that is, a constant.

Let E be an expression and $\theta = \{X_1/T_1, \ldots, X_n/T_n\}$ a substitution. Then Eθ, the *instance* of E by θ, is the expression obtained from E by simultaneously replacing each occurrence of the variable X_i in E by the term T_i.

A *fact* is a normal clause of the form: $p(\bar{a}) \leftarrow$, where $p(\bar{a})$ is a ground atom. The fact $p(\bar{a}) \leftarrow$ may also may be denoted by $p(\bar{a})$.

A *deductive rule* is a normal clause of the form:

$p(\bar{T}) \leftarrow L_1 \land \ldots \land L_m$ with $m \geq 1$

where *p* is the *derived* predicate defined by the deductive rule.

A *condition* is a formula of the (*denial*) form:

$\leftarrow L_1 \land \ldots \land L_m$ with $m \geq 1$

which states a condition that cannot hold.

A *database schema S* is a tuple (*DR*, *IC*) where *DR* is a finite set of deductive rules and *IC* is a finite set of conditions. Literals occurring in the body of deductive rules and conditions in *S* are either ordinary or built-in. The predicate symbols in ordinary literals range over the *extensional database* (EDB) predicates, which are the relations that will be stored directly in the database, and the *intensional database* (IDB) predicates, which are the relations defined by the deductive rules in *DR*. EDB predicates cannot be derived. Conditions in IC define the *integrity constraints* of the schema *S*.

Deductive rules as well as conditions are required to be *safe*, that is, every variable occurring in the head or in negated or built-in atoms of their body must also occur in an ordinary positive literal of the same body.

For a database schema *S* = (*DR*, *IC*), a *database state*, *database instance*, or just *database*, *D* is a tuple (*E*, *S*) where *E* is an EDB, that is, a set of ground facts about EDB predicates. *DR*(*E*) denotes the whole set of ground facts about EDB and IDB predicates that are inferred from a database state *D* = (*E*, *S*). *DR*(*E*) corresponds to the fixpoint model of $DR \cup E$.

A database *D violates* (*does not satisfy)* a condition $\leftarrow L_1 \land \ldots \land L_n$ if there exists a ground substitution θ such that $D \models (L_1 \land \ldots \land L_n)\theta$. In other words, when $\{L_1\theta, \ldots, L_n\theta\} \subseteq DR(E)$. A database *D* is *consistent*, or *sound*, when it violates no condition in *IC*.

3

Validation of the Structural Schema

The structural part of a conceptual schema, or *structural schema,* is defined by means of a UML class diagram (OMG 2007), with its graphical constraints, and a set of user-defined constraints, which can be specified in any language. According to (Warmer and Kleppe 2003), we assume they are formalized in OCL.

Under the concept of *Validation* we include two kinds of reasoning tests. On the one hand, there are those tests devoted to determining whether the conceptual schema is correctly defined, in the sense that it allows, for instance, creating instances of all its classes and associations, or that it does not include any redundancies among constraints. These tests, which can be performed without any knowledge about the domain, are sometimes referred to as *verification*, and correspond to answering to the question *Is the structural schema right?*

On the other hand, the fact that a conceptual schema is correctly defined in the sense explained above, does not necessarily imply that it correctly represents the domain. Thus, it is necessary to perform additional tests to ensure the correctness of a schema from the point of view of its correspondence to the requirements. This is usually known as *validation*, which aims to answer the question *Is it the right structural schema?*

Roughly, our approach consists in automatically translating the UML structural schema, with its OCL constraints, into a logical representation, as explained in section 3.1. In section 3.2 we formalize the tests that can be performed on a structural schema, to determine both its internal and external correctness.

Once the translation is done and the tests are defined, we are ready to reason on the schema, both to determine whether *it is right*, and whether it is *the right one*. Our reasoning procedure for both kinds of tests will be seen in chapter 4.

3.1 Translating a UML Conceptual Schema into Logic

In this section we propose a set of rules that, applied to a UML class diagram and a set of OCL constraints, result in a set of first-order formulas that represent the structural schema. The subset of first-order logic considered does not provide functions; and rules and conditions are required to be safe, that is, every variable occurring in their head or in atoms of their body that are negated or built-in must also occur in an ordinary positive literal of the same body.

We will explain first how to obtain the formulas for the class diagram, taking into account its implicit and graphical constraints. Later, we propose a translation for the user-defined OCL constraints. The complete logic representation of the schema can be found in the Appendix A.

To illustrate the translation of a UML structural schema, with its constraints specified in OCL, we will use a subset of the example in Figures 1 and 2. In particular, we omit the information about working teams, so the simplified schema is as shown in Figures 4 and 5.

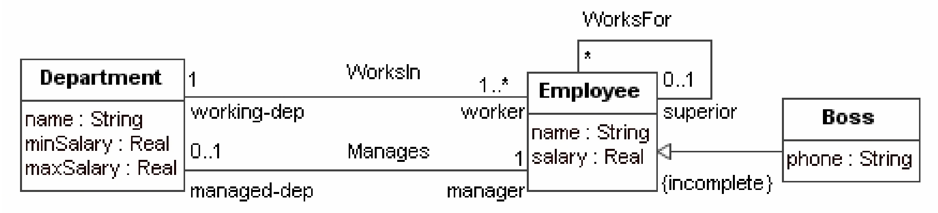


Fig. 4. Subset of the running example showing only Employees and Departments

Due to the logical representation we use as a target of our translation, the OCL operations that can appear in all the OCL expressions we consider are those that result in a boolean value. Exceptions are `select` and `size` that, despite returning a collection and an integer, can also be handled by our method. Additionally, those OCL operators that can only be used in preconditions or postconditions of operations, namely `oclIsNew` and `@pre`, are neither considered when dealing with the structural schema alone.

```
Integrity constraints

1.   context Department inv UniqueDep:
     Department.allInstances()->isUnique(name)

2.   context Employee inv UniqueEmp:
     Employee.allInstances()->isUnique(name)

3.   context Department inv MinimumSalary:
     self.minSalary > 1000

4.   context Department inv CorrectSalaries:
     self.minSalary < self.maxSalary

5.   context Department inv ManagerIsWorker:
     self.worker->includes(self.manager)

6.   context Department inv ManagerHasNoSuperior:
     self.manager.superior->isEmpty()

7.   context Boss inv BossIsManager:
     self.managed-dep->notEmpty()

8.   context Boss inv BossHasNoSuperior:
     self.superior->isEmpty()

9.   context Boss inv SuperiorOfAllWorkers:
     self.employee->includesAll(self.managed-dep.worker)
```

Fig. 5. Integrity constraints corresponding to the fragment of the schema in Figure 4.

3.1.1 Translation of a UML Class Diagram

A UML class diagram is translated into a set of first-order formulas according to the following rules.

Translation of Classes

For each class C not being an association class we define a unary predicate *c*, where its term represents the internal object identifier (OID).

For example, the class *Employee* is translated into a predicate *employee(E).*

Translation of Attributes, Associations and Association Classes

Let R be an association between classes $C_1,...,C_n$. If R is not an association class, we define a base predicate *r*$(C_1,...,C_n)$. Otherwise, if R is an association class we define a base predicate *r*$(R,C_1,...,C_n)$. Although it is not strictly necessary, we also include an OID *r* so that all classes can be treated uniformly.

For example, the association *WorksIn* that relates *Employees* and *Departments* is translated into the predicate *worksIn(E,D)*.

Attributes can be regarded as binary associations between a class C and a datatype. Then, for each attribute a_i in C we define a binary predicate $cA_i(C,A_i)$. Note that, since several classes can have an attribute with the same name, we need to use the class name in the definition of the predicates representing attributes.

For example, the attribute *name* of *Employee* is translated into *employeeName(E,N).*

Translation of implicit and graphical constraints

First of all, we must guarantee that there cannot exist two instances with the same OID. This is already guaranteed for instances of the same class, since they are represented by unary predicates. Then, we must define rules to prevent the existence of two literals of different predicates with the same OID, defining the following constraint for each pair of predicates not representing classes in the same hierarchy:

$\leftarrow c_1(X) \land c_2(X)$

According to this rule, we must define the following constraint in our example:

$\leftarrow employee(X) \land department(X)$

Class hierarchies also require the definition of a set of constraints to guarantee that an instance of each subclass C_{subi} is also an instance of its superclass C_{super}. This is done by means of the rule:

$\leftarrow c_{subi}(C) \land \neg c_{super}(C)$

In the example, the hierarchy of employees requires the following constraint:

$\leftarrow boss(E) \land \neg employee(E)$

Moreover, additional rules are sometimes required to guarantee that an instance of the superclass is not an instance of several subclasses simultaneously (*disjoint* constraint), or that an instance of the superclass is an instance of at least one of its subclasses (*complete* constraint). Then, for each pair of subclasses C_{subi} , C_{subj} we define a constraint stating that an instance cannot belong to both of them simultaneously:

$\leftarrow c_{subi}(C) \land c_{subj}(C)$

and another one stating that an instance of C_{super} must belong to at least one of the C_{subi}. To do this we need a derived predicate *isKindOfC*$_{super}$, with a rule for each C_{subi}:

$\leftarrow c_{super}(C) \land \neg isKindOfC_{super}(C)$
$isKindOfC_{super}(C) \leftarrow c_{subi}(C)$

Another set of constraints is needed to guarantee the implicit constraint that an instance of an association can only relate existing instances of the classes that define it. Then, for each association R, being or not an association class with OID r, represented by the predicate $r([R,],C_1,...,C_n)$, we define the following constraint for each C_i:

$\leftarrow r([R,],C_1,...,C_n) \land \neg c_i(C_i)$

In our example, the association *WorksIn* requires the addition of the rules:

$\leftarrow worksIn(E,D) \land \neg employee(E)$
$\leftarrow worksIn(E,D) \land \neg department(D)$

Similarly, we must define constraints to guarantee that the first term of a predicate representing an attribute corresponds to an instance of the class to which the attribute belongs. In our example:

$\leftarrow employeeName(E,N) \wedge \neg employee(E)$
$\leftarrow bossPhone(E,P) \wedge \neg boss(E)$
$\leftarrow departmentName(D,N) \wedge \neg department\ (D)$

Additionally, for the definition of association classes, we must guarantee that there are not several instances of an association class having the same value in the terms defining the instance. Then, if *R* is an association class, defined by classes $C_1,...,C_m$, we define the following constraint:

$\leftarrow r(R1,C_1...C_m) \wedge r(R2,C_1...C_m) \wedge R1<>R2$

Finally, let *min..max* be a cardinality constraint attached to an attribute or to a class C_i in an association *R* defined by classes $C_1,...,C_n$. If *min*>0 we must add the following constraint:

$\leftarrow c_1(C_1) \wedge ... \wedge c_{i-1}(C_{i-1}) \wedge c_{i+1}(C_{i+1}) \wedge ... \wedge c_n(C_n) \wedge \neg minR(C_1,...,C_{i-1},C_{i+1},...,C_n)$
$minR(C_1,...,C_{i-1},C_{i+1},...,C_n) \leftarrow r([R1,]C_1,...,C_{i-1},C_i1,C_{i+1},...,C_n) \wedge ...$
$\wedge\ r([Rmin,]C_1,...,C_{i-1},C_imin,C_{i+1},...,C_n) \wedge$
$\wedge\ C_i1<>C_i2 \wedge ... \wedge C_i1<>C_imin \wedge ... \wedge C_imin\text{-}1<>C_imin$

And if *max* < *, the following constraint is needed:

$\leftarrow r([R1,]C_1,...,C_{i-1},C_i1,C_{i+1},...,C_n) \wedge ... \wedge$
$r([Rmax+1,]C_1,...,C_{i-1},C_imax+1,C_{i+1},...,c_n) \wedge C_i1<>C_i2 \wedge ...$
$\wedge\ C_i1<>C_imax+1 \wedge ... \wedge C_imax<>C_imax+1$

As an example, we must define the following constraint to guarantee the lower multiplicity of class Employee in the association *WorksIn*:

$\leftarrow department(D) \wedge \neg oneWorker(D)$
$oneWorker(D) \leftarrow worksIn(E,D)$

And we also have to define the following one due to the upper multiplicity of *Department* in the same association:

$\leftarrow worksIn(E,\ D1) \wedge worksIn(E,\ D2) \wedge D1<>D2$

For attributes, if no multiplicity is specified in the class diagram, it is assumed that they are single-valued and not optional. Then, in our example we need the constraints:

$\leftarrow employee \wedge \neg oneEmployeeName(E)$

$oneEmployeeName(E) \leftarrow employeeName(E,N)$

$\leftarrow employeeName(E,N1) \wedge employeeName(E,N2) \wedge N1<>N2$

Analogous constraints are needed for *phone* in *Boss* and *name* in *Department*.

3.1.2 Translation of OCL Integrity Constraints

We perform the translation of OCL integrity constraints into first-order logic in two steps. First, we transform each OCL expression into an equivalent one expressed in terms of the operations `select` and `size`. Both `select` and `size` are OCL operations that apply to collections of elements, `select` returns the subset of the collection that satisfies a condition, and `size` returns the number of elements in the collection. The aim of this transformation is to reduce the number of OCL constructs to be translated, so that a uniform treatment can be applied to all constraints in order to obtain the corresponding logic formulas.

Simplification of OCL operations

The first step in the translation process consists in the reduction of the number of OCL operations that appear in the constraints. Table 1 shows the OCL operations we consider, and gives their equivalent simplified expressions. These translations are iteratively applied until the only OCL operations that appear in the expression are `select` and `size`.

Table 1. Equivalences of OCL operations

Original expression	Equivalent expression with *select* and *size*
source->includes(obj)	source->select(e | e=obj)->size()>0
source->excludes(obj)	source->select(e | e=obj)->size()=0
source->includesAll(c)	c->forall(e| source->includes(e))
source->excludesAll(c)	c->forall(e| source->excludes(e))
source->isEmpty()	source->size()=0
source->notEmpty()	source->size()>0
source->exists(e | body)	source->select(e | body)->size()>0
source->forall(e | body)	source->select(e | not body)->size()=0
source->isUnique(e | body)	source->select(e |source->select(e2 | e <>e2 and e2.body = e.body))->size()=0
source->one(e | body)	source->select(e | body)->size()=1
source->reject(e | body)	source->select(e | not body)

As an example we give the simplified form of *ManagerIsWorker* in our example:

```
context Department inv ManagerIsWorker:
self.worker->select(e |e = self.manager)-> size() > 0
```

Notice that if a department could have many managers (and we still wanted all of them to be workers of the same department) the expression obtained from the simplification would have been different, since the operation `includesAll` would appear instead of `includes` in the original OCL expression.

Translation of OCL invariants into logic

Once simplified, an OCL invariant has the following form:

```
context C inv: path-exp->select(e| body)->size() opComp k
```

where *C* is a class, *path-exp* is a sequence of navigations through associations, *opComp* is a comparison operator <, >, = or <> and *k* is an integer not lower than zero[1].

The translation of the simplified OCL invariants into logic depends on the specific operator after `size()`. We are going to see first how to translate the navigation defined by *path-exp* and the translation of the `select` operation. The select expression does not necessarily appear in the simplified OCL invariant, in which case it is not translated.

Tr-path(path-exp). Let path-exp = obj.r1...rn[.att] be a path starting from an instance obj of a class C, or from a call to the allInstances operation on C, navigating through roles r1 to rn and, optionally, ending with the access to an attribute. Let C(obj) be the literal resulting from the translation of the class to which obj belongs, and Ri(obji-1, obji,...) be the literals corresponding to the association between roles ri-1 and ri, and C2 be the class where the attribute att is defined. Then, this navigation path is translated into logic by means of the clause:

$$c(Obj) \wedge r_1(Obj,Obj_1,\ldots.) \wedge \ldots \wedge r_n(Obj_{n-1}, Obj_n,\ldots) [\wedge c_2(Obj_n) \wedge c_2Att(Obj_n, Att)]$$

For instance, the navigation `self.worker` appearing in constraint *ManagerIsWorker* will be translated into *department(D)* $\wedge$ *worksIn(E,D)*.

Tr-select(e| body). We provide here the translation of a select expression in its most simplified and usual form, where body = path1 opComp path2. In this case, the select operation is translated into:

$$Tr\text{-}path(path1) \wedge Tr\text{-}path(path2) \wedge Obj1\ opComp\ Obj2$$

where *Obj1* and *Obj2* are the objects obtained as a result of the navigation paths *path1* and *path2*, respectively. Note that if any of the paths is a constant or *e*, then it must not be translated.

For instance, the translation of the expression `select(e| e=self.manager)` appearing in the simplified OCL invariant of the constraint *ManagerIsWorker* will be translated into *department(D)* $\wedge$ *manages(E2,D)* $\wedge$ *E=E2*.

The body of a `select` operation can also contain, recursively, other `select` and `size` operations, that is,

```
body = path-exp->select(e| body)->size() opComp k
```

where the `select` operation may not appear, and then it is not translated. We define the translation in terms of the translation of `path-exp` and the `select` operation as follows, depending on `opComp`, when `opComp` is <, > or =:

a) `obj.r`$_1$`... r`$_{n-1}$`.r`$_n$` -> select(e| body)-> size() < k`
becomes

$Tr\text{-}path(obj.r_1\ldots r_{n-1}) \wedge \neg aux(e,\ldots,e_m,c)$

$aux(e,\ldots,e_m,c) \leftarrow Tr\text{-}path_1(r_n) \wedge Tr\text{-}select_1(e|\ body)$
$\wedge \ldots \wedge Tr\text{-}path_k(r_n) \wedge Tr\text{-}select_k(e|\ body)$

[1] When ≤k or ≥k appear in the original invariant and k is an integer greater than 0, they are translated into <k+1 and >k-1. Those cases in which k is equal to 0 do not represent valid constraints and, thus, they are not taken into account.

b) `obj.r`$_1$`...  r`$_{n-1}$`.r`$_n$` -> select(e| body)-> size() > k`
becomes

$Tr\text{-}path(obj.r_1...\ r_{n-1}) \wedge Tr\text{-}path_1(r_n) \wedge Tr\text{-}select_1(e|\ body)$
$\wedge \ldots \wedge Tr\text{-}path_{k+1}(r_n) \wedge Tr\text{-}select_{k+1}(e|\ body)$

c) `obj.r`$_1$`...  r`$_{n-1}$`.r`$_n$` -> select(e| body)-> size() = k`
becomes

$Tr\text{-}path(obj.r_1...\ r_{n-1}) \wedge Tr\text{-}path_1(r_n) \wedge Tr\text{-}select_1(e|\ body)$

$\wedge \ldots \wedge Tr\text{-}path_k(r_n) \wedge Tr\text{-}select_k(e|\ body) \wedge \neg aux(e,...,e_m,c)$

$aux(e,...,e_m,c) \leftarrow Tr\text{-}path(obj.r_1...\ r_{n-1}) \wedge Tr\text{-}path_1(r_n) \wedge Tr\text{-}select_1(e|\ body)$

$\wedge \ldots \wedge Tr\text{-}path_{k+1}(r_n) \wedge Tr\text{-}select_{k+1}(e|\ body)$

where the variables $e,...,e_m$ needed by each Aux predicate correspond to the iteration variables of all the select operations in which the translated *body* is included, and *c* represents the contextual object.

Each translation Tr-path or Tr-select may be performed several times depending on the constant *k*. Each of the Tr-path$_i$ or Tr-select$_i$ expressions refers to the same translation but with different variables for those terms not coming from the translation of `obj.r`$_1$`...r`$_{n-1}$.

As an example, consider the OCL simplified invariant of constraint *SuperiorOfAllWorkers*:

```
context Boss inv SuperiorOfAllWorkers:
self.managed-cat.worker-> select(e | self.employee->
  select(e2 | e2=e)->size()=0)->size()=0
```

applying the translation c) above we obtain, as a translation of the inner `select`:

$Tr\text{-}path(self) \wedge \neg aux(e,b)$
$aux(e,b) \leftarrow Tr\text{-}path(self) \wedge Tr\text{-}path(employee) \wedge Tr\text{-}select(e2|\ e2=e)$

and, after translating paths and selects, we get the following fragments of formulas:

$boss(B) \wedge \neg aux(E,B)$
$aux(E,B) \leftarrow boss(B) \wedge worksFor(B, E2) \wedge E2=E$

Translation of an OCL invariant. Let path-exp=obj.r1...rn-1.rn. Depending on the comparison operator, we define the translation of an OCL invariant in terms of the translation of the path expression (Tr-path) and the select (Tr-select) as follows:

a) **`context`** `C` **`inv:`**
`obj.r`$_1$`...  r`$_{n-1}$`.r`$_n$` -> select(e| body)-> size() < k`
becomes

$\leftarrow c(C) \wedge Tr\text{-}path(obj.r_1...\ r_{n-1}) \wedge Tr\text{-}path_1(r_n) \wedge Tr\text{-}select_1(e|\ body)$
$\wedge \ldots \wedge Tr\text{-}path_k(r_n) \wedge Tr\text{-}select_k(e|\ body)$

b) **`context`** `C` **`inv:`**
`obj.r`$_1$`...  r`$_{n-1}$`.r`$_n$` -> select(e| body)-> size() > k`
becomes

$\leftarrow c(C) \wedge \neg aux(C)$
$aux(C) \leftarrow Tr\text{-}path(obj.r_1...\ r_{n-1}) \wedge Tr\text{-}path_1(r_n) \wedge Tr\text{-}select_1(e|\ body)$

$\wedge \ldots \wedge$ *Tr-path$_{k+1}$(r_n)* $\wedge$ *Tr-select$_{k+1}$(e| body)*

c) **context** C **inv:**
```
   obj.r1... rn-1.rn -> select(e| body)-> size() = k
```
becomes

$\leftarrow$ *c(C)* $\wedge \neg$*aux(C)*

aux(C) $\leftarrow$ *Tr-path(obj.r$_1$... r$_{n-1}$)* $\wedge$ *Tr-path$_1$(r_n)* $\wedge$ *Tr-select$_1$(e| body)*

$\wedge \ldots \wedge$ *Tr-path$_k$(r_n)* $\wedge$ *Tr-select$_k$(e| body)*

$\leftarrow$ *c(C)* $\wedge$ *Tr-path(obj.r$_1$... r$_{n-1}$)* $\wedge$ *Tr-path$_1$(r_n)* $\wedge$ *Tr-select$_1$(e| body)*

$\wedge \ldots \wedge$ *Tr-path$_{k+1}$(r_n)* $\wedge$ *Tr-select$_{k+1}$(e| body)*

d) **context** C **inv:**
```
   obj.r1... rn-1.rn -> select(e| body)-> size() <> k
```
becomes

$\leftarrow$ *c(C)* $\wedge$ *Tr-path(obj.r$_1$... r$_{n-1}$)* $\wedge$ *Tr-path$_1$(r_n)* $\wedge$ *Tr-select$_1$(e| body)*

$\wedge \ldots \wedge$ *Tr-path$_k$(r_n)* $\wedge$ *Tr-select$_k$(e| body)* $\wedge \neg$*aux(C)*

aux(C) $\leftarrow$ *Tr-path(obj.r$_1$... r$_{n-1}$)* $\wedge$ *Tr-path$_1$(r_n)* $\wedge$ *Tr-select$_1$(e| body)*

$\wedge \ldots \wedge$ *Tr-path$_{k+1}$(r_n)* $\wedge$ *Tr-select$_{k+1}$(e| body)*

Each translation *Tr-path* or *Tr-select* may be performed several times depending on the constant *k*. Each *Tr-path$_i$* or *Tr-select$_i$* expressions refer to the same translations but with different variables for those attributes not coming from `obj.r1... rn-1`. Clearly, the previous formalization becomes much simpler in the usual cases where *k* is 0 or 1.

Intuitively, we may see that the translation of each OCL invariant defines a denial stating that a given situation cannot hold. The first part of each denial includes the logic representation of the path leading to the collection of instances to which the select and the size operations are applied. The second part, the one defined by the subindexes 1 to *k*, is required to guarantee that the cardinality of the set of elements that fulfill the select condition satisfies also the required comparison.

As an example, consider the simplified invariant of constraint *ManagerIsWorker*:
```
context Department inv:
self.worker -> select(e| e=self.manager)-> size() > 0
```
applying the translation b) above we obtain[2]:

$\leftarrow$ *department(D)* $\wedge \neg$*aux(D)*

aux(D) $\leftarrow$ *Tr-path(self)* $\wedge$ *Tr-path(worker)* $\wedge$ *Tr-select(e| e=self.manager)*

and, after translating paths and selects, we get the following formulas which force all departments to have at least one worker who is also a manager.

$\leftarrow$ *department(D)* $\wedge \neg$*aux(D)*

aux(D) $\leftarrow$ *department(D)* $\wedge$ *worksIn(E,D)* $\wedge$ *manages(E2,D)* $\wedge$ *E=E2*

It may also happen that the original expression does not include any OCL operation. Then the constraint has not been simplified and has the form:
```
context C inv: path-exp opComp value
```

[2] Note that, since k=0, the translation of the select and the path must be performed only once.

where *value* is either a constant or another navigation path. The translation of these invariants into logic is:

$\leftarrow c(C) \land Tr\text{-}path(path\text{-}exp) \land Tr\text{-}path(value) \land Obj1\ invOpComp\ Obj2$

where *Obj1* and *Obj2* are the objects obtained as a result of the navigation path(s) *path-exp* and *value*. The new comparison operator *invOpComp* corresponds to the inverse of the original (that is, if *opComp* is > then *invOpComp* is ≤, and so on). Notice that if *value* is a constant then it must not be translated.

3.2 Validation Tests

Our approach to validation is aimed at providing the designer with different kinds of tests that allow him to assess the correctness of the conceptual schema being defined.

We express all the tests in terms of checking the satisfiability of a derived predicate. In this way, for each validation test to be performed, a derived predicate (with its corresponding derivation rule) that formalizes the desired test is defined.

We illustrate our approach using the translation of our example obtained as explained in the previous section (Appendix A).

3.2.1 Is the Structural Schema Right?

Those tests devoted to check the internal correctness of the schema can be automatically defined. Automatic tests can be performed without the designer intervention and allow checking the internal correctness of the schema by means of answering to the question *is the structural schema right?*. Some of them correspond to well known reasoning tasks (such as schema satisfiability), while others correspond to additional properties that can be automatically drawn from the definition of each conceptual schema.

For all of them, the fact that the test is not satisfiable necessarily indicates that the schema has some kind of flaw. The way of repairing the errors found cannot be automatically provided, since there are several alternatives that depend on the domain being modeled and, thus, the designer must decide which solution has to be adopted.

Satisfiability of a Schema

A schema is satisfiable if there is a non-empty state of the IB in which all its integrity constraints are satisfied.

To perform this test, we need to define a derived predicate such that it is true when the schema is satisfiable, that is, if it is possible to populate at least one of its classes and associations. For each class $class_i$ and association $assoc_j$ of the schema, we need to define the following rules:

$sat \leftarrow class_i(X)$

$sat \leftarrow association_j(Y_1,...,Y_n)$

That is, the derived predicate sat will be true if the schema admits an instance of a certain class or association.

To check the satisfiability of the structural schema in Figure 1, we need to formalize this test as follows:

sat ← department(D)
sat ← employee(E)
sat ← boss(E)
sat ← workingTeam(T)
sat ← manages(E,D)
sat ← worksIn(E,D)
sat ← worksFor(E,E2)
sat ← audits(E,T)
sat ← member(M,E,T)
sat ← hasRecruited(M,M2)

As explained in section 1.3.1, this schema is satisfiable, as shown by the following sample instantiation:

{employee(john), worksIn(john,sales), department(sales), manages(john,sales)}

In order to have a valid instance of *Employee*, we also need at least an instance of the association *WorksIn* due to the cardinality constraint 1 of *working-dep*. This requires an instance of *Department* which, in turn, needs a *manager* due to the cardinality constraint 1 in the association *Manages.* According to the constraint *ManagerIsWorker*, this manager must be one of the employees working in the department.

There is no need to populate the association *WorksFor* or the class *WorkingTeam*, since they are not required in order to have a valid instance of the schema.

Other solutions may exist, consisting in populating other classes or associations that are not populated in the sample instantiation obtained. However, a single solution is required to prove the satisfiability of the schema.

Liveliness of a Class or Association

Even if a schema is satisfiable, it may turn out that some class or association is empty in every valid state. Liveliness of classes or associations determines if a certain class or association can have at least one instance.

In this case, the derived predicate that formalizes this test must be defined as follows, where *element(X_1,...,X_n)* is the class or association for which liveliness is to be tested:

livelyElement ← element(X_1,...,X_n)

Classes *Employee* and *Department* of the schema in Figure 1, as well as the associations *WorksIn* and *Manages* are lively, since there exists at least a state satisfying all the constraints in which they are populated. This is proven by the same sample instantiation obtained when testing the satisfiability of the schema.

Let us see if the rest of classes and associations of the schema are lively too. For instance, to test the liveliness of the association *WorksFor*, we should test:

livelyWorksFor ← worksFor(X,Y)

As a result we obtain that there is at least a state in which it is not empty, which consists in an employee that works for another one, both of them working in the same department and the superior employee being the manager of the department. For example *{worksFor(mary, john), employee(mary), employee(john), worksIn(mary, sales), worksIn(john, sales), department(sales), manages(mary, sales)}*.

In contrast, if we reason on the liveliness of *Boss*, we see that to have an instance of *Boss* we need that he or she is the superior of all the workers of the department managed by that boss (constraint *SuperiorOfAllWorkers*). A state satisfying this condition would be one in which a boss does not manage any department, but this is prevented by the constraint *BossIsManager*. Another way of satisfying this condition would be a state in which the department managed by the boss does not have workers, but the constraint *ManagerIsWorker* forces each department to have at least a worker, its manager. Then, the only option is to have a boss that manages a department and that all the workers of that department (including the boss himself) are subordinates of that boss. But this is impossible according to the constraint *ManagerHasNoSuperior* and, therefore, the class *Boss* is not lively, which may not be easy to see at first sight.

When eliminating either of these constraints, a state fulfilling the rest of conditions can be found and *Boss* becomes lively. For instance, if we remove *BossIsManager* we can obtain a state in which the boss works in a department but is not a manager. Since every department must have a manager, it is another employee the one who manages the department: *{boss(john), employee(john), worksIn(john,sales), department(sales), manages(mary,sales), worksIn(mary,sales), employee(mary)}*.

Regarding the liveliness of the class *WorkingTeam*, it can be seen that a working team requires at least three employees (an inspector and a member, which cannot be the inspector according to the constraint *InspectorNotMember*, and a third employee that recruits the member in order to satisfy the constraints *OneRecruited* and *NotSelfRecruited*). Thus, a valid instantiation of this class, together with the required associations *Member, Audits* and *HasRecruited* is:

{ workingTeam(team1), audits(mary,team1), member(m1,john,team1),

member(m2,peter,team1), hasRecruited(john,peter), employee(mary), employee(john),

employee(peter), worksIn(mary,sales), worksIn(john, sales), worksIn(peter, sales),

manages(mary, sales), department(sales)}

Redundancy of an Integrity Constraint

An integrity constraint is redundant if integrity does not depend on it, that is, if the states of the IB it does not allow are already prevented by the rest of constraints.

The fact that a constraint is redundant does not necessarily imply that the schema is erroneous. Nevertheless, redundancies make the modifiability of the schema more difficult and error prone since, when a requirement changes, the designer must take care of modifying every constraint affected. This can also lead to inconsistencies if the corresponding requirement is not modified in all the constraints in which it appears. Additionally, redundant constraints may derive in inefficiencies of the final product if they are directly translated into an implementation.

Intuitively, an integrity constraint is redundant if there is no IB state in which it is violated. Hence, proving redundancy could involve looking for an infinite number of states. Therefore, it is more appropriate to search for the lack of redundancy. A constraint is not redundant if there is at least an IB state in which it is violated.

Thus, to check whether a constraint *ic* of a schema is redundant, it must be removed from the schema and test the satisfiability of the following predicate. This predicate may be automatically defined for each constraint *ic* in the logic formalization of the schema, which includes the textual, graphical and implicit constraints:

nonRedundantIC ← ic

If this predicate is satisfiable, then the constraint *ic* is not redundant. Otherwise, *ic* is redundant.

For instance, we can test whether the constraint *BossHasNoSuperior* is not redundant:

nonRedundantBossHasNoSuperior ← boss(B) ∧ worksFor(S,B)

That is, we can try to build a state where *nonRedundantBossHasNoSuperior* holds. Intuitively, this consists in a state in which the constraint *BossHasNoSuperior* is violated while the rest are not, that is, a state in which there is a boss that has a superior. However, this is not possible since a boss must be the manager of some department (constraint *BossIsManager*). Additionally, the constraint *ManagerHasNoSuperior* prevents the manager of a department from having a superior and thus, the constraint *BossHasNoSuperior* can never be violated and can be eliminated from the schema.

There are other redundancies in this example, for instance between a graphical and an OCL constraint. In particular, *ManagerIsWorker* implies that a department must have at least a worker, since it must have a manager and he must be one of its workers. At the same time, the cardinality constraint 1..* of *Employee* in the association *WorksIn* already forces the existence of at least a worker in each department. This redundancy is detected by the following test:

nonRedundantCardinality ← department(D) ∧ ¬oneWorker(D)

oneWorker(D) ← worksIn(E,D)

Similarly, the constraint 1..* of *Employee* in the association *Member* is entailed by the fact that a working team must have at least one member recruited by another member of the same team (constraint *OneRecruited*).

However, redundancies involving cardinalities may not be eliminated, since generalizing them may damage the understandability of the schema.

Automatically Generated Tests

In addition to the typical tests just reported, there are other situations in which it is clear that there is some mistake in the definition of the schema, despite having successfully passed all the previous tests. These situations correspond to states that are potentially admitted by the class diagram but, however, are prevented by the constraints.

Some of such tests can be drawn from the concrete schema to be validated, and are an original contribution of this thesis, following the ideas suggested by model-based testing approaches (Utting and Legeard 2006). Note, however, that we can already determine these situations at the conceptual schema level while, in general, model-based testing requires an implementation of the software system to execute the tests.

For instance, a cardinality constraint may be defined with an upper bound greater than 1 but, however, the rest of constraints force that a single instance can be related to each instance at the other end. This may mean that some constraint should be removed or that the cardinality constraint is too general and should be modified. A set of additional possible tests is defined in the following.

Test 1: Maximum cardinality. This test shows us whether it is possible to have as many instances as specified by the maximum cardinality *max* associated to the same instances at the other association ends. The general formalization for n-ary associations is:

$$\text{maxCardAssoc} \leftarrow \text{assoc}([A_1], P_1,\ldots,P_{i1},\ldots,P_n) \wedge \ldots \wedge \text{assoc}([A_{max}],P_1,\ldots,P_{imax},\ldots,P_n) \wedge P_{ij}{<>}P_{ik}$$

for each $j<>k$ between 1 and *max*, where the OID *A* is needed when *Assoc* is an association class, $P_1,\ldots,P_n$ are the participants and P_i is the participant with the cardinality to be checked.

Test 2: Minimum cardinality. When we have a cardinality constraint with a lower bound = 0, this test shows whether it is possible that an instance(s) at the other end(s) does not participate in the association

$$\text{minCardAssoc} \leftarrow \text{assoc}([A_1],P_1,\ldots,P_{i1},\ldots,P_n) \wedge \ldots \wedge \text{assoc}([A_{min}],P_1,\ldots,P_{imin},\ldots,P_n) \wedge \neg\text{assoc}([A_{min+1}], P_1,\ldots,P_{imin+1},\ldots,P_n) \wedge P_{ij}{<>}P_{ik}$$

for each $j<>k$ between 1 and *min+1* representing a participant of *Assoc*, where P_i is the participant with minimum cardinality to be checked.

Test 3: Incomplete hierarchy. If hierarchy is specified as {incomplete}, we can check if it is really such, that is, if an instance may only belong to the superclass. Otherwise, it should be specified as {complete}, or some other constraint should be removed or modified.

$$\text{incompleteSuperclass} \leftarrow \text{superclass}(X) \wedge \neg\text{subclass}_1(X) \wedge \ldots \wedge \neg\text{subclass}_n(X)$$

Test 4: Overlapping hierarchy. If hierarchy is specified as {overlapping}, we can check if it is really such, that is, if an instance may belong more than one subclass simultaneously. Otherwise, it should be specified as {disjoint}, or some other constraint should be removed or modified. For each pair i, j of subclasses:

$$\text{overlappingSuperclass} \leftarrow \text{subclass}_i(X) \wedge \text{subclass}_j(X)$$

Note that all these tests correspond to situations admitted by the schema and do not imply that the schema is ill-defined, since an IB satisfying all the graphical and textual constraints can be found. However, they warn the designer that some graphical constraints should be strengthened or that some textual constraints should be weakened, so that the graphical part corresponds to the textual one.

3.2.2 Is It the Right Structural Schema?

Although there are some tests that can be executed without the intervention of the designer, as we have seen in the previous subsection, the validation process is not fully formalizable. The reason is that validating a conceptual schema consists in checking if it correctly specifies the requirements, which are not usually formalized. This means that it is desirable to help the designer to analyze the schema so that he can decide whether it represents the intended domain.

Thus, during the validation process and once the internal correctness is ensured by the previous tests, the designer will need to check the external correctness of the schema. Again, this task can be partly automated by generating additional tests that check other kinds of properties.

After the hints given by the automatically generated tests, the designer may finalize the validation process by defining his own tests and comparing the obtained results to those expected according to the requirements, in order to do an exhaustive validation of the schema.

Automatically Generated Tests

There are some validation tests that can be automatically drawn from the schema under validation, that help the designer to find potentially undesirable situations. For instance, when a schema contains a recursive association, such as *WorksFor* and *HasRecruited* in our example, it may be interesting to test whether it is reflexive, that is, if an instance of the association can relate the same instance of the associated class. This is a useful test since, in many cases, recursive associations are not reflexive and, thus, a textual constraint is needed in order to prevent that an instance is associated to itself, which the designer may have overlooked.

In our schema, we could test whether an employee can be superior of himself by means of the association *WorksFor*, using the following test:

reflexiveWorksFor ← worksFor(X,X)

If the predicate *reflexiveWorksFor* is satisfiable, it means that an employee can work for himself in our schema. If we execute this test in our schema, the result is as follows: *{worksFor(mary,mary), employee(mary), department(sales), worksIn(mary,sales), employee(john), manages(john,sales), worksIn(john, sales)}*

This sample instantiation proves that an employee may work for himself, as long as he is not a manager, due to the constraint *ManagerHasNoSuperior.* Now the designer should decide whether this correctly represents the domain. If not, the following constraint should be added to the schema:

```
context Employee inv: self.superior <> self
```

We have another recursive association in our schema, so we will also test whether it is reflexive:

reflexiveHasRecruited ← hasRecruited(X,X)

This time, the predicate *reflexiveHasRecruited* is not satisfiable, which means that the association is not reflexive. The reason is that the constraint *NotSelfRecruited* prevents a member from recruiting himself.

There are other typical constraints that usually appear in conceptual schemas, which have been formalized in (Costal, Gómez et al. 2008). Following the classification of constraints provided in this paper, we will formalize automatic tests that can be useful to check whether a constraint is missing in our schema, knowing that it is a constraint that usually appears in practice. In the following, we formalize some tests that can provide interesting information to the designer. Additional tests of interest can be defined in a similar way.

Test 5: Irreflexive constraint. If a recursive association can relate the same instance, then an irreflexive constraint may be missing. This test can be defined for each recursive association in the schema, as we have already seen in the example.

reflexiveAssoc ← assoc([A],X,X)

where *A* is the OID needed in case *assoc* is an association class.

If the predicate is satisfiable and the designer decides that the constraint is missing, then the following constraint, as formalized in (Costal, Gómez et al. 2008), should be added to the schema:

```
context A inv: self.r1 -> excludes(self)
```

where *r1* is one of the roles of the recursive association *assoc*. This information can also be automatically provided to the designer.

The same can be done for the following tests:

Test 6: Identifier constraint. If a class admits several valid instances with the same value in all its attributes, then an identifier constraint may be needed to state the set of attributes that uniquely identify each instance. Note that this constraint may not be required in certain domains, since objects may not be externally distinguishable, so the designer must decide whether he wants to add the constraint. This test can be generated for each class of the schema, as long as it is not an association class or a subclass, since in these cases their identifiers are already known.

non-uniqueClass ← class(X) ∧ class(Y) ∧ X<>Y ∧ attr1(X,V) ∧ attr1(Y,V) ∧ ...
... ∧ attrn(X,W) ∧ attrn(Y,W)

Applying this test to the classes in our schema, we can see that the predicate non-uniqueClass is not satisfiable by any of them, since all the classes have their identifiers.

When the same two classes are related by two different associations, some kind of restriction may be imposed on the instances found at both ends. Let C and D be the classes related by the associations R and S. Then the following tests can be defined:

Test 7: Path inclusion. If c is an instance of C, the set of instances of D related to c by means of R is a subset of the set of instances of D related to c by means of S, or the other way around.

not-inclusionRS ← r([R],C,D) ∧ ¬inS(C,D)
inS(C,D) ← s([S],C,D)

not-inclusionSR ← s([S],C,D) ∧ ¬inR(C,D)
inR(C,D) ← ([R],C,D)

where *R* and *S* are the OIDs needed in case the corresponding association is an association class. Note that the derivation rules will not be needed when OIDs are not present, so the rules will be simpler in this case:

not-inclusionRS ← r(C,D) ∧ ¬s(C,D)

not-inclusionSR ← s(C,D) ∧ ¬r(C,D)

These predicates are satisfiable when an instance of one of the associations that does not belong to the other one can exist. This will help the designer to decide whether a path inclusion constraint is needed and, in case it is, in which direction.

In our schema, we can define this test for the associations *WorksIn* and *Manages*, and for *Audits* and *Member*.

not-inclusionWorksInManages ← WorksIn(E,D) ∧ ¬Manages(E,D)

In this case the predicate is satisfiable as proved by the following example: *{worksIn(mary, sales), employee(mary), department(sales), employee(john), manages(john, sales), worksIn(john, sales)}*, which shows that Mary works in Sales, but the manager of this department is another employee.

non-inclusionManagesWorksIn ← Manages(E,D) ∧ ¬WorksIn(E,D)

This time the predicate is not satisfiable, which means that it is impossible that an employee manages a department without working in it. This is due to the constraint *ManagerIsWorker* which, in fact, defines an inclusion constraint between these two associations, so the designer will probably find that the schema is correct in this sense.

Regarding the other pair of associations we can perform the tests:

not-inclusionAuditsMember ← audits(E,T) ∧ ¬inMember(E,T)
inMember(E,T) ← member(M,E,T)

not-inclusionMemberAudits ← member(M,E,T) ∧ ¬audits(E,T)

In this case, both predicates are satisfiable, so the designer may want to add some constraint, if the two paths are not restricted in any other way. Before making the decision, we will see the following kind of path comparison constraint.

Test 8: Path exclusion. If c is an instance of C, the set of instances of D related to c by means of R excludes the set of instances of D related to c by means of S. In this case, only one test is needed:

not-exclusionRS ← r([R],C,D) ∧ s([S],C,D)

It does not make sense to perform this test on *WorksIn* and *Manages,* since they are already constrained by a path inclusion. Thus, we will test *Audits* and *Member*:

not-exclusionAuditsMember ← audits(E,T) ∧ member(M,E,T)

This predicate is not satisfiable due to the constraint *InspectorNotMember*, which is in fact an exclusion constraint. Thus, this constraint is not missing, neither an inclusion on these two paths.

User-defined Tests

As a last step in the validation process, the designer can define his own tests in the form of derived predicates, as the ones we have generated automatically. In this way, he will be able to cover those requirements that cannot be taken into account by the automatic tests. For instance, an interesting question could be "Can an employee's salary be lower than the minimum salary of its department?". Obviously, the answer should be negative, but we must check if this is so according to the schema. The designer will have to formalize the derivation rule:

correctSalary ← employeeSalary(E,S) ∧ worksIn(E,D) ∧
departmentMinSalary(D,M) ∧ S<M

A possible result for this test could be:

{employee(mary), employeeSalary(mary,500), worksIn(mary, it), manages(mary, it), departmentMinSalary(it, 1000)}

which shows that an employee's salary can be lower than his department's minimum salary according to the schema. This means that a textual constraint to restrict the salary of employees must be defined.

Similarly, one could test the schema by explicitly giving a sample instantiation to be checked. A question we could ask is whether a given employee, in this case Mary, can recruit himself in some team:

selfRecruiting ← member(m1, mary, team1) ∧ member(m2, mary, team2) ∧
hasRecruited(m1, m2)

A possible answer is:

{member(m1, mary, team1), member(m2, mary, team2), hasRecruited(m1, m2), (1)

employee(mary), team(team1), team(team2), (2)

department(sales), worksIn(mary, sales), manages(mary, sales), (3)

employee(susan), audits(susan, team1), audits(susan,team2), (4)

worksIn(susan, sales), (5)

employee(peter), worksIn(peter, sales), (6)

member(m3, peter, team1), member(m4, peter, team2), (7)

hasRecruited(m1, m3), hasRecruited(m2, m4)} (8)

Since the answer includes the definition of the derived predicate to be tested (1), it shows that this situation is accepted by the schema. In addition to the given instances, additional ones have had to be added in order to satisfy all the constraints. In particular, the referential constraints (2), the cardinality constraints of *working-dep* and *manager* (3), where the same employee Mary can be used according to the constraint *ManagerIsWorker*, the cardinality constraint of *inspector* (4), that forces to create at least a new employee that is not a member of the teams according to the constraint *InspectorNotMember*, which must work in a department according to the cardinality of *working-dep* (5), and yet another employee with its department (6), which must be created to satisfy the constraints *OneRecruited* and *NotSelfRecruited*, ensuring that each team has at least a member recruited by another member of the same team (7) and (8).

However, this situation is possibly what the constraint *NotSelfRecruited* intended to prevent. Although the constraint is present in the schema, it is not correctly defined in OCL, and this test has been useful to find this erroneous formalization. The reason is that the constraint is defined on members, stating that a member cannot be recruited by the same instance of member. However, two different instances of member can correspond to the same employee, and it should be the employee the one that cannot recruit himself. Thus, the correct definition in OCL should be:

```
context Member inv: self.recruiter.employee <> self.employee
```

Note that providing explicit values when formalizing a test is only useful when it is a counterexample, that is, when we want to check whether a certain situation is accepted by the schema when it should be not. In other words, if the previous test *selfRecruiting* had failed, this would not guarantee that no employee can recruit himself, since the test only ensures it for the employee *mary*.

4

A Reasoning Procedure for the Structural Schema

As we have seen in the previous chapter, the validation of a schema consists in the definition and execution of a set of tests in terms of the logic formalization of the schema. Since OCL in its full expressiveness leads to undecidability, the tests performed on the result of its translation into logic may not terminate, because the constraints can force the schema to have models with an infinite number of instances.

At this point, knowing that the problem we want to solve is undecidable, some decisions must be taken. On the one hand, we have the option to limit the values that instances can take, or the total number of instances that a model can have. However, this implies renouncing to completeness, which we do not find acceptable.

Another possibility is to restrict the expressiveness of the input schema, for example disallowing general constraints, in order to ensure decidability. This is neither a valid option for us, since our aim is to validate a UML schema with OCL constraints, as expressive as possible.

Finally, the third possibility consists in allowing general constraints, but without guaranteeing termination in order to be complete. This option seems theoretically better, but a procedure that may not terminate is not very useful in practice.

Thus, since we do not want to renounce to completeness, neither to a high expressiveness of the constraints, and we want a procedure that is applicable in practice, we do not strictly follow neither of these directions. Once we have translated the UML structural schema with its OCL constraints into logic, and previously to reasoning on it, we determine whether it is possible to perform any test with the guarantee that it will terminate. That is, we admit an expressiveness of the constraints for which decidability is not guaranteed, but then we determine whether termination can be ensured for each particular schema defined.

This is possible since the logic representation of the schemas obtained from the translation of a UML schema with its OCL constraints has a specific structure. In particular, it has only one level of derivation in most cases, that is, derived predicates are never defined by other derived predicates. To guarantee that there is exactly one level of derivation in the schema, and thus be able to apply our approach, we impose a restriction on the OCL expressions that the schema can contain. In particular, although the OCL constraints can include the operations `includes`, `includesAll`, `notEmpty`, `exists` and `one`, these specific operations (or their equivalent expressions) cannot be recursively combined in an OCL expression.

For instance, the OCL constraints included in our example:

```
context Boss inv SuperiorOfAllWorkers:
    self.employee->includesAll(self.managed-dep.worker)

context WorkingTeam inv OneRecruited:
    self.member->exists(m|m.recruiter.workingTeam=self)
```

are allowed, since they do not have nested combinations of the operations mentioned. Also, other constraints combining OCL operations are allowed. For example, the following constraint meaning that each working team must have at least a member that is not a manager, is accepted by our method:

```
context WorkingTeam inv:
    self.employee->exists(e|e.managed-dep->isEmpty())
```

Or the following one, stating that there is at least a working team entirely composed by bosses:

```
context WorkingTeam inv:
    WorkingTeam.allInstances()->exists(t|t.employee->
    forall(e|e.oclIsTypeOf(Boss)))
```

Or, finally, this other constraint stating that all the employees that manage some deptarment have recruited some member of a working team:

```
context Department inv:
    Department.allInstances().manager->forall(e|e.member->
    exists(m|m.recruiter.employee=e))
```

However, the following ones are not accepted in our approach, since they contain forbidden combinations of OCL operations.

```
context WorkingTeam inv:
    self.employee->exists(e|e.managed-dep->notEmpty())

context WorkingTeam inv:
    WorkingTeam.allInstances()->exists(t|t.employee->
    exists(e|e.oclIsTypeOf(Boss)))
```

Our approach to check the decidability of reasoning on a given schema that satisfies this restrictions is detailed in section 4.1.

After determining the decidability, any theorem prover or reasoning method that is able to deal with our logic formulas can be used. If the conditions of termination are satisfied, then it is guaranteed that any reasoning task performed on the schema will terminate. Otherwise, it may happen that the reasoning tasks performed do not provide a result in finite time.

However, despite being able to use an existing procedure to reason on the schema, we provide a new procedure that always terminates and works more efficiently than in the general case. This procedure takes advantage both of the fact that termination is guaranteed, and of the known structure of the logic formalization obtained from the translation. The reasoning procedure is proposed in section 4.2.

4.1 Dealing with Decidability

Since our approach to validate a schema consists in building a sample state of the IB that satisfies a certain property, we can say that this task will terminate as long as all the sates of the IB admitted by the schema are finite. Then, any sample IB that can be built to prove a property will have a finite number of instances and, thus, the construction will always terminate.

In this section we present our approach to determine whether any reasoning task performed on a schema will terminate. We provide a set of conditions that guarantee that the schema does not have any infinite model. In this way, any theorem prover or reasoning method could be used knowing that any reasoning task performed on the schema will terminate.

The constraints defined in the schema (both the ones obtained from the translation of the UML class diagram, and those explicitly defined by the designer as OCL constraints) are the ones that cause a constructive reasoner to add new facts to the sample IB under construction. It may happen that a constraint is violated as a result of repairing another one, so we must analyze the interactions between constraints in order to determine whether any sequence of violations is finite.

To do this, we obtain a graph from the set of constraints of the schema that shows the dependencies between them. We formalize the construction of the dependency graph for a given schema in section 4.1.1, and in section 4.1.2 we explain how to use this graph to determine the absence of infinite models in a given schema.

Throughout this section we will use the schema in Figures 6 and 7, corresponding to the subset of the running example regarding working teams. For the sake of simplicity, we have omitted all the attributes.

As can be seen in Figure 6, each *WorkingTeam* is related to an *Employee,* which is its *inspector*, and to a set of employees that are its *Members*. Each member has been recruited at most by another member (the *recruiter*). The textual constraints in Figure 7 impose that the inspector of a working team cannot be one of its members, and that each team must have at least one member recruited by another member of the same team different from himself.

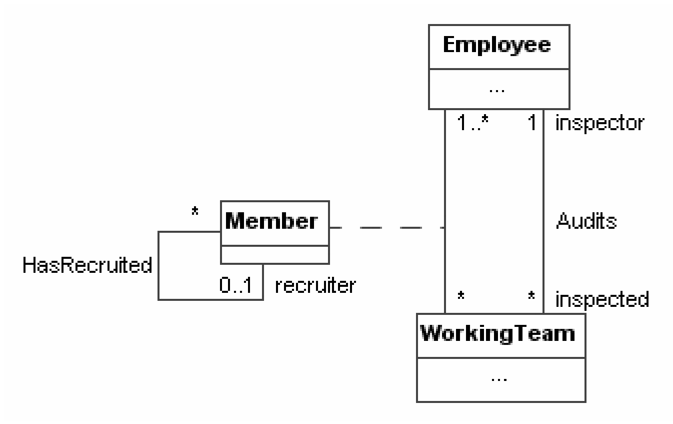


Fig. 6. Subset of the running example showing only employees and working teams

```
Integrity constraints

1.   context WorkingTeam inv InspectorNotMember:
     self.employee->excludes(self.inspector)

2.   context Member inv NotSelfRecruited:
     self.recruiter<>self

3.   context WorkingTeam inv OneRecruited:
     self.member->exists(m|m.recruiter.workingTeam=self)
```

Fig. 7. Integrity constraints corresponding to the fragment of the schema in Figure 6.

The translation of this schema into logic results in the following rules, which have been obtained according to the translation defined in the previous section. For the sake of understandability, we have simplified the logic representation of the schema obtained from the automatic translation. In particular, we have omitted the implicit constraints regarding OIDs, since they do not affect the results of the decidability analysis. Additionally, we have simplified some of the logic expressions obtained from the automatic translation of OCL constraints, in such a way that the resulting expression is equivalent to the original one.

For instance, the translation of the OCL constraint *InspectorNot Member* provided by our implementation is:

$\leftarrow$ workingTeam(T) $\wedge$ audits(E,T) $\wedge$ member(M,E,T)

instead of:

$\leftarrow$ audits(E,T) $\wedge$ member(M,E,T)

as can be seen in constraint 12 below. In our example, these constraints are equivalent, since the term *T* of *audits* and *member* will always correspond to a *workingTeam* due to constraints 1 and 3. Thus, the literal *workingTeam(T)* is not needed in the body of this constraint.

Some of the rules use derived predicates in their definitions, which are needed in order to make all of the conditions *safe* (see the *Basic Concepts* chapter). Conditions 1 to 6 correspond to the referential constraints of associations. In this case, since all associations are binary, two such constraints are needed for each of them (one constraint for each association end). Condition 7 specifies that there cannot be two instances of *Member* relating the same *Employee* and *WorkingTeam,* which is an implicit constraint of association classes. Conditions 8 to 11 are the cardinality constraints of associations (the upper and lower bounds of *inspector* in the association *Audits*, the lower bound of *Employee* in the association *Participant*, and the upper bound of *recruiter* in *HasRecruited*). Finally, conditions 12 to 14 are the translation of the textual OCL constraints. We have ommitted those constraints needed to guarantee the uniqueness of OIDs, since they do not affect the study of decidability, in order to keep the graph as simple as possible.

1. ← audits(E,T) ∧ ¬ workingTeam(T)
2. ← audits(E,T) ∧ ¬ employee(E)
3. ← member(M,E,T) ∧ ¬ workingTeam(T)
4. ← member(M,E,T) ∧ ¬ employee(E)
5. ← hasRecruited(M,R) ∧ ¬isMember(M)

   isMember(M) ← member(M,E,T)
6. ← hasRecruited(M,R) ∧¬isMember(R)
7. ← member(M,E,T) ∧ member(M2,E,T) ∧ M≠M2
8. ← workingTeam(T) ∧ ¬oneInspector(T)

   oneInspector (T) ← audits(E,T)
9. ← audits(E,T) ∧ audits(E2,T) ∧ E≠E2
10. ← workingTeam(T) ∧ ¬oneMember(T)

    oneMember(T) ← member(M,E,T)
11. ← hasRecruited(M,R) ∧ hasRecruited(M2,R) ∧ M≠M2
12. ← audits(E,T) ∧ member(M,E,T)
13. ← HasRecruited(M,M)
14. ← workingTeam(T) ∧¬oneRecruited(T)

    oneRecruited(T) ← member(M,E,T) ∧ hasRecruited(M,R) ∧ member(R,E2,T)

4.1.1 The Dependency Graph

As seen in chapter 2, a condition consists of a set of positive literals, a set of negative literals and a set of built-in literals. Positive literals are the ones that may violate the constraint, whereas negative literals repair the constraint in case it is violated or, in other words, avoid violation in case that all the positive literals hold in the EDB.

Definition 4.1. A literal *$p(\bar{X})$* is a *potential violation* of a condition *ic* if it appears positively in its body. We denote by *V(ic)* the set of potential violations of the condition *ic*.

For example, *V(ic1)* = {*audits(E,T)*} since the existence of a fact *audits(a,b)* in the EDB causes the violation of *ic1* if the EDB does not contain the fact *workingTeam(b)*.

Definition 4.2. Given a condition *ic*, there is a repair *$R_i(ic)$* for each negative literal $\neg L_i$ in the body of *ic*. If L_i is base then $R_i(ic) = \{L_i\}$, otherwise $R_i(ic) = \{ p_1(\bar{X}_1),\ldots, p_n(\bar{X}_n) \}$, where each $p_j(\bar{X}_j)$ is a literal that appears positively in the body of the derivation rule that defines L_i.

Each repair of a condition gives an alternative way to avoid its violation. A condition has alternative repairs if it is of the form:

$\leftarrow p(\bar{X}_1) \wedge \neg q(\bar{X}_2) \wedge \neg r(\bar{X}_3)$

where $R_1 = \{ q(\bar{X}_2)\}$ and $R_2 = \{ r(\bar{X}_3)\}$, or of the form:

$\leftarrow p(\bar{X}_1) \wedge \neg r(\bar{X}_2)$

$r(\bar{X}_3) \leftarrow s(\bar{X}_4) \wedge t(\bar{X}_5)$

$r(\bar{X}_3) \leftarrow v(\bar{X}_6)$

where $R_1 = \{ s(\bar{X}_4), t(\bar{X}_5)\}$ and $R_2 = \{v(\bar{X}_6)\}$

In our example, all conditions have a single repair. For instance, the repair of condition 1 is *R(ic1)* = {*workingTeam(T)*}, and the repair of condition 14 is *R(ic14)* = {*member(M,E,T), hasRecruited(M,R), member(R,E2,T)*}. There are some constraints, such as *ic7*, that cannot be repaired once they are violated, unless their potential violations are removed from the EDB.

In the proposed approach, the set *IC* of constraints of a schema is associated with a directed graph *G*, that we call *dependency graph*. This graph shows, for each condition ic_i of the schema, which conditions may be violated as a result of each possible repair of ic_i.

Definition 4.3. A dependency graph *G* is a graph such that each vertex corresponds to a condition ic_i of the schema. There is an arc from ic_i to ic_j, labeled $R_k(ic_i)$, if there exists a predicate *p* such that $p(\bar{X}) \in R_k(ic_i)$ and $p(\bar{Y}) \in V(ic_j)$.

Note that *G* is sometimes a multigraph, since two different repairs of a condition ic_i may lead to the violation of a same other condition ic_j.

Figure 8 depicts the dependency graph built from the conditions of our example. For the sake of clarity, conditions 5 and 6 have been collapsed in a single vertex, since the predicates belonging to their sets of potential violations and repairs coincide. For instance, it can be seen that the repair of conditions *ic5* and *ic6* can violate *ic4, ic7, ic3* and *ic12*, since the predicate *Member*, which is a repair for conditions *ic5* and *ic6*, belongs to their sets of potential violations.

However, not all the arcs that appear in the graph represent the violation of the condition in the terminal vertex when repairing the initial one. Sometimes, the existence of an arc means the opposite: that the repair of the condition in the initial vertex guarantees the non-violation of the condition in the terminal vertex. We say this kind of arcs are *superfluous*. Examples of superfluous arcs, which are depicted with dashed lines in Figure 8, are the ones between conditions *ic1* and *ic8*. When *ic1* is violated due to the insertion of a fact *audits(a,b)* in the EDB, the insertion of the corresponding repair *workingTeam(b)* guarantees that *ic8* is fulfilled. Similarly, when the first to be violated is *ic8* because of the presence of a fact *workingTeam(b)*, the violation is repaired by the insertion of *audits(a,b)*, which guarantees the satisfaction of condition *ic1*.

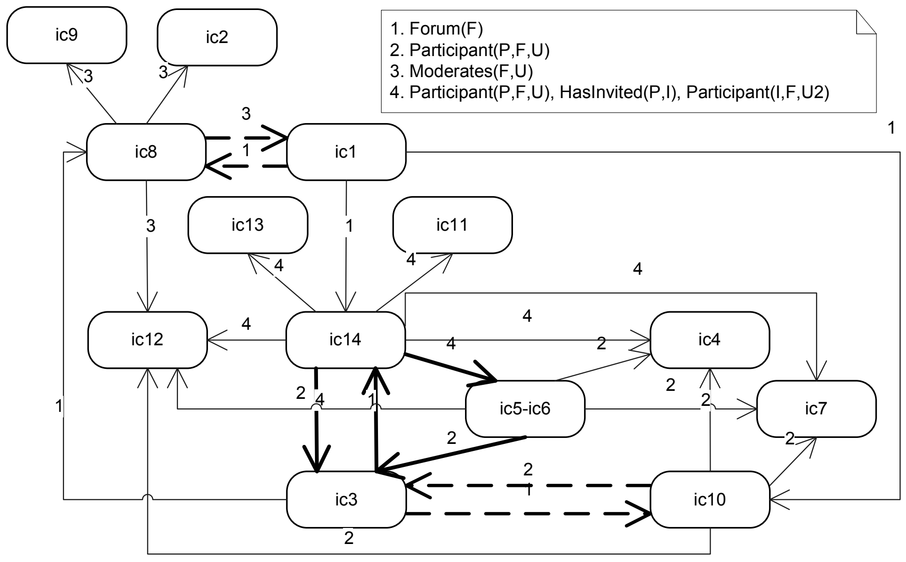


Fig. 8. Dependency graph. Superfluous arcs are dashed, and cycles are highlighted.

Formally, an arc r_i from the constraint ic_i to ic_j is superfluous when $V(ic_j) = r_i\theta$ and there is some repair $R_k(ic_j)$ such that $R_k(ic_j) = V(ic_i)\theta$, where θ is a unifier of the sets $V(ic_i) \cup r_i$ and $V(ic_j) \cup R_k(ic_j)$ that assigns a different term to each distinct variable. This guarantees that ic_j is never violated after the repair of ic_i, since although the facts added by r_i potentially violate ic_j , this condition is always satisfied because its repair also belongs to the EDB. Thus, once these superfluous arcs are identified, they can be left aside since they indicate the ending of any sequence of repairs.

Let $C = (ic_1\ r_1\ \ldots\ ic_n\ r_n\ ic_{n+1} = ic_1)$ be an alternating sequence of vertices and arcs that define a cycle in a dependency graph G. The existence of C implies that the repair r_i of a condition ic_i may violate other conditions whose repairs could violate ic_i again.

As can be seen in Figure 8, there are two cycles in our dependency graph, defined by the conditions (*ic3 ic14*) and (*ic3 ic14 ic5/6*). Since each constraint has a single repair, an enumeration of vertices suffices to identify each cycle. Note that the existence of superfluous arcs significantly reduces the number of cycles in the dependency graph.

4.1.2 Determining the Decidability

Our approach to reasoning is aimed at constructing a database state which shows that a certain property holds. That is, a sample EDB where both the particular condition that defines the reasoning task and all the integrity constraints in the schema are satisfied. Therefore, our approach requires to perform integrity maintenance when trying to build such a sample EDB.

It can be seen that the constraints that form a cycle in a dependency graph are the only reason for the existence of infinite models. Clearly, a condition that does not belong to a cycle will not cause an infinite sequence of repairs, since it will not be violated again when it has been maintained once for a certain set of facts. On the contrary, constraints that belong to cycles can be violated a potentially infinite number of times, since once they have been maintained, the same facts inserted by the repairs may cause new violations and new repairs, which can result in an infinite model. Then, if we can identify which are the cycles that do not cause an infinite sequence of violations, we can determine whether a schema is suitable to perform any reasoning task in finite time.

In this section we study the cycles of the dependency graph to ensure that the process of integrity maintenance does not loop forever. To do this, we provide a set of theorems that allow to discard the presence of infinite models in the constraints that define each cycle. When all the models of a cycle of constraints are finite we call it a *finite cycle*.

A first condition that guarantees that a cycle is finite is that it includes a constraint whose violation requires facts that are not inserted in the EDB by some repair in the same cycle. This implies that the cycle will not lead to an infinite sequence of repairs, since there is necessarily a condition in the cycle that will not be violated at some time. This is formalized in theorem 4.4.

Theorem 4.4. *A cycle* $C = (ic_1\ r_1 \ldots ic_n\ r_n\ ic_{n+1} = ic_1)$ *is a* finite cycle *if*

$$\bigcup_{i=1}^{n}\left(\bigcup_{p(\bar{X})\in r_i} p\right) \subset \bigcup_{i=1}^{n}\left(\bigcup_{q(\bar{Y})\in V(ic_i)} q\right)$$

Intuitively, since the union of repairs of the conditions in the cycle is a proper subset of the union of potential violations, at least one potential violation of a constraint ic_j in the cycle is an EDB predicate which is not updated during maintenance of the rest of the constraints. Therefore, since the set of facts in the sample EDB at the beginning of the process is finite, ic_j may always be violated only a finite number of times.

An example is the following set of constraints, which define a cycle since the repair of the first one is a potential violation of the second, and viceversa:

$\leftarrow$ p(X) $\wedge$ q(X) $\wedge$ ¬r(X)

$\leftarrow$ r(X) $\wedge$ ¬aux(X)

aux(X) $\leftarrow$ p(Y) $\wedge$ Y $\neq$ X

In this example, the potential violation *q(X)* in the first condition is not added by the repair of the second one. Thus, even when the first constraint is violated because $p(X) \wedge q(X)$ holds in the EDB for some *X*, the repairs of the second condition may only lead to new violations of the first one a finite number of times (one for each fact *q(a)* contained in the initial EDB when the process of maintaining the previous costraints started).

A cycle may be finite although it does not satisfy the previous condition. Examples can be found such that all the facts that are potential violations are created inside the cycle and, however, the cycle is not potentially infinite. An example is the cycle (*ic3 ic14*), which does not satisfy the previous condition but is finitely satisfiable. For instance, when a *member(a,b,c)* is added to the EDB, *ic3* requires the insertion of *workingTeam(c)* which, in turn, violates *ic14*. In order to repair this violation, the facts *member(a2,b2,c), hasRecruited(a2,a3)* and *member(a3,b3,c)* may be inserted, but they will never violate *ic3* again, since the *workingTeam(c)* required by *member(a2,b2,c)* and *member(a3,b3,c)* is already in the EDB. This situation is formalized in theorem 4.6.

Definition 4.5. A variable *x* is free in a repair $R_i(ic)$ if $x \in$ variables($R_i(ic)$) and $x \notin$ variables(*V(ic)*).

Theorem 4.6. A cycle *C = (ic1 r1 ... icn rn icn+1 = ic1)* is a *finite cycle* if $\forall$i, 1 ≤ i ≤ n, $\forall$p such that $p(X_1,...,X_m) \in ri$ and $p(Y1,...,Ym) \in V(ici+1)$, $\forall$k, 1 ≤ k ≤ m, X_k is free in *ri* $\Rightarrow Y_k \notin$ variables(*ri+1*).

Intuitively, free variables in a repair are the source of infinity since they propagate the violations to new objects other than the ones that initially violated the constraints in the cycle. The previous condition guarantees that the free variables in the repair of the first constraint are not propagated by the repair of the second constraint. Since such a condition is required for each two consecutive constraints, it is guaranteed that the cycle will not loop forever since no new objects will be infinitely introduced by the repairs.

Applying this condition to the cycle consisting of *ic3* and *ic14* we can conclude that it is a finite cycle, since the objects added by *ic14* (the free variables in its repair) do not appear in the repair of *ic3*, which means that the new objects are not propagated in the cycle.

There are two other cycle in our example, defined by the constraints (*ic14 ic5 ic3*) and (*ic14 ic6 ic3*), that do not satisfy the previous condition. However, these cycles are not infinite, since the free variables in *ic14* are propagated by *ic5/ic6* but not by *ic3*, which means that the new facts do not cause a new violation of *ic14*.

We propose another theorem to identify this kind of cycles, which determines whether all the constraints of a cycle are violated at most once.

Definition 4.7. Let *C = (ic1 r1 ... icn rn icn+1=ic1)* be a cycle in *G*, where each r_i corresponds to some repair $R_j(ici)$. Let $V(ici) = \{ p_1(\bar{X}_1),..., p_m(\bar{X}_m) \}$. Then,

$$Facts(ic_1) = (V(ic_1) \cup r_1)\theta_1$$

$$Facts(ic_i) = \bigcup_{k=1}^{t} r_i\theta_k \cup Facts(ic_{i-1}), i > 1$$

where θ_l is a substitution that bounds a distinct constant to each variable, each $\theta_k = \theta_j \cup \theta'_j$ is one of the *t* possible substitutions such that $Facts(ic_{i-1}) \models (p_1(\bar{X}_1),..., p_m(\bar{X}_m))\theta_j$ and θ'_j assigns a new constant to each variable *X* such that $X \in$ variables(r_i) and $X \notin$ variables($V(ic_i)$) if $\theta_j \neq \emptyset$, otherwise $\theta_k = \emptyset$.

Theorem 4.8. *C* is a finite cycle if, for each possible starting ic_i, $\exists$k, $1 \leq k \leq n$, such that $Facts(ic_k) = Facts(ic_{k+1})$.

Intuitively, for each i>1, the set $Facts(ic_i)$ extends the set $Facts(ic_{i-1})$ by taking into account the repairs required to satisfy ic_i. Therefore, the condition $Facts(ic_k) = Facts(ic_{k+1})$ guarantees that the constraint ic_{k+1} is not violated and, hence, the maintenance of the constraints in the cycle will not loop forever.

The previous results allow us to determine whether reasoning on a given conceptual schema will always terminate. Note, however, that this set of theorems is not complete due to the undecidability of this problem.

As an example of infinite cycle, we show the application of our approach to the classic ER schema shown in Figure 9 (Lenzerini and Nobili 1987).

Applying the same translation rules as in the case of UML, the logic representation of this schema is:

1. $\leftarrow b(X) \wedge a(X)$
2. $\leftarrow r(A,B) \wedge \neg a(A)$
3. $\leftarrow r(A,B) \wedge \neg b(B)$
4. $\leftarrow q(A,B) \wedge \neg a(A)$
5. $\leftarrow q(A,B) \wedge \neg b(B)$
6. $\leftarrow a(A) \wedge \neg minV(A)$
   $minV(A) \leftarrow r(A,B)$
7. $\leftarrow r(A,B) \wedge r(A,B2) \wedge B \neq B2$
8. $\leftarrow b(B) \wedge \neg minU(B)$
   $minU(B) \leftarrow r(A,B)$
9. $\leftarrow a(A) \wedge \neg minZ(A)$
   $minZ(A) \leftarrow q(A,B) \wedge q(A,B2) \wedge B \neq B2$
10. $\leftarrow q(A,B) \wedge q(A2,B) \wedge A \neq A2$

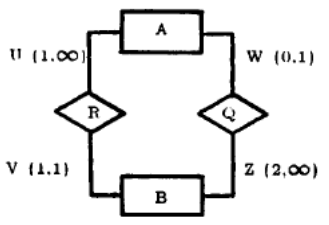


Fig. 9. A schema not finitely satisfiable

The dependency graph corresponding to this schema is shown in Figure 10. It has two cycles: the one defined by the constraints ic4 - ic9, and the other by ic2-ic9-ic5-ic8.

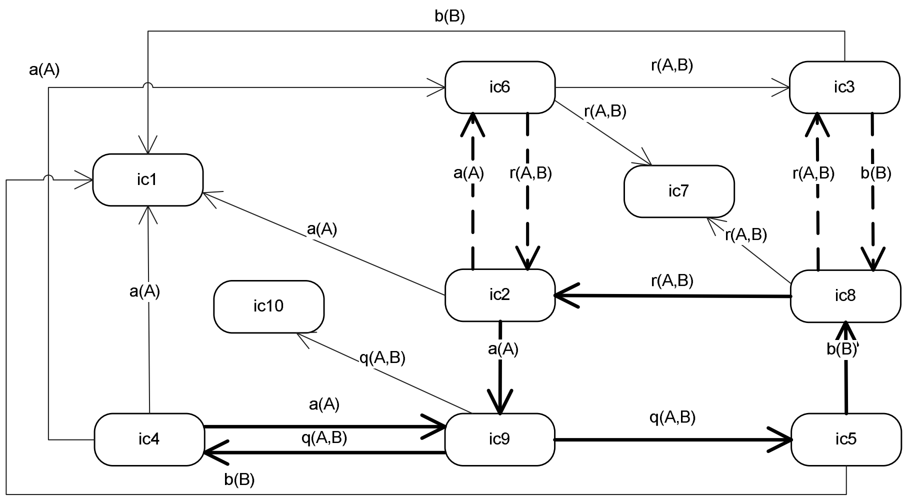


Fig. 10. Dependency graph. Cycles are highlighted, and superfluous arcs are dashed.

According to Theorem 4.4, the cycle ic4-ic9 is not finite, since the union of the potential violations of all the constraints in the cycle coincides with the union of potential repairs. The same happens with the cycle ic2-ic9-ic5-ic8.

If we apply Theorem 4.6 to the cycle ic4-ic9, we obtain that this cycle is finite. The reason is that constraint ic4 does not contain free variables (Definition 4.5), and the free variables in constraint ic9 are not propagated to the repairs of ic4.

However, according to the same theorem, the cycle ic2-ic9-ic5-ic8 is not finite, since the free variables in ic9 (the ones corresponding to the second term of the predicate *q*) are propagated by the repairs of ic5.

Thus, we must apply Theorem 4.8 in order to try to determine finiteness of this cycle. This consists in simulating a sequence of repairs of the constraints belonging to the cycle. We start by forcing the violation of ic2, and then we apply the repairs of the next constraints in the cycle:

Facts(ic2) = {r(1,2), a(1)}

Facts(ic9) = {r(1,2), a(1), q(1,3), q(1,4)}

Facts(ic5) = {r(1,2), a(1), q(1,3), q(1,4), b(3), b(4)}

Facts(ic8) = {r(1,2), a(1), q(1,3), q(1,4), b(3), b(4), r(5,3), r(6,4)}

Facts(ic2) = {r(1,2), a(1), q(1,3), q(1,4), b(3), b(4), r(5,3), r(6,4), a(5), a(6)}

If we do the same starting by any other constraint, we will obtain a similar result. Thus, according to Theorem 4.8, this cycle is not finite, since after having repaired all the constraints in the cycle, the set of *Facts* keeps growing. Intuitively, it can be seen that the constraints of this cycle will be violated infinitely, since the last facts added (*a(5)* and *a(6)*) require two new additions of predicate *q* and so on.

Hence, we cannot say that the cycle ic2-ic9-ic5-ic8 is finite and, thus, we cannot ensure that reasoning on this schema will always terminate.

4.2 Reasoning in the Case of Decidability

Once we have determined that all the models of a conceptual schema are finite (as it happens in our example), we can take advantage of the characterization of the logic formulas obtained from our UML and OCL schemas to define a reasoning procedure that works more efficiently than in the general case.

Several reasoning tasks have been considered in the literature (Baader, Calvanese et al. 2003; Brucker and Wolff 2006; Formica 2002; Hartmann 2001; Lenzerini and Nobili 1987), such as *satisfiability* (i.e. checking whether the schema admits a non-empty state that satisfies all the constraints), *liveliness of a predicate* (i.e. determining whether a certain class or association can have at least one instance) or *reachability of partially specified states* (i.e. assessing whether certain goals conceived by the designer may be satisfied). In general, each reasoning task can be formulated in terms of a particular goal to attain.

A well-known approach to deal with this problem is to define methods whose purpose is to construct a database state (i.e. an EDB) for which the tested property holds. That is, a sample EDB where both the particular goal to attain and all the integrity constraints in the schema are satisfied. In this way, these methods can uniformly deal with all reasoning tasks.

In this section we propose a new reasoning procedure based on this approach. We divide it in two different steps: *goal satisfaction* and *integrity maintenance*. These steps are defined in sections 4.2.1 and 4.2.2, respectively.

4.2.1 Goal Satisfaction

Our method is aimed at building a sample EDB which proves that the schema fulfills a specific property defined in terms of a certain goal *G* to attain. We assume that *G* is a conjunction of (positive and negative) literals corresponding to EDB predicates and built-in literals; which suffices to handle schema satisfiability, predicate liveliness and reachability of partially specified states (Queralt and Teniente 2006a).

The first step of our method determines the EDB facts that are required to satisfy G without taking into account whether they violate any integrity constraint. Positive literals in *G* define facts that are necessarily required to satisfy *G* while negative literals in *G* identify facts that the sample EDB under construction must not contain. Built-in literals state conditions over the values that the variables of positive and negative literals in *G* may take.

One of the most difficult tasks is the assignment of concrete values to the variables appearing in *G* in order to construct the sample EDB. Each possible choice defines a different alternative that satisfies *G*, i.e. a different sample EDB. We use Variable

Instantiation Patterns (VIPs) (Farré, Teniente et al. 2005) for this purpose. These VIPs guarantee that the number of sample EDBs to be considered is kept finite, by taking into account only those variable instantiations that are relevant for the schema, without losing completeness. I.e. the VIPs guarantee that if a solution is not found by instantiating the variables in the goal using only the constants they provide, then no solution exists. VIPs are selected according to the syntactic properties of the schema considered in each test:

1. The Simple VIP: for schemas without negation and integrity constraints.
2. The Negation VIP: for schemas with negation and/or integrity constraints.
3. The Dense Order VIP: for schemas with order comparisons over a dense domain (e.g. real numbers).
4. The Discrete Order VIP: for schemas with order comparisons over a discrete domain (e.g. integer numbers).

In our example, the appropriate VIP is the Negation VIP, since the schema has negation and integrity constraints, but not order comparisons. With this VIP, each variable of the fact to be included in the EDB is instantiated either with a previously used constant or with a new one. For instance, assume that p(X) must be instantiated and that the only constant used up this moment is 0. Then, according to this VIP, the only relevant instantiations are p(0) and p(1).

Step 1: Goal Satisfaction. Formally, the set *EDB* of facts required to satisfy *G* and the set Unw_{EDB} of facts that *EDB* may never include to fulfill the tested property are obtained as stated in definition 4.9. There is a different alternative *EDB* for each possible substitution θ provided by the corresponding VIP.

Definition 4.9. Let $G = \leftarrow p_1(\bar{X}_1) \wedge \ldots \wedge p_n(\bar{X}_n) \wedge \neg q_1(\bar{Y}_1) \wedge \ldots \wedge \neg q_m(\bar{Y}_m) \wedge b_1 \wedge \ldots \wedge b_s$, where p_i, q_j are base predicates and b_k are built-in literals.

Let θ be one of the possible ground substitutions obtained via an instantiation of *variables(G)* and such that $\forall$i, $1 \leq i \leq s$, $b_i\theta$ evaluates to true. Then,

– The set of facts required to satisfy *G* is $EDB = \{p_1\theta,\ldots,p_n\theta\}$

– The set of facts unwanted to satisfy *G* is $Unw_{EDB} = \{ q_1\theta,\ldots,q_m\theta \}$

As an example, assume that the designer wants to check the liveliness of the association *Audits* in the conceptual schema of Figure 1. *Audits* will be lively if the goal $G = \leftarrow audits(E,T)$ succeeds for some instantiation. Applying the step 1 of our method we will obtain two different EDBs that satisfy *G* according to the Negation VIP: EDB_1 = *audits*(0,0) and EDB_2 = *audits*(0,1).

4.2.2 Integrity Maintenance

Once we have determined the set of EDB facts that satisfies the goal *G* to attain, the problem of reasoning on the schema may be reduced to that of integrity maintenance (Moerkotte and Lockemann 1991). Note that, in fact, we already know that the property checked will be satisfied if the EDB resulting from Step 1 does not violate any constraint of the schema. If this is not the case, we must look for additional base facts (i.e. repairs) that make consistent the sample EDB being constructed.

Unfortunately, as explained in section 1.6.3, we may not rely on existing integrity maintenance methods to perform this activity. To our knowledge, the most appropriate method to perform the kind of integrity maintenance we require is the CQC-Method (Farré, Teniente et al. 2005). However, it is a semidecision procedure with efficiency limitations that make questionable its use in practical situations.

Thus, we need to build a new reasoning procedure, which can take advantage both of the dependency graph and the characterization of the logic formulas obtained from our schemas to work efficiently. Since the graph shows the interactions between the constraints, it provides the order in which they should be maintained. In principle, all constraints in the graph must be considered for maintenance since all of them may be violated by the EDB obtained as a result of Step 1. Vertices with no incoming arcs or whose incoming arcs have already been maintained are selected with priority so that a constraint is not considered until all the constraints that may violate it have already been maintained.

An integrity constraint *ic* must be repaired if its potential violations hold in the sample EDB. Maintenance of *ic* results in the inclusion of its repairs in the sample EDB being constructed. Note that *ic* may be violated by several different instantiations of its potential violations. Each of them gives raise to different repairs to be added in the EDB. For instance, if EDB = {*..., audits(e1,t1), audits(e1,t2), audits(e2,t3), ...*} and constraint *ic1* has to be repaired at this moment, the facts to be added to EDB are {*workingTeam(t1), workingTeam(t2), workingTeam(t3)*}.

If a constraint with an empty set of repairs is violated, the sample EDB being constructed must be discarded since it is impossible to make it satisfy such a constraint.

The process of integrity maintenance is formalized as follows. Note that we also use the VIPs to assign concrete values to the existential variables that appear in the repairs of a constraint. Backtracking must be performed each time that the sample EDB under construction reaches a situation where the selected *ic* can not be repaired. Such backtracking involves considering a different repair of one of the constraints that has been maintained before *ic*.

Step 2: Integrity Maintenance. Let $ic = \leftarrow p_1(\bar{X}_1) \wedge \ldots \wedge p_n(\bar{X}_n) \wedge \neg q_1(\bar{Y}_1) \wedge \ldots \wedge \neg q_m(\bar{Y}_m) \wedge b_1 \wedge \ldots \wedge b_s$ be the condition selected for maintenance from the dependency graph, where p_i, q_j are base predicates and b_k are built-in literals. Let EDB_i be the set of required facts at that moment. Let *EvalV(ic)* and *EvalR(Ri(ic))* be the set of built-in literals that appear in the body of *ic* and in the body of the rule from which *Ri(ic)* is obtained, respectively. Then EDB_{i+1} is computed as follows, where each $\theta_k = \theta_j \cup \theta'_j$ is a substitution such that $EDB_i \models (V(ic) \wedge EvalV(ic))\theta_j$ and θ'_j is one of the possible substitutions obtained from an instantiation of all the variables in *variables(Ri(ic))* \ *variables(V(ic))* such that $EvalR(Ri(ic))\theta'_j$ evaluates to true, if $\theta_j \neq \varnothing$, otherwise $\theta_k = \varnothing$:

```
if Ri(ic) = $\emptyset$ and EDBi $\models$ $(p_1(\bar{X}_1) \wedge ... \wedge p_n(\bar{X}_n) \wedge b_1 \wedge ... \wedge b_s)\theta_j$
        then error (ic cannot be repaired)
else
        $EDB_{i+1} = \bigcup_{k=1}^{t} R_i(ic)\theta_k \cup EDB_i$
        if $\exists q_j \in Unw_{EDB}$ such that $q_j \in EDB_{i+1}$
                then error (ic cannot be repaired)
```

Figure 11 shows an execution of the integrity maintenance step of our method for one of the EDBs obtained in Step 1. Each row in the figure shows the integrity constraint being maintained (as selected through the order defined by the dependency graph) and the additions to the EDB required to repair the constraint, if any. A row contains several constraints when none of them is violated by the EDB under construction. As a result of the execution, our method obtains a sample EDB which confirms that the association *Audits* is lively.

The Step 1 consists in the addition of a fact *audits(0,1)*, which corresponds to one of the alternatives provided by de VIP. If no consistent EDB can be found with this instantiation, the other alternative (*audits(0,0))* should be selected.

The constraint *ic1* is selected first since it is the only vertex with no incoming arcs in the graph. It is violated since V(*ic1*)= *audits(E,T)* holds in the EDB with substitution *θj* ={E/0, T/1}. Then, since R(*ic1*)=*workingTeam*(*T*), the repair *workingTeam*(1) is added to the sample EDB to ensure that it does not violate *ic1*.

The next constraint to be selected is *ic10* since all its only predecessor (*ic1*) has already been maintained. Similarly, it is violated since V(*ic10*)=*workingTeam*(*T*) holds in the EDB with substitution *θj*={T/1}. Since *R(ic10)=member*(M,E,T), the fact *participant*(2,3,1) is added to the sample EDB since we assume that the substitution obtained is *θ'j* ={M/2, E/3}.

The rest of alternative substitutions provided by the VIP, are *θ'j* ={M/0, E/0}, *θ'j* ={M/0, E/1}, *θ'j* ={M/0, E/2}, *θ'j* ={M/0, E/3}, *θ'j* ={M/1, E/0}, *θ'j* ={M/1, E/1}, *θ'j* ={M/1, E/2}, *θ'j* ={M/1, E/3}, *θ'j* ={M/2, E/0}, *θ'j* ={M/2, E/1}, *θ'j* ={M/2, E/2}, that is, the combinations corresponding to the use of all the existing constants (0 and 1) plus a new one (2) for the variable M, and all the existing constants (0, 1 and 2) plus a new one (3) for the variable E. This alternatives will be used in case the current derivation fails. The alternative chosen in this example is the first that leads to a successful derivation, which is the first one in which *member* has the OID 2, different from the ones used up to now for other classes, and *employee* has the OID 3, also different from the ones used for other classes and different from 1, which would violate the constraint *ic12*.

The method proceeds then with *ic14*, since its only predecessor at this moment is *ic3*, which belongs to its same cycle. Its other predecessor is *ic1*, which has already been maintained. The violation of *ic14* requires considering two additional repairs whose concrete values have also been obtained via the application of a VIP.

	Selected constraint(s)	*Additions to the EDB*
Step 1		{*audits*(0,1)}
Step 2	ic1	{*workingTeam*(1)}
	ic10	{*member*(2,3,1)}
	ic14	{*hasRecruited*(2,4), *member*(4,5,1)}
	ic13, ic11, ic5, ic6, ic3, ic8	
	ic2	{*employee*(0)}
	ic9, ic7	
	ic4	*{employee(3), employee(5) }*
	ic12	
Sample EDB	***{audits(0,1), workingTeam(1), member(2,3,1), hasRecruited(2,4), member(4,5,1), employee(0), employee(3), employee(5)}***	

Fig. 11. An execution that proves that *Audits* is lively

Note that the rest of alternatives different from *hasRecruited*(2,4) fail due to some constraint. In particular *hasRecruited*(2,0), *hasRecruited*(2,1) and *hasRecruited*(2,3) use existing OIDs in *employee*, *workingTeam* and *member* for a new instance of member, which is not allowed, and *hasRecruited*(2,2) violates the constraint *ic13*. The same happens with *member*(4,5,1), since the facts *member*(4,0,1), *member*(4,1,1), *member*(4,2,1) and *member*(4,4,1) do not guarantee the uniqueness of OIDs. On the other hand, *member*(4,3,1) violates *ic7*, since *member*(2,3,1) already belongs to the EDB. Thus, the only successful alternative is *member*(4,5,1).

The next constraints to be considered according to the graph are *ic11*, *ic13, ic5, ic6* and *ic3,* all of them successors of *ic14* and currently with a single predecessor, and *ic8*, whose predecessor is *ic3,* which has just been checked. None of these constraints is violated by the current EDB, so no facts are added. Since *ic3* has not been violated and, thus, not repaired, there is not need to check *ic14* again, despite being successor of *ic3* in the cycle.

The next step takes *ic2*, successor of *ic8,* which is violated by the first fact inserted, *audits(0,1).* The repair consists in adding *employee(0)* to the EDB.

The next constraint, *ic9*, which has no repairs, is not violated since the EDB only contains an instance of the predicate *audits*, so the maintenance continues with *ic7*, which is neither violated.

The only constraints that remain to be treated are *ic4* and *ic12*, which have no active predecessors at this moment. We take for example *ic4*, which is violated by the facts *member(2,3,1)* and *member(4,5,1).* To repair these violations, two new facts, corresponding to the employees required by each instance of *member*, are added in the EDB.

The last constraint to be considered is *ic12*, which is not violated. Since all the constraints have already been checked and successfully repaired when necessary, the execution ends, and the EDB constructed provides an example that demonstrates the property checked, i.e. the liveliness of the association *Audits.* The EDB is {*audits(0,1), workingTeam(1), member(2,3,1), hasRecruited(2,4), member(4,5,1), employee(0), employee(3), employee(5)*}. Note that seven additional facts have been added to the EDB to ensure that it does not violate any integrity constraint.

Let us see now an unsuccessful execution of the method, that is, an execution that is not able to provide a sample EDB satisfying the goal and, thus, demonstrates that such a goal is impossible to reach according to the structural schema defined. The test we want to perform is

auditorAndMember $\leftarrow$ audits(0,1) $\wedge$ member(M,0,1)

that is, we want to check whether it is possible that an employee that audits a team also belongs to it. This is not possible according to *ic12*, so let us see how the method realises of this.

As can be seen in Figure 12, the step 1 consists in placing a set of facts satisfying the goal *G* = $\leftarrow$ *audits(0,1)* $\wedge$ *member(M,0,1)* in the EDB. The alternatives provided by the VIP to instantiate the variable M are 0, 1 and 2. Since OIDs 0 and 1 correspond to an employee and a team, respectively, the only possibility is to instantiate the OID M of the member with the value 2.

The constraints will be considered in the same order as in the previous example, since the sequence of violations is very similar. The alternatives of instantiation are very similar too, so we will not explain them in the same detail.

The first constraint to be considered is *ic1*, which adds the fact *workingTeam(1)* to the EDB.

The next one is *ic10*, that this time is not violated since a member of the working team 1 already belongs to the EDB. As in the previous example, *ic14* is violated and needs an instance of *hasRecruited* and another instance of *member* to be repaired. Again, we choose the right alternative to instantiate both predicates, in order not to violate any other constraint.

The next violation is of *ic2*, which requires the addition of an *employee* in the EDB, given by *audits*(0,1). The last repairs also imply the addition of employees according to *ic4.*

	Selected constraint(s)	*Additions to the EDB*
Step 1		{*audits*(0,1), *member*(2,0,1)}
Step 2	ic1	{*workingTeam*(1)}
	ic10	
	ic14	{*hasRecruited*(2,3), *member*(3,4,1)}
	ic13, ic11, ic5, ic6, ic3, ic8	
	ic2	{*employee*(0)}
	ic9, ic7	
	ic4	{*employee(1), employee(4)* }
	ic12	**error (ic12 cannot be repaired)**

Fig. 12. An execution that proves that *auditorAndMember* is not satisfiable.

The last constraint to be considered is *ic12*, which is violated by the facts in the EDB and cannot be repaired. Thus we have to reconsider the last alternative taken. The only alternative instantiation possible is in the repair of *ic14*, where *hasRecruited*(3,2) could have been considered instead of *hasRecruited*(2,3). However, it is easy to see that the result is exactly the same. Thus, since there are neither other alternative repairs for any of the constraints, nor other successful instantiations provided by the VIPs to be considered, we can conclude that the goal G is unsatisfiable.

To evaluate our efficiency improvement regarding integrity maintenance, we compare how many integrity constraints are checked in the CQC Method and in our integrity maintenance step. Let N be the number of integrity constraints and R the number of repairs needed to obtain a solution. Then, we have that the CQC Method will check in average (N/2)*R+N constraints to ensure that the solution is valid. We assume that the CQC will need to check again half of the constraints (i.e. N/2) for each repair.

Our integrity maintenance step will check exactly N constraints if the dependency graph has no cycles. If all the cycles in the graph satisfy theorems 4.6 or 4.8 (and do not satisfy theorem 4.4), at most N+K constraints will be checked, where K is the number of constraints that belong to cycles, since each such constraint is considered at most twice.

The number of constraints we will check when a cycle satisfies theorem 4.4 is more difficult to establish because it depends on the contents of the IB before repairing the constraints in the cycle. In this kind of cycles, the number of constraints considered is N+K*S, where S is the number of times that the constraints in the cycle are checked.

Thus, the number of constraints considered by our procedure in the worst case is N+K*R, which coincides with the worst case of the CQC Method. Note, however, that our worst case only happens when all the constraints of the schema belong to cycles satisfying theorem 4.4.

5

Validation of a Conceptual Schema with Operations

In this chapter we explain our approach to the validation of a complete conceptual schema. This does not mean to check separately which properties are fulfilled by the structural part, as explained in chapter 3, and which ones are satisfied by the behavioral part. Since the operations defined in a schema specify the only changes that can be performed on the IB, the structural part cannot be validated independently of them. Thus, as well as checking properties of the operations, such as applicability and executability, validating the behavioral part of a schema also means taking into account the operations defined when determining which properties are fulfilled by the structural part.

When taking the behavioral schema into account in the validation, it is essential to establish how operation contracts must be interpreted, in order to infer which are the changes they make on the IB. There are two alternative semantics of operation contracts, the *strict interpretation* and the *extended interpretation,* which are discussed in section 5.1. As will be seen, the strict interpretation provides several advantages over the extended one and, thus, we choose the strict interpretation for our operation contracts.

In section 5.2 we explain how to incorporate the semantics of operations in the logic formalization of the schema, so that we can ensure that the only changes allowed are those specified by the operations.

In section 5.3 we formalize and exemplify a set of reasoning tests that can be performed on a complete conceptual schema. Some of them correspond to properties of the complete conceptual schema, taking into account the meaning of operations, while others are specific to validate the correct definition of operations.

Finally, in section 5.4 we explain how the existing CQC Method has been extended and used to validate a conceptual schema with operations.

5.1 Semantics of Operation Contracts

Operation contracts define the effect of an operation on the IB. They consist of a *precondition*¸ which expresses a condition that the IB and the parameters must satisfy when the call to the operation is made, and a *postcondition,* which expresses a condition that the IB must satisfy after the application of the operation.

The execution of an operation results in a set of one or more *structural events* to be applied to the IB. A *structural event* is an elementary change in the population of an entity type or a relationship type, i.e. the creation, deletion or modification of instances of a given type. The precise number and meaning of structural events depend on the conceptual modeling language used. In this paper, we assume the following kinds of structural events:

- Entity insertion: creates a new instance of an entity type
  insert(EntityType(attribute$_1$,...,attribute$_n$))
- Entity deletion: deletes an instance of an entity type
  delete(EntityType(instance))
- Entity generalization: an instance is moved from an entity type to its supertypes
  generalize(instance, EntityType)
- Entity specialization: an instance is moved from an entity type to one of its subtypes
  specialize(instance, EntityType(attribute$_1$,...,attribute$_n$))
- Relationship insertion: links a set of instances of entity types. When the relationship type is an association class, values for its attributes must be indicated
  newLink(RelationshipType(participant$_1$,...,participant$_n$, [attr$_1$,...,attr$_n$])).
- Relationship deletion: deletes a link between a set of instances
  deleteLink(RelationshipType (participant$_1$,...,participant$_n$)
- Attribute update: changes the value of an attribute
  update(EntityType(instance, attribute, value))

More complex events can be expressed in terms of these ones. For instance, an entity migration can be specified as an entity generalization and an entity specialization.

The application of a set of structural events to a state *IB* of the information base results in a new state *IB'*. Given a state of the information base *IB,* there are several sets of structural events that lead to new states satisfying an operation postcondition. Of these, we are only interested in the minimal ones. A set *S* of structural events is minimal if no proper subset of *S* is also a set of structural events that satisfies the postcondition. This is the way in which we deal with the frame problem (Borgida, Mylopoulos et al. 1995).

As we will see, the particular semantics given to an operation contract determines the set of structural events to be applied to the information base when the operation is executed.

In addition to preconditions and postconditions, integrity constraints play an important role in the definition of the semantics of operation contracts, since every operation can assume that the integrity constraints are true when it is entered and must in return ensure that they are true on its completion (Hoare 1972). Previous work on conceptual modeling has provided precise definitions for integrity constraints and definitions for preconditions and postconditions, but sometimes without explicitly establishing a clear relationship between them. Thus, little attention has generally been paid to the precise semantics of operation contracts, in terms of how the satisfaction of integrity constraints is guaranteed after each operation is executed.

In many proposals, integrity constraints are kept separated from the discussion of operation contracts when, in fact, they determine the way contracts are specified. In some studies, definitions are given for integrity constraints and for preconditions and postconditions, but the implications that constraints have on the way operation contracts are specified is not really discussed (Martin and Odell 1999; Pressman 2004; Rumbaugh, Jacobson et al. 2004; Wieringa 1998). In one chapter, it is even stated explicitly that integrity constraints are not included in the discussion of preconditions and postconditions for the sake of simplicity (Larman 2004). All of these approaches make clear that integrity constraints must hold before and after every operation execution. However, they do not define whether the operation must guarantee consistency by repairing the information base or by rejecting the operation when a violation occurs.

In contrast, the relationship between operation contracts and integrity constraints is clearly established by some authors (D'Souza and Wills 1998; Meyer 1997; Olivé 2004). According to them, the state of the information base generated by the execution of an operation must satisfy both the postcondition and the integrity constraints every time the precondition is satisfied. This means that each operation is responsible for recovering the consistency of the information base when this is lost during execution of the operation. As we will see, this semantics corresponds to our extended interpretation of operation contracts, and as we will discuss, involves several drawbacks from the point of view of the characteristics of good software specification.

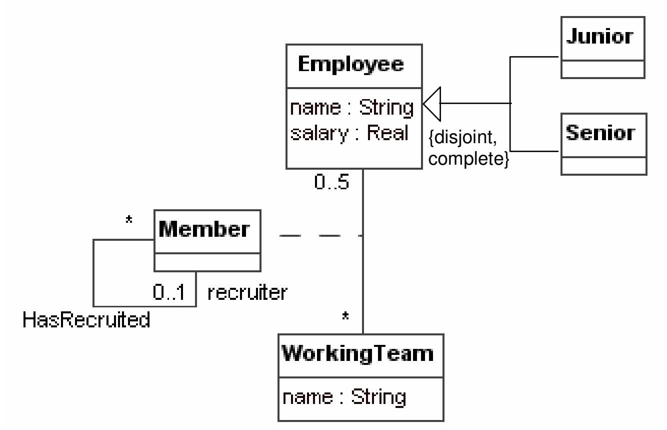
Finally, this relationship has been disputed when it deals with the specification of business rules (Devos and Steegmans 2005). In that analysis, the authors concluded that preconditions, postconditions and invariants should not be offered as concepts in object-oriented analysis and, thus, two new special kinds of constraints (class and event constraints) must be introduced to model business rules. In this paper, we show that a strict interpretation of operation contracts allows successful use of preconditions, postconditions and invariants in conceptual modeling, without the need for consideration of new concepts.

We have analyzed the traditional semantics of operation contracts, which we call e*xtended interpretation*, and show that it has important drawbacks in terms of the desirable properties of software specifications. To solve these problems we propose a different way of specifying contracts, called *strict interpretation.* The main difference between them lies in the way operation postconditions and integrity constraints are guaranteed, which has an impact on the desirable properties of operation contracts.

Given an information base *IB* and an operation *Op*, the semantics of *Op* defines the conditions under which *Op* can be applied and the new *IB'* obtained as a result of applying *Op* to *IB*. Since there is a one to one correspondence between each operation and the operation contract specifying its behavior, we will use the two terms interchangeably. Taking this into account, in the following subsections we formalize and discuss both interpretations.

In short, a strict interpretation assumes a passive behavior of operations, since it prevents an operation from being applied if an integrity constraint is violated (although both its preconditions and postconditions are satisfied). In contrast, an extended interpretation entails reactive behavior of operations, since it must ensure that integrity constraints are satisfied whenever they are violated, so that the operation will always be applied if its precondition is satisfied.

In this section we use a variation of the example, shown in Figure 13, that does not include any semantic flaws in order to focus on the different interpretations of operation contracts. This time the schema includes a classifictation of *Employees*, which can be either *Junior* or *Senior*, that are grouped into *WorkingTeams.*



Integrity constraints

- Employees are identified by name
- Working Teams are identified by name
- A member cannot recruit himself

Fig. 13. New example about Employees and Working Teams

The behavioral schema of this example is not shown here because the formalization of its operation contracts depends on the particular semantics assumed. The operations will appear during the formalization and discussion of the different semantics.

5.1.1 The Extended Interpretation

An operation contract states that if the conditions specified in the precondition are satisfied, then the state described in the postcondition is guaranteed, together with the integrity constraints specified in the schema (Meyer 1992). In other words, satisfaction of the precondition always implies satisfaction of the postcondition and the integrity constraints. We will call this *extended interpretation*, since integrity constraints are considered as clauses implicitly added to the postcondition, i.e. it is the responsibility of the operation to ensure satisfaction not only of the postcondition but also of the integrity constraints of the schema.

Let *IB* and *IB′* be states of the information base. Let *Op(IB, IB′)* denote that *IB′* is the result of applying an operation *Op* on *IB*. Let *Pre* and *Post* be the precondition and postcondition of *Op* respectively. Let *IB′ = S(IB)* denote that *IB′* is obtained as a result of applying all the structural events in *S* to *IB*. Let *S*, S_2 and S_3 be sets of structural events and *S* be the minimal set that satisfies *Post* when applied to *IB*[3]. Let *IC* be the set of integrity constraints defined in the conceptual schema.

Definition 4.1: *Extended Interpretation* of an operation *Op*

$\forall$ *IB*, *IB′* such that *Op(IB, IB′)*, the following four conditions hold:

a) $IB \models Pre \wedge IC$
b) $IB' = S_2(IB)$ and $S \subseteq S_2$
c) $IB' \models Post \wedge IC$
d) $\neg\exists\, S_3,\ S \subseteq S_3 \subset S_2$ such that $S_3(IB) \models Post \wedge IC$

Intuitively, the first condition states that the transition is only possible if *IB* satisfies *Pre* and it is consistent. The second condition asserts that to obtain *IB′* at least the structural events in *S* must be applied. However, it does not rule out the application of additional structural events to *IB*. The third condition requires *IB′* to satisfy *Post* and to be consistent. Finally, the fourth condition states a minimality condition on S_2 in the sense that there is no proper subset of S_2 that satisfies *Post* and *IC*.

An extended interpretation allows for several different sets of structural events S_i to be applied to *IB*, provided that all of them include at least the events in *S* and satisfy the minimality condition. The additional structural events in S_i must be such that they guarantee that no constraint is violated in the resulting state, even though some of them were violated by the events in *S*. Clearly, if *S* itself does not violate any constraint there is no need to consider additional structural events.

We will use the operation *newWorkingTeam,* aimed at creating an instance of *WorkingTeam*, to illustrate that the previous definition formalizes the traditional meaning of a contract. The contract shown in Figure 14 specifies this operation.

Here, *S*={*insert(WorkingTeam(name))*} is sufficient to satisfy the postcondition of *newWorkingTeam(name)*, since it is the only structural event that causes *IB′* to satisfy the postcondition.

[3] Exceptionally, several minimal sets *S* can exist but only in those cases in which some kind of random behavior is desired. In that case, any of them can be arbitrarily chosen.

```
Op:     newWorkingTeam(name: String)
Pre:    --there is no other team with the same name
        not WorkingTeam.allInstances() → exists(t|t.name=name)
Post:   --a new instance t of WorkingTeam, identified by
        --name is created
        WorkingTeam.allInstances()->exists(t | t.oclIsNew() and
        t.name = name)
```

Fig. 14. Contract for the operation *newWorkingTeam*

We will now show the conditions under which *newWorkingTeam* can be applied to *IB* and the new *IB′* arising from the application of *newWorkingTeam* according to the extended semantics. Two different situations must be distinguished, depending on the contents of *IB*:

1. *IB* does not contain any working team identified by *name*. In this case:
   - Condition a) is guaranteed, since *Pre* is satisfied and *IB* is consistent.
   - b) states that *IB′* = $S_2(IB)$ and $S \subseteq S_2$, i.e. any information base resulting from applying a set S_2 of structural events that contains *S*={*insert(WorkingTeam(name))*} will satisfy this condition.
   - c) states that *IB′* must imply both the postcondition (this is always true since it is satisfied by *S*) and all integrity constraints. Under the assumption that a working team with the same *name* does not already exist, no integrity constraint can be violated by the application of *S*, so $S = S_2$ in this case.
   - Condition d) is also satisfied, since there is no subset of S_2 = {*insert(WorkingTeam(name))*} satisfying *Post.*
2. *IB* contains a working team identified by *name*. In this case, the operation will not be executed, since its precondition is not satisfied and condition a) of Definition 4.1 does not hold.

In summary, according to an extended interpretation the semantics of the operation *newWorkingTeam* is such that if *IB* does not contain any working team identified by *name* then the operation leads to the insertion of a new working team, with the specified name. Otherwise, the operation will not be applied and *IB* will remain unchanged.

However, we see an important drawback in the previous operation contract. The problem is that its precondition is redundant, since the same aspect of the specified system (two working teams with the same name cannot exist) is already guaranteed by the *uniqueTeam* constraint. Non-redundant conceptual schemas provide several advantages regarding desirable properties of software specifications and ease of design and implementation, such as conciseness, modifiability and consistency (Costal, Sancho et al. 2002).

To avoid redundancy in the specification of the operation *newWorkingTeam* we should define an operation contract like the one shown in Figure 12 but with an empty precondition, as shown in Figure 15.

Assuming an extended interpretation of the previous operation contract, we obtain the following behavior. Since the postcondition has not changed, the set *S* will be the same as before, *S*={*insert(WorkingTeam(name))*}. We distinguish the same two relevant situations depending on the contents of *IB*.

```
Op:      newWorkingTeam(name: String)
Pre:
Post:    --a new instance t of WorkingTeam, identified by
         --name is created
         WorkingTeam.allInstances()->exists(t | t.oclIsNew() and
         t.name = name)
```

Fig. 15. Non-redundant contract for operation *newWorkingTeam*

1. *IB* does not contain any team identified by *name*. In this case the behavior is the same as in the contract shown in Figure 12. Hence, *IB'* is obtained by inserting a working team with *name* into *IB*.
2. *IB* contains a working team *wt* identified by *name*:
   – a) is guaranteed.
   – b) states that any *IB'* resulting from applying a set S_2 , $S \subseteq S_2$, of structural events that contains *S* will satisfy this condition.
   – c) states that *IB'* must imply both the postcondition and all integrity constraints. Since *S* itself violates the first integrity constraint, S_2 must be a superset of *S*.
   – d) states that S_2 must be minimal. Adding either *delete(WorkingTeam(wt))* or *update(WorkingTeam(wt, name, newName))* to *S* is sufficient both to satisfy this condition and repair the previous violation, where *newName* is different from all the existing names of working teams.

In summary, according to the semantics of the non-redundant operation contract for *newWorkingTeam* based on an extended interpretation, if *IB* does not contain a team identified by *name* the operation results in the application of the structural event *insert(WorkingTeam(name))*. Otherwise, if *IB* contains a working team *wt* identified by *name*, it results in the application of $S_2 = S \cup$ *{delete(WorkingTeam(wt))}*, or $S_2 = S \cup$ *{update(WorkingTeam(wt, name, newName))}* in order to satisfy the postcondition and repair the violation.

Besides having undesired collateral effects, the previous operation contract admits two different sets of structural events to be applied to *IB*. In addition to the events in *S*, one of them deletes the existing working team with the same name, while the other changes the name of the already existing instance (note that both of them satisfy both *Post* and *IC*). Clearly, both alternatives correspond to completely different business rules and random behavior of this operation is not acceptable.

This is a clear example of how assuming an extended interpretation may lead to ambiguous contracts. The problem is that, as stated in the IEEE Recommended Practice for Software Requirements Specifications (SRS) (Various Authors 1998), a good SRS must be unambiguous in the sense that each requirement, and in particular the ones stated in the operation contracts, must have a unique meaning.

There are two possible ways to avoid ambiguity that depend on the expected behavior of the operation. One possibility is to strengthen the operation precondition to ensure that no integrity constraint violation will be produced, in case the designer wants the operation to be rejected if there is a constraint violation, as in the contract in Figure 14. Clearly, this is the intended behavior of this operation, that is, if a working team with the same name already exists, the operation should not be applied .

The problem is that this way of avoiding ambiguity results in a redundant operation precondition. As mentioned, despite being acceptable, this situation would not follow suggested good practices for conceptual modeling (Costal, Sancho et al. 2002).

In contrast, if the designer wants the operation to recover the consistency of the information base, the solution to avoid ambiguity is to explicitly state how to repair integrity constraint violations in the operation postcondition. This could be done in the previous example by choosing one of the two possible ways to repair the constraint violation (i.e. either to delete the working team with the same name or to rename it) and specifying the corresponding OCL expression in the operation postcondition.

Besides the explicit integrity constraints, either graphical or textual, a structural schema also includes several constraints that are implicit in its specification, for instance, that an association class does not have two different instances defined by the same instances of the participants.

In the previous example, we have seen how the textual constraints must be guaranteed by the precondition. In what follows, we illustrate how implicit and graphical constraints must also be taken into account when specifying an operation contract under an extended interpretation. To this end, let us consider the following operation *newMember*, which registers a new member in a team and associates it to a recruiter.

```
Op:     newMember(emp:Employee, team:WorkingTeam, rec:Member)
Pre:    --the employee emp does not already belong to the team
        team.employee->excludes(emp)
        --the team does not already have 5 members
        team.employee->size()<5
        --the recruiter rec is not the member we are going to
        --create
        rec.employee<>emp or rec.workingTeam<>Team
Post:   Member.allInstances()->exists(m|m.oclIsNew() and
        m.employee=emp and m.workingTeam=team and
        m.recruiter=rec)
```

Fig. 16. Contract for the operation *newMember*

The first precondition, which ensures that the member we are going to create does not already exist, is needed because of the implicit constraint in associations stating that an association cannot have duplicated instances. The second precondition prevents the violation of the cardinality constraint stating that a team must have at least 5 members. Finally, the third precondition guarantees that the constraint *notSelfRecruited* will not be violated after satisfying post.

Now, *S*={*newLink(Member(emp,team)), newLink(HasRecruited(rec,newMember))*}, where *newMember* is the instance of the association class *Member* created by *newLink(Member(emp,team)),* is sufficient to satisfy the postcondition of *newMember*, since it is the minimal set of structural events that causes *IB'* to satisfy the postcondition.

Following the same reasoning as in the previous cases, when the *IB* previous to the execution of the operation satisfies all the preconditions, the operation results in the creation of an instance of member with the given parameters. Otherwise, the operation is not applied.

Notice that, to obtain this behavior, we have needed to add preconditions that are redundant with constraints, either textual (third precondition), graphical (second precondition) or implicit constraints (first precondition). Again, trying to remove them has undesirable consequences.

We will start analyzing the first precondition. Assume that it is removed, so the relevant situations we need to study are the following, assuming in both cases that the rest of preconditions are satisfied:

1. The employee *emp* is not a member of *team* in *IB*:
   - Condition a) is guaranteed, since *Pre* is satisfied and *IB* is consistent.
   - b) states that $IB' = S_2(IB)$ and $S \subseteq S_2$, i.e. any information base resulting from applying a set S_2 of structural events that contains *S*={*newLink(Member(emp,team)), newLink(HasRecruited(rec,newMember))*} will satisfy this condition.
   - c) states that *IB'* must imply both the postcondition (this is always true since it is satisfied by *S*) and all integrity constraints. Since the same member does not already exist, no integrity constraint can be violated by the application of *S*, so $S = S_2$ in this case.
   - Condition d) is also satisfied, since there is no subset of $S_2 = S$ satisfying *Post.*
2. The employee *emp* is already a member of *team* in *IB*:
   - a) is guaranteed.
   - b) states that any *IB'* resulting from applying a set S_2 of structural events that contains *S* will satisfy this condition.
   - c) states that *IB'* must imply both the postcondition and all integrity constraints. Since the same member *m* already belongs to the *IB*, it is impossible to satisfy the postcondition, that is, to add *emp* as a new member of *team*, and the integrity constraints, in particular the uniqueness of the instances of an association. Thus, since no additional events can be added to repair this violation, this condition can never be satisfied.

In summary, according to an extended interpretation, the first precondition of the contract in Figure 16 cannot be removed despite being redundant, since this would cause the operation to be unapplicable when the *IB* already contains the same member.

The same happens when removing the third one, since it is impossible to satisfy simultaneously both the postcondition and the constraint *notSelfRecruited* when the new member to be created is exactly the recruiter *rec.* Finally, removing the second precondition leads to an unexpected and random behavior, as happens in the contract of Figure 15.

It can therefore be concluded that assuming the extended interpretation forces redundancy of operation contracts, either in the precondition or in the postcondition.

5.1.2 The Strict Interpretation

To solve the problem of redundancy caused by the extended interpretation, we propose an alternative way to understand operation contracts. This interpretation prevents the application of the operation when an integrity constraint is violated during execution. We call this approach *strict interpretation*, since the operation needs only be responsible for satisfying the postcondition and does not address the integrity constraints. This approach is formalized in Definition 4.2.

Let *IB* and *IB'* be states of the information base. Let *Op(IB, IB')* denote that *IB'* is the result of applying an operation *Op* on *IB*. Let *Pre* and *Post* be the precondition and postcondition of *Op*, respectively. Let *IB' = S(IB)* denote that *IB'* is obtained as a result of applying all the structural events in *S* to *IB*. Let *S* be the minimal set of structural events that satisfies *Post* when applied to *IB*. Let *IC* be the set of integrity constraints defined in the conceptual schema.

Definition 4.2: *Strict Interpretation* of an operation *Op*

$\forall$ *IB, IB'* such that *Op(IB, IB')*, the following three conditions hold:

a) $IB \models Pre \wedge IC$
b) $IB' = S(IB)$
c) $IB' \models Post \wedge IC$

As in Definition 4.1, the first condition states that there is a transition from an information base *IB* to an information base *IB'* as a result of applying an operation *Op* only if *IB* satisfies *Pre* and it is consistent (i.e. it satisfies all integrity constraints). Moreover, according to the second condition, *IB'* is obtained exactly as a result of applying the minimal set *S* of structural events that satisfies *Post* to *IB*. Finally, the third condition requires *IB'* to satisfy *Post* (always true according to b)) and to be consistent. If any of the conditions do not hold, *Op* is not applied to *IB*.

Notice that the resulting state will always be consistent, since integrity constraints were satisfied before the operation and they will always be satisfied at the end. If the execution of the operation does not violate any of the integrity constraints, then they are already guaranteed. On the other hand, if the execution violates any of them, the operation is rejected and the state remains unchanged.

An example is provided by the conditions under which the non-redundant contract of *newWorkingTeam* as specified in Figure 15, can be applied to *IB* and the new *IB'* resulting from its execution according to the strict interpretation. Recall that *S*={*insert(WorkingTeam(name))*} is the minimal set of structural events that satisfies the postcondition. We distinguish the same two situations as before depending on the contents of *IB*:

1. *IB* does not contain any working team identified by *name*:
   - a) is guaranteed.
   - b) states that *IB′ = S(IB)*.
   - *IB′*, as obtained according to b), satisfies c) since *S* necessarily satisfies the postcondition and applying *S* to *IB* never violates any integrity constraint.
2. *IB* contains a working team identified by *name*:
   - a) is guaranteed.
   - b) states that *IB′ = S(IB)*, i.e. *IB′* is obtained by inserting a working team with *name* into *IB*
   - c) is not satisfied since *IB′* will always violate the second integrity constraint.

Thus, *newWorkingTeam* cannot be applied in such an *IB* since it is impossible to satisfy all conditions required by a strict interpretation.

In summary, according to a strict interpretation the semantics of the operation *newWorkingTeam* is such that if *IB* does not contain any working team identified by *name* then the operation results in the application of the structural event *insert(WorkingTeam(name))*. Otherwise, the operation will not be applied since it leads to the violation of an integrity constraint. Clearly, this is the intended behavior of this operation and it has been obtained without the need to specify a redundant precondition.

Also, it can be seen that if all the preconditions of the contract in Figure 16 are removed, its semantics under a strict interpretation is as expected, that is, the same as that of the redundant contract under an extended interpretation. The non-redundant contract that should be specified under a strict interpretation is shown in Figure 17.

```
Op:     newMember(emp:Employee, team:WorkingTeam, rec:Member)
Pre:
Post:   Member.allInstances()->exists(m|m.oclIsNew() and
        m.employee=emp and m.workingTeam=team and
        m.recruiter=rec)
```

Fig. 17. Non-redundant contract for the operation *newMember*

5.1.3 More on the Strict and Extended Interpretations

As we have just seen, the strict interpretation solves the problem of redundancy in contracts in those cases in which the intended behavior of an operation is to avoid execution if it is about to violate an integrity constraint. However, in some specific situations, the designer's intention is not to reject the operation but to apply it while still maintaining the consistency of the information base. In those cases, an extended interpretation may be better. The next example is useful to illustrate this situation.

If we want to define an operation *upgradeEmployee* aimed at promoting junior employees, we can specify the operation contract in Figure 18.

According to a strict interpretation, the minimal set of structural events that satisfies *Post* is *S*={*specialize(e, Senior), generalize(e, Junior)*}. Then, according to Definition 4.2, *IB′* is obtained from *IB* by inserting *e* as an instance of *Senior* and removing it from *Junior*.

Note that this operation always results in a state *IB′* that satisfies both the postcondition and the integrity constraints.

```
Operation:  upgradeEmployee(e:Junior)
Pre:
Post:       --the employee e becomes an instance of Senior
            --and ceases to be an instance of Junior
            e.oclIsTypeOf(Senior) and not e.oclIsTypeOf(Junior)
```

Fig. 18. Contract for the operation *upgradeEmployee*

Clearly, the previous semantics is the one expected for the operation *upgradeEmployee*. However, the postcondition of the contract explicitly states that *e* must no longer be a junior employee. Moreover, the class diagram already includes the condition that if *e* is *Senior* (as also enforced by the contract postcondition) it may not be *Junior* (because of the disjointness constraint). Thus, we could argue that the same behavior would be achieved by removing *not e.oclIsTypeOf(Junior)* from the postcondition, as shown in Figure 19.

```
Operation:  upgradeEmployee(e:Junior)
Pre:
Post:       --the employee e becomes an instance of Senior
            --and ceases to be an instance of Junior
            e.oclIsTypeOf(Senior)
```

Fig. 19. New contract for the operation *upgradeEmployee*

The problem is that a strict interpretation of the contract would never allow the operation *upgradeEmployee* to be applied, since the information base resulting from inserting *e* as a senior employee (the minimal set of structural events that would now satisfy the postcondition) would always violate the disjointness constraint. Hence, a strict interpretation requires *not e.oclIsTypeOf(Junior)* to be explicitly stated in the operation postcondition.

An extended interpretation does not involve the same drawback since we do not need to specify in the postcondition of *upgradeEmployee* that *e* is no longer junior. The reason is that the reactive behavior of an extended interpretation when a constraint is violated will be sufficient to detect that this change is also required without the need to state it explicitly.

Thus, according to Definition 4.1, the semantics of the contract in Figure 19 when an extended interpretation is assumed is as follows:

- a) is guaranteed.
- b) states that *IB′* = S_2*(IB)* and *S* $\subseteq$ S_2, *S*={*specialize(e, Senior)*}.
- Note that the application of *S* alone would violate the disjointness constraint of the *Employee* specialization. Then, according to c), S_2 must be a superset of *S*.
- The minimal superset of *S* that satisfies both *Post* and *IC* is S_2 = *S* $\cup$ *{generalize(e, Junior)}*.

Thus, the previous contract always requires the application of two structural events: {*specialize(e, Senior), generalize(e, Junior)*}. However, only one of them is explicitly stated in the postcondition.

The previous example shows that there are some specific situations in which an extended interpretation is not ambiguous without the need to specify additional information in the operation contracts. In this example, a disjointness constraint is violated, and since the only possible way to repair it is by deleting *e* as a junior employee, no ambiguity exists at all. In general, the extended interpretation of an operation contract will not be ambiguous when all integrity constraints that are violated by the execution of the operation only allow for a single repair.

Summarizing, the main differences between strict and extended interpretations lie in the way integrity constraints are enforced. When there is no violation of integrity constraints, the semantics of a given operation contract is equivalent in both interpretations. In this case, the new state of the information base is always obtained by applying the minimal set *S* of structural events that satisfy the operation postcondition. However, if the violation of a constraint occurs when the structural events in *S* are applied, the semantics of the contract depends on the chosen interpretation.

We summarize in Table 2 how (in addition to the intended behavior of the operation) the treatment of integrity constraints should be specified in a contract, depending both on the interpretation chosen and whether we want the operation to be rejected or applied when the events in *S* violate an integrity constraint.

Table 2. Avoiding violation of integrity constraints

	Reject the operation	**Apply the operation**
Strict	Nothing else needs to be done	Specify how to satisfy the constraints in *Post*
Extended	Add redundant checks to *Pre*	Specify how to satisfy the constraints in *Post* (if the contract is ambiguous)

On the one hand, under a strict interpretation the violation of a constraint means that the operation is not applied and, thus, the information base remains unchanged. Consequently, no redundant checks are needed in the precondition. A strict interpretation of the contract for *newWorkingTeam* in Figure 15 serves as an example of this situation. Alternatively, if we do not want the operation to be rejected, its postcondition must explicitly state how to satisfy the constraints after execution of the operation. This can be seen in the contract for *upgradeEmployee* in Figure 18.

On the other hand, under an extended interpretation, the operation may be applied even though some constraint is violated by *S*, and the new state of the information base is obtained by applying a set of structural events *S′*, a superset of *S*, that guarantees that no constraint is violated. Note that *S′* is unique as long as there is a single way to guarantee the constraints, as just shown with a disjointness constraint violation. If *S′* is not unique the operation contract is ambiguous. To avoid ambiguities, we must specify the way to preserve consistency in the postcondition (as must be made in a strict interpretation).

Alternatively, if we want to maintain consistency by rejecting the operation, its precondition must include redundant checks to ensure that execution of the operation does not violate the constraints. An example of this situation can be found in the contract for the operation *newWorkingTeam* specified in Figure 14.

5.1.4 Discussion

We have compared strict and extended interpretations from the point of view of the relevant characteristics of a good software requirements specification (SRS) (Various Authors 1998; Davis 1993). As recommended there, we assumed an unambiguous operation contract specification. In general, this is always true in a strict interpretation, since deterministic behavior is usually desired. Regarding the extended interpretation, in this section we will concentrate on those situations in which non-ambiguity is achieved by strengthening either the operation pre- or postcondition, since this is the most frequent case in practice.

Completeness

An SRS is complete if it includes the definition of the responses of the software to all possible classes of input data in all possible classes of situations. From this point of view, both approaches can be considered complete. An extended interpretation avoids erroneous execution of an operation by means of its precondition, while a strict one assumes that the response to undesired situations is the rejection of the changes made by the operation. Moreover, when the precondition is not satisfied, both approaches act in the same way, by rejecting the operation.

For instance, understanding the operation *newWorkingTeam* (see Figure 14) from the point of view of an extended interpretation, we always obtain a defined result when the precondition is satisfied, which is exactly the one specified in the postcondition (a new working team with the specified name is created) plus the additional changes, if any, required to satisfy all integrity constraints.

Understanding *newWorkingTeam* from the point of view of a strict interpretation (see Figure 15), we have two kinds of results when the precondition holds. In those cases in which no integrity constraint is violated, the resulting state of the information base is the one specified in the postcondition, as occurs with an extended interpretation. On the other hand, when an integrity constraint is violated, *newWorkingTeam* is rejected and the information base remains unchanged.

Consistency

An SRS is consistent if, and only if, no conflict occurs between subsets of individual requirements described within it.

Although neither of the approaches leads directly to an inconsistent specification, a strict interpretation facilitates having a consistent one while an extended interpretation is more prone to the specification of conflicting requirements. The reason is that, since integrity constraints are sometimes specified both in the structural schema and as preconditions of operations in the behavioral schema, they can be in contradiction and, therefore, lead to an inconsistent specification.

For instance, it will be more difficult to keep the specification of *newMember* consistent with the operation contract specified in Figure 16 than with the contract in Figure 17. The reason is that, for example, we could easily have specified in the precondition that

```
team.employee->size()<=5
```

instead of

```
team.employee->size()<5
```

This would be clearly inconsistent with the cardinality constraint of *Employee* in the association *Member,* which forces a team to have at most 5 members.

Verifiability

An SRS is verifiable if, and only if, every requirement stated therein is verifiable. A requirement is verifiable if, and only if, there exists some finite cost-effective process with which a person or machine can check that the software product meets the requirement.

Unlike the previous criterion, verifiability is more easily achieved with an extended interpretation. Although both approaches allow the verification of the software product, this process can become more complicated when a strict interpretation is assumed due to the dispersion of the requirements that affect an operation.

For example, to verify the correct behavior of the operation *newWorkingTeam* as defined in Figure 15, we must also take into account the integrity constraint *uniqueWorkingTeam*. However, taking the contract in Figure 14, no additional information is needed in order to verify it.

Modifiability

An SRS is modifiable if, and only if, its structure and style are such that any changes to the requirements can be made easily, completely, and consistently. Modifiability generally requires that an SRS is not redundant.

In the case of modifiability, an extended interpretation is again more prone to errors due to the necessary duplication of integrity constraints in the preconditions. When changing a requirement, it is easy to forget to change it in every precondition in which it appears, and this can lead both to inconsistencies and to wasting more time.

Assume that a requirement changes and we want to increase the maximum number of members in a team, for example 10. In this case, the cardinality constraint must be changed in the class diagram in order to express this requirement. Moreover, with an extended interpretation we will also have to modify the precondition of the operation *newMember* (see Figure 16) stating again the same condition in order to be consistent, and ensure that the same is done in every contract affected by the change. However, with a strict interpretation, we do not need to make any additional changes, since requirements stated by integrity constraints are only stated in the structural schema.

Much less frequently, we find a similar drawback when the postcondition already specifies how to maintain a certain integrity constraint. This drawback is shared by both approaches. On the one hand, a strict interpretation always needs to specify this reactive behavior in the postcondition, while on the other, an extended interpretation requires doing the same to guarantee that there is only one possible reaction to the violation of each integrity constraint.

Conciseness

Given two SRS for the same system, each exhibiting identical levels of all the qualities mentioned previously, the SRS that is shorter is also better.

Taking conciseness into account, it is clear that the strict interpretation approach helps to obtain shorter specifications, since each integrity constraint is specified in exactly one place. This can easily be seen by comparing the contracts in Figures 16 and 15. It is clear that both of them have the same meaning, while the one in Figure 17, in addition to being correct under a strict interpretation, is shorter.

Summary

The following table summarizes the discussion in this section. Rows correspond to desirable properties of a good software requirements specification, while columns refer to the interpretations we have defined in this paper. The symbol ✓ in a cell denotes the appropriateness of the corresponding interpretation to achieve the property.

As we can see, completeness is achieved in both interpretations, while consistency, modifiability and conciseness are easier to achieve in a strict interpretation, and verifiability in an extended one. For this reason, we can conclude that in general terms a strict interpretation of operation contracts provides several advantages over an extended one in conceptual modeling.

Table 3. Comparison of the approaches

	Extended interpretation	**Strict interpretation**
Completeness	✓	✓
Consistency		✓
Verifiability	✓	
Modifiability		✓
Conciseness		✓

5.2 Translating a UML Conceptual Schema into Logic

Validation tests that consider the structural schema alone are aimed at checking that an instantiation fulfilling a certain property and satisfying the integrity constraints can exist. In this case, classes, attributes and associations can be translated into base predicates that can be instantiated as desired, as long as integrity constraints are satisfied, in order to find a state of the IB that proves a certain property, as we have seen in chapter 3.

However, if we want to include the behavioral schema in the validation, we must guarantee that the population of classes and associations is determined by the execution of operations. In other words, the state of the IB at a certain time *t* is just the result of all the operations that have been executed before *t*, since the instances of classes and associations cannot be created or deleted as desired. That is, an instance *i* exists at time *t* if, and only if, some operation has created it at some time before *t*, and no operation has removed it between its creation and *t*.

To guarantee that the population of classes and associations depends on the operations executed, we propose that operations are the basic predicates of our logic formalization, since their instances are directly created by the user. Classes and associations will be represented by means of derived predicates instead of basic ones, and their derivation rules will ensure that their instances are precisely given by the operations executed. Due to the advantages seen in the previous section, we assume a strict semantics of operations in the translation. However, the extended semantics could also be handled by our approach, as long as it incorporated in the translation process.

5.2.1 Deriving Instances from Operations

Classes and associations are represented by means of derived predicates whose derivation rules ensure that their instances are given by the occurrence of operations, which are the base predicates of our formalization of the schema.

To simplify the definition of the derivation rules, we have modified the translation of our previous proposal (Queralt and Teniente 2006a). In particular, instead of defining separate predicates for each attribute of a class, we add them as terms of the predicate representing the class, as long as their cardinality is 1, since the life of their instances coincides with that of the instances of the class to which they belong. We also need to add a term *t*, representing the time in which an instance exists.

For instance, the predicate representing the class *Employee* is, according to this new translation:

employee(E,Name,Salary,T)

instead of

employee(E)

employeeName(E,Name)

employeeSalary(E,Salary)

As will be seen, this change simplifies the derivation of the instances from the operations.

Also, for the same reason, in class hierarchies we define the superclass as derived from the instances of its subclasses. That is, instead of adding constraints in order to ensure that an instance of a subclass also belongs to the superclass, we define a derivation rule ensuring that when an instance of a subclass exists, then the corresponding instance of the superclass also does.

For instance, the translation of the subclass *Boss* will be:

boss(E,Name,Salary,Phone,T)

and the following rule will ensure that bosses are also employees:

employee(E,Name,Salary,T) $\leftarrow$ boss(E,Name,Salary,Phone,T)

Moreover, for each operation *op* with parameters $P_1,...,P_n$ defined in the schema, the following base predicate is defined:

op([O_1,..., O_m], P_1,...,P_n, T)

where the term *T* represents the time in which the operation occurs, and each O_i represents the OID of an instance created by the operation *op*.

For example, the predicate representing an occurrence of the operation *newDept(d-name: String, minSal, maxSal: Real, managerName: String, managerSal: Real)* is

newDept(D, DName, Min, Max, E, MgrName, MgrSal, T)

where *D* and *E* are the OIDs of the department and the employee created by the operation, *T* is the occurrence time of the operation, and the rest of terms correspond to the parameters defined in its signature.

Then, an instance of a predicate *p* representing a class or association exists at time *t* if it has been added by an operation at some time *t2* before *t*, and has not been deleted by any operation between *t2* and *t*. Formally, the general derivation rule is:

$$p([P,],P_1,...,P_n,T) \leftarrow addP([P,]P_1,...,P_n,T2) \wedge \neg deletedP(P_i,...P_j,T2,T) \wedge T2 \leq T \wedge time(T)$$

$$deletedP(P_i,...,P_j,T1,T2) \leftarrow delP(P_i,..,P_j,T) \wedge T>T1 \wedge T \leq T2 \wedge time(T1) \wedge time(T2)$$

where *P* is the OID (Object Identifier), which is included if *p* is a class. $P_i,...,P_j$ are the terms of *p* that suffice to identify an instance of *p* according to the constraints defined in the schema. In particular, if *p* is a class (or association class), $P=P_i=P_j$.

The predicate *time* indicates which are the time variables that appear in the derived predicate we are defining. As well as the predicates representing operations, *time* is a base predicate since its instances cannot be deduced from the rest of information in the schema.

Predicates *addP* and *delP* are also derived predicates that hold if some operation has created or deleted an instance of *p* at time *T*, respectively. They are formalized as follows.

Let *op-addP*$_i$ be an operation of the behavioral schema, with parameters $Par_1,...,Par_n$ and precondition pre_i such that its postcondition specifies the creation of an instance of a class or association *p*. For each such operation we define the following rule:

$$addP([P,]Par_i,...,Par_k,T) \leftarrow op\text{-}addP_i([P,]Par_1,...,Par_m,T) \wedge pre_i(T_{pre}) \wedge T_{pre}=T-1 \wedge time(T)$$

where *Par_i,..,Par_k* are those parameters of the operation that indicate the information required by the predicate *p*, that is, terms corresponding to the attributes of *P*, and *T* is the time in which the operation occurs. The literal *$pre_i(T_{pre})$* is the translation of the precondition of the operation, following the same rules used to translate OCL integrity constraints explained in section 3.1. Note that, since the precondition must hold just before the occurrence of the operation, the time *T_{pre}* of all its facts is *T-1*. If some of the parameters corresponds to the OID of an object, then the existence of this object must also be guaranteed at *T_{pre}*. Thus, this condition will be added as a precondition in the translation.

Similarly, for each operation *$op\text{-}delP_i(Par_1,...,Par_n,T)$* with precondition *$pre_i$* that deletes an instance of *p* we define the derivation rule:

$$delP(Par_i,...Par_j,T) \leftarrow op\text{-}delP_i(Par_1,...,Par_n,T) \wedge pre_i(T_{pre}) \wedge T_{pre}=T\text{-}1 \wedge time(T)$$

where *Par_i,...,Par_j* are those parameters of the operation that identify the instance to be deleted. Thus, if *p* is a class or association class, *delP* will have a single term in addition to *T*, which corresponds to the OID of the deleted instance.

To completely define the above derivation rules for each predicate representing an element of the structural schema, we need to know which OCL operations of the behavioral schema are responsible for creating or deleting its instances. For our purpose, we assume that operations create instances with the information given by the parameters or delete instances that are given as parameters. A single operation can create and/or delete several instances. We are not interested in query operations since they do not affect the correctness of the schema.

Several OCL expressions can be used to specify that an instance exists or not at postcondition time. For the sake of simplicity, we consider a single way to specify each of these conditions, since other OCL expressions with equivalent meaning can be easily rewritten in terms of the ones we consider. Under this assumption, we define the rules to identify the creation and deletion of instances in OCL postconditions:

R1. An instance *$c(I,A_1,...,A_n,T)$* of a class *C* is added by an operation if its postcondition includes the OCL expression:

```
C.allInstances()-> exists(i| i.oclIsNew() and i.attri=ai)
```

or the expression:

```
i.oclIsTypeOf(C) and i.attri=ai
```

where each *$attr_i$* is a single-valued attribute of *C*.

R2. An instance *$c(I,P_1,...,P_n,A_1,...,A_m,T)$* of an association class *C* is added by an operation if its postcondition includes the OCL expression:

```
C.allInstances()-> exists(i| i.oclIsNew() and i.part1=p1 and
... and i.partn=pn and i.attr1=a1 and ... and i.attrm=am)
```

or the expression:

```
i.oclIsTypeOf(C) and i.part1=p1 and ... and i.partn=pn and
i.attr1=a1 and ... and i.attrm=am
```

where each *$part_i$* is a participant that defines the association class, and each *$attr_j$* is a single-valued attribute of *C*.

R3. An instance *r(C_1,C_2,T)* of a binary association *R* between objects *C_1* and *C_2*, with roles *role-c_1* and *role-c_2* in *r* is added by an operation if its postcondition contains the OCL expression:
`c`$_i$`.role-c`$_j$` = c`$_j$, if the multiplicity of *role-c_j* is at most 1
or the expression:
`c`$_i$`.role-c`$_j$`-> includes(c`$_j$`)`, if the multiplicity of *role-c_j* is greater than 1.
This rule also applies to multi-valued attributes. Creation or deletion of instances of n-ary associations with n>2 cannot be expressed in OCL unless they are association classes, which are considered in the previous rule.

R4. An instance *c(I,A_1,...,A_n,T)* of a class *C* is deleted by an operation if its postcondition includes the expression:
`C`$_{gen}$`.allInstances()->excludes(i)`
or the expression:
`not i.oclIsTypeOf(C`$_{gen}$`)`
where *C_{gen}* is either the class *C* or a superclass of *C*.

R5. An instance *c(I,P_1,...,P_n,A_1,...,A_m,T)* of an association class is deleted by an operation if its postcondition includes the expression:
`C.allInstances()->excludes(i)`
or the expression:
`not i.oclIsTypeOf(C)`
or if any of its participants (*P_1,...,P_n*) is deleted.

R6. An instance *r(C_1,C_2,T)* of a binary association *R* between objects *C_1* and *C_2*, with roles *role-c_1* and *role-c_2* in *r* is deleted by an operation if its postcondition includes the OCL expression:
`c`$_i$`.role-c`$_j$` ->excludes(c`$_j$`)`
or if any of its participants (*C_1* or *C_2*) is deleted.

According to these rules, class hierarchies are considered as follows. Let *Super* be the superclass of a hierarchy, and *Sub* be a subclass. When an instance of *Super* is explicitly created in a postcondition, the translation guarantees the creation of an instance only in the superclass. On the contrary, when an instance of *Sub* is created in a postcondition, the translation ensures the creation of this instance both in *Sub* and *Super*, since it is not possible that an instance belongs to a subclass and not to its superclass. This is achieved by the derivation rule that we have defined, stating that an instance of a subclass implies the existence of the same instance in the superclass.

Regarding deletions, if an instance of *Sub* is deleted in a postcondition, it is deleted only fromo *Sub* in our translation, since it may belong only to *Super*. In contrast, when an instance of *Super* is deleted, it must be deleted also from all the subclasses to which it belongs, although it is not explicitly stated in the postcondition.

For instance, according to the previous translation rules, the class *Employee* of our example will be represented by means of the clauses:

employee(E,Name,Sal,T) $\leftarrow$ addEmployee(E,Name,Sal,T2) $\wedge$ ¬deletedEmployee(E,T2,T) $\wedge$ T2≤T $\wedge$ time(T)

deletedEmployee(E,T1,T2) $\leftarrow$ delEmployee(E,T) $\wedge$ T>T1 $\wedge$ T≤T2 $\wedge$ time(T1) $\wedge$ time(T2) where *E* corresponds to the unique OID required by every instance of a class. In turn, *addEmployee* and *delEmployee* are derived predicates whose definition depends on the operations of the behavioral schema that insert and delete instances of the class *Employee*. The operation *hire* creates an instance of *employee(E,N,S,T)* according to R1, since its postcondition includes the expression `Employee.allInstances()->exists(e| e.oclIsNew() and e.name=e-name and e.salary=sal and e.working-dep=dep)`. The last equality indicates the creation of an instance of the association *WorksIn* according to R3. Since the operation *newDept* also creates an instance of *Employee*, there are two derivation rules for *addEmployee*:

addEmployee(E,Name,Sal,T) ← hire(E,Name,Sal,Dep,T)
∧ department(Dep,DName, MinSal, MaxSal,T_{pre})
∧ T_{pre} = T-1 ∧ time(T)

addEmployee(E,Name,Sal,T) ← newDept(D,N,Min,Max,E,Name,Sal,T_{pre})
∧ T_{pre} = T-1 ∧ time(T)

The first rule corresponds to the operation *hire*. Since this operation has a parameter corresponding to an instance of *Department*, a literal ensuring that the corresponding department exists at T_{pre} is needed. The second rule corresponds to the operation *newDept*, which does neither have any precondition, nor any object as a parameter.

The operation *hire* also creates an instance of the association *WorksIn*, so we also include the following rule in our logic formalization of the schema:

addWorksIn(E,Dep,T) ← hire(E,Name,Sal,Dep,T)
∧ department(Dep,DName, MinSal, MaxSal,T_{pre})
∧ T_{pre} = T-1 ∧ time(T)

which belongs to the derivation of the predicate *worksIn(E,D,T)*.

We also need to find which operations are responsible for deleting instances of *Employee* in order to specify the derivation rule of *delEmployee.* The operation *fire* is the only one that deletes instances of *Employee* according to R4, since it includes the OCL expression `Employee.allInstances()->excludes(emp)`. Now the precondition is not empty, and requires that the employee to be deleted does not belong to any department. This operation has an instance of *Employee* as a parameter, so the derivation rule in this case is:

delEmployee(E,T) ←fire(E,T) ∧ employee(E,N,S,T_{pre}) ∧ ¬hasDep(E,T_{pre})
∧ T_{pre} = T-1 ∧ time(T)

hasDep(E,T) ← worksIn(E,D,T) ∧ time(T)

That is, in addition to the literal *fire(E,T)* corresponding to the operation, and the literal *¬hasDept(E,T_{pre})* corresponding to the precondition, the literal *employee(E,N,S,T_{pre})* ensuring that *E* exists at precondition time is added since the first term of the predicate *fire* corresponds to an instance of employee, which must exist in order to execute the operation.

According to R5 and R6, all the instances of the associations in which the deleted employee may participate must also be deleted, since they cannot exist without one of its participants. Thus, the following rules are also generated, that will be used in the derivation rules of the corresponding associations. They derive the deletion of all the affected associations when an instance of *Employee* is deleted:

delWorksIn(E,D,T) ← delEmployee(E,T) ∧ worksIn(E,D,T_{pre}) ∧ T_{pre}=T-1 ∧ time(T)

delManages(E,D,T) ← delEmployee(E,T) ∧ manages(E,D,T_{pre}) ∧ T_{pre}=T-1 ∧ time(T)

delWorksFor(E,E2,T) ← delEmployee(E,T) ∧ worksFor(E,E2,T_{pre}) ∧ T_{pre}=T-1 ∧ time(T)

delWorksFor(E2,E,T) ← delEmployee(E,T) ∧ worksFor(E2,E,T_{pre}) ∧ T_{pre}=T-1 ∧ time(T)

delAudits(E,W,T) ← delEmployee(E,T) ∧ audits(E,W,T_{pre}) ∧ T_{pre}=T-1 ∧ time(T)

delMember(M,T) ← delEmployee(E,T) ∧ member(M,E,W,T_{pre}) ∧ T_{pre}=T-1 ∧ time(T)

That is, for each association *Assoc* in which the class *Employee* participates, we add a derivation rule *delAssoc*, defined by the occurrence of a deletion of *Employee*, and also by an additional literal that determines which instances of *Assoc* must be deleted, which are the ones in which the employee *E* participates at time T_{pre}. In this example, note that, since the association *WorksFor* is recursive, two rules are needed: one for the case in which the deleted employee is the superior, and another for the rest of employees. Note also that, since *Member* is an association class, its OID is used instead of its participants in the head of the rule.

Additionally, according to R4, since *Boss* is a subclass of *Employee*, the deletion of an employee *E* implies also the deletion of this employee as an instance of *Boss* if *E* also belongs to this subclass. Thus, the following rules must be added:

delBoss(E,T) ← delEmployee(E,T) ∧ time(T)

Since this subclass Boss does not participate in any association, no additional rules are needed. In case there was an association Assoc with the class Boss as a participant, we should add the corresponding rule to delete the instances of Assoc when delBoss occurs, in the same way as we have done with all the associations of Employee.

A modification can be regarded as a deletion followed by an insertion and, thus, no specific derived predicates are needed to deal with them. To exemplify the migration of an instance from one subclass to another, as happens in the operation *upgradeEmployee* specified in Figure 18, assume that employees can be classified either as junior or senior, and that we have also specific operations to create instances of these subclasses (*hireJunior* and *hireSenior*, respectively). We omit all the attributes for the sake of simplicity. Then, the translation of this hirarchy is:

junior(E,T) ← addJunior(E,T2) ∧ ¬deletedJunior(E,T2,T) ∧ T2≤T ∧ time(T)

deletedJunior(E,T1,T2) ← delJunior(E,T) ∧ T>T1 ∧ T≤T2 ∧ time(T1) ∧ time(T2)

addJunior(E,T) ← hireJunior(E,T) ∧ time(T)

delJunior(E,T) ← upgradeEmployee(E,T) ∧ time(T)

delJunior(E,T) ← fire(E,T) ∧ time(T)

$senior(E,T) \leftarrow addSenior(E,T2) \wedge \neg deletedSenior(E,T2,T) \wedge T2 \leq T \wedge time(T)$

$deletedSenior(E,T1,T2) \leftarrow delSenior(E,T) \wedge T>T1 \wedge T \leq T2 \wedge time(T1) \wedge time(T2)$

$addSenior(E,T) \leftarrow hireSenior(E,T) \wedge time(T)$

$addSenior(E,T) \leftarrow upgradeEmployee(E,T) \wedge time(T)$

$delSenior(E,T) \leftarrow fire(E,T) \wedge time(T)$

That is, an instance of *Junior* is deleted if the employee is fired or is promoted to senior. This means that an additional derivation rule of the predicate *delJunior* is required. Similarly, a new instance of *Senior* exists if it is hired or if some junior employee is upgraded to senior, which implies the definition of an additional derivation rule for *addSenior*. The operation *upgradeEmployee*, which deletes an instance of *Junior* and adds an instance of *Senior*, appears in the definition of both new rules.

5.2.2 Constraints Generated

Since events cannot happen simultaneously, we need to define constraints to guarantee that two operations cannot occur at the same time. As usual, constraints are expressed as formulas in denial form, which represent conditions that cannot hold in any state of the IB. Therefore, for each operation *o* with parameters $P_1,\ldots,P_n$ we define the following constraint for each parameter P_i:

$\leftarrow o(P_11,\ldots,P_n1,T) \wedge o(P_12,\ldots,P_n2,T) \wedge P_i1 <> P_i2$

And for each pair *o, o2* of operations we define the constraint:

$\leftarrow o(P_1,\ldots,P_n,T) \wedge o2(Q_1,\ldots,Q_m,T)$

In our example, *newTeam(W,N,I,T)* requires the constraints:

$\leftarrow newTeam(W,N,I,T) \wedge newTeam(W2,N2,I2,T) \wedge W <> W2$

$\leftarrow newTeam(W,N,I,T) \wedge newTeam(W2,N2,I2,T) \wedge N <> N2$

$\leftarrow newTeam(W,N,I,T) \wedge newTeam(W2,N2,I2,T) \wedge I <> I2$

and, for each other operation of the schema, a constraint like:

$\leftarrow newTeam(W,N,I,T) \wedge newDept(D,N2,Min,Max,E,MgrN,MgrS,T)$

Moreover, the constraints of the UML structural schema are also translated into this kind of formulas. The set of constraints needed is exactly the one resulting from the translation of the structural schema (Queralt and Teniente 2006a), but now they are defined in terms of derived predicates instead of basic ones.

The complete translation of the behavioral part of our example can be found in the Appendix B.

5.3 Validation Tests

As when validating the structural schema alone, our approach to validation consists in providing the designer with a set of tests that allow to assess the correctness of the conceptual schema. However, now they take into account both the structural and the behavioral parts of the schema. Additionally, some new tests regarding the correct definition of operations appear in order to ensure, for example, that each operation can be successfully executed or that it does not have any redundant preconditions.

As before, we express all the tests in terms of checking the satisfiability of a derived predicate. However, the sample instantiations resulting from the tests will be different, since they will be composed of the basic predicates we have now in our formalization, which correspond to instances of operations. For example, a sample instantiation obtained from the validation of the structural schema was composed of instances of classes and associations:

{employee(john), department(sales), ...}

Now the sample instantiations will be a sequence of operation calls that leads to a valid state, since operations are the base predicates. The term representing the occurrence time of operations gives the order in which the operations must be executed to be successful. The instances of classes and associations can be obtained from this result by means of their derivation rules:

{newDept(sales,..., 1), hire(john,..., 2), ..., time(1), time(2), ...}

For the sake of clarity, those terms representing OIDs are omitted in the sample instantiations.

5.3.1 Is the Conceptual Schema Right?

In this section we review some of the tests aimed at checking the internal correctness of the schema. The definition and formalization of the properties to be checked (satisfiability of the schema, liveliness of a class or association and the rest of automatically generated validation tests) is exactly the same as when the structural schema alone is validated. However, the new translation of the schema, now taking into account the behavioral part, ensures that the only changes allowed are those defined in the operations specified.

We exemplify this validation by re-executing some of the tests performed on the structural schema, to show how they results change when considering the operations. Also, we define some new tests formalizing additional properties, such as applicability and executability (Costal, Teniente et al. 1996), to check the correctness of the operations themselves.

Satisfiability of a Schema

As when taking into account only the structural part of the schema, a complete schema is satisfiable if there is a non-empty state of the IB in which all its integrity constraints are satisfied. In the presence of operations, this means checking whether they allow creating at least a non-empty valid instantiation.

As before, the formalization of the test is as follows, for each *class$_i$* and *association$_j$* in the schema, now taking into account the new translation of classes:

sat ← $class_i$(X,...,T)

sat ← $association_j$(X,...,T)

The difference is that, with the new logic formalization of the schema that considers operations, each *$class_i$* and *$association_j$* is, in turn, a derived predicate defined by the rules explained in the previous section. This restricts the cases in which the schema is satisfiable to those instantiations that can be obtained by means of the operations and satisfy the integrity constraints at the same time.

For this reason, although the structural part of the schema of our example is satisfiable on its own, it is not satisfiable when taking the operations into account.

The instantiation obtained when checking the satisfiability of the structural schema is:

{employee(john), worksIn(john,sales), department(sales), manages(john,sales)}

However, this state cannot be created with the operations given, since the operation that creates *Employees* requires an existing instance of *Department* to be executed. But a department can never exist, since the operation *newDept*, that should create a new department, does never succeed because the cardinality constraint 1..* in *worker* is always violated after its execution. Since it is impossible to create *Employees* and *Departments,* the classes *Boss* (subclass of *Employee*) and *WorkingTeam* (that requires at least two *Employees*) will also be always empty, as well as all the associations of the schema. This means that the schema is not satisfiable at all with the operations defined.

To repair this erroneous specification of the schema, several corrective actions can be taken. On the one hand, we can weaken the constraints of the structural schema so that the operations we have defined are successful. In particular, we can change the constraint 1..* of *worker* to simply * and remove the constraint *ManagerIsWorker*. In this way, which implies that departments can be created before knowing which employees are assigned to them, the behavioral schema does not have to be modified.

On the other hand, we can leave the structural schema as is, and define the operations in such a way that they guarantee the constraints after execution. For example, the operation *newDept*, as well as creating a department with its corresponding manager, can also be responsible for assigning this manager as a worker at the same time, which is the minimum required in order to satisfy all the constraints:

```
Op:   newDept(d-name: String, minSal, maxSal: Real,
      managerName: String, managerSal: Real)
Pre:
Post: Department.allInstances()-> exists(d | d.oclIsNew() and
      d.name=d-name and minSalary=minSal and maxSalary=maxSal
      and Employee.allInstances->exists(e | e.oclIsNew() and
      e.name=managerName and e.salary=managerSal and d.manager=e
      and d.worker->includes(e)))
```

Fig. 20. Revised contract for the operation *newDept*

With this new operation contract instead of the initial one, the following sample instantiation proves that the schema is satisfiable:

{newDept(sales, 1500, 2500, john, 1500, 1)}

The execution of this operation will derive in the following IB:

{department(sales, 1500, 2500,1), employee(john, 1500,1), worksIn(john, sales,1), manages(john, sales,1)}

The corrective action taken will depend on what the designer considers is the best option according to the domain to be represented.

Liveliness of a Class or Association

We will check now the liveliness of the class *WorkingTeam,* which proved to be lively in our structural schema. The test we must execute is:

livelyWorkingTeam ← workingTeam(X, Name, T)

We can see that it is not possible to create an instance of *WorkingTeam* with the operations defined in Figure 3. The reason is that the only operation that can create teams is, again, not correctly defined.

The opertion *newTeam* creates an instance of *WorkingTeam* and assigns an existing *Employee* as its inspector. However, each working team requires at least one member, that cannot be the same inspector, and this is not considered in the operation. Note that an operation *newMember*, which assigns employees to teams, exists but, however, the cardinality constraint 1..* fo member is always violated after the execution of *newTeam*.

Assume, then, that the cardinality constraint 1..* of *Member* is changed to *. This does not solve the problem, because the OCL constraint *OneRecruited* still forces that each working team has at least one member. So the solution that requires less changes in the structural schema is to redefine the contract of *newTeam,* similarly to what has been done with *newDept* in Figure 20, but taking into account that we need to create at least two members in order to satisfy the constraint *OneRecruited*.

As can be seen, this operation is rather strange, since it creates not only a team with its inspector and a member, but also another member to recruit the first one. In view of this, probably the best option would be to remove or modify the constraint *OneRecruited*, since it forces to specify a quite unusual behavior in order to satisfy it.

```
Op:     newTeam(t-name:  String,   insp:  Employee,   member:
        Employee, recruiter: Employee)
Pre:    Department.allInstances()->notEmpty()
Post:   WorkingTeam.allInstances()-> exists(t | t.oclIsNew()
        and t.name = t-name and t.inspector = insp and
        t.member->exists(m | m.oclIsNew() and
        m.employee=member and recruiter.member->exists(r |
        r.oclIsNew() and r.working-team=t and
        m.recruiter=r)))
```

Fig. 21. Redefinition of the operation *newTeam*

Automatically Generated Tests

When dealing with operations, we can apply the same automatically generated tests that we have defined for the structural schema. However, the results can be different from those obtained in that case. Now this kind of tests detect situations that are potentially admitted by the class diagram but that are prevented either by the constraints, as before, or by the operations defined.

For instance, we will apply the Test 1, defined in section 3.2.1 to our recursive association *HasRecruited*, in order to test whether the operations allow to create as many instances as stated by its maximum cardinality, specified as *. Since this maximum cardinality is undetermined, we will test if it is possible that an employee has recruited at least two other members:

maxCardHasRecruited ← hasRecruited(M,M2,T) ∧ hasRecruited(M,M3,T) ∧ M2≠M3

This predicate is not satisfiable with our operations since, as can be seen from the operation contracts, it is impossible to create such an instantiation: the only operation that populates this association is *newTeam*, but only uses members that are created in the same operation, so an existing member can never recruit another one and, thus, the maximum number of possible recruitments is one per member. This means that this element of the structural schema has been overlooked when defining the behavior, so an additional operation with this purpose should be added.

If we apply now the Test 2 to the association *Member* we will be able to see that, in fact, the minimum cardinality of employee is greater than 1 according to the operations:

minCardMember ← member(M,E,W,T) ∧ ¬anotherOne(E,W,T)

anotherOne(E,W,T) ← member(M2,E2,W,T) ∧ E≠E2

The absence of a sequence of operations satisfying *minCardMember* shows that the conceptual schema does not admit teams with a single member and, therefore, that the cardinality constraint of *employee* is in fact stronger according to the instances really provided by the operations.

Applicability of an Operation

An operation is applicable if there is a state where its precondition holds. This property can be formalized as follows, for an operation *O* with precondition *pre*:

applicableO ← pre(T)

If the predicate *applicableO* is not satisfiable, then de operation can never be applied.

A simple example can be seen in the operation contract of *fire* in Figure 3, which deletes the indicated employee *emp*.

```
Op:     fire(emp: Employee)
Pre:    emp.working-dep->isEmpty()
Post:   Employee.allInstances()->excludes(emp)
```

However, the precondition of this operation requires that *emp* does not work in any department, which is not possible due to the cardinality constraint 1 of *working-dep*. Thus, this operation is not applicable, that is, it can never be executed because its precondition can not be satisfied. This means that, probably, this precondition should be removed.

Executability of an Operation

Although an operation is applicable, it may never be successfully executed because it always leaves the IB in an inconsistent state. Thus, another interesting property to be checked is the executability of an operation. An operation is executable if it can be executed at least once, that is, if there is a state where its postcondition holds, together with the integrity constraints, and such that its precondition was also true in the previous state.

To illustrate this property, let us consider the contract of the operation *removeDept* in Figure 3:

```
Op:     removeDept(dep: Department)
Pre:
Post:   Department.allInstances()->excludes(dep)
```

This operation is applicable, since its precondition can be satisfied, but the postcondition removes a department, which necessarily has some worker according to the cardinality constraint 1..* of *worker.* Since this operation does not remove the employees that work in the department, the resulting state of the IB will always violate the cardinality constraint 1 of *working-dep* for all the employees that were assigned to *dep* before the execution of the operation. This means that this operation will always be rejected because it is impossible to satisfy its postcondition and the integrity constraints at the same time.

An additional example can be seen with the operation *promote,* also in Figure 3:

```
Op:     promote(emp: Employee, phone: String)
Pre:    emp.managed-dep->isEmpty()
Post:   emp.oclIsTypeOf(Boss) and emp.oclAsType(Boss).phone = phone
```

The precondition indicates that the employee to be promoted must not be the manager of any department. Since it is possible to find an employee that is not a manager, this operation is applicable. However, after the application of *promote* the constraint *BossIsManager* is always violated, so this operation is not executable. Probably, the best option would be to change the precondition to

```
emp.managed-dep->notEmpty()
```

In this way, the intuitive semantics of the operation is that only employees that are managers can be promoted.

To check executability, an additional rule has to be added to the translation of the schema to record the execution of the operation. In this case, if *executedO* is satisfiable, then *O* is executable:

$\text{executedO} \leftarrow \text{o}(P_1,\ldots,P_n,T) \wedge \text{pre}(T_{pre}) \wedge T_{pre} = T\text{-}1$

In our example, we would define this rule for all the operations of the schema. For instance, the rules corresponding to the operations that we have just discussed are:

$\text{executedRemoveDept} \leftarrow \text{removeDept}(D,T)$

$\text{executedPromote} \leftarrow \text{promote}(E,P,T) \wedge \neg\text{isManager}(E,T_{pre}) \wedge T_{pre}=T\text{-}1$

$\text{isManager}(E,T) \leftarrow \text{manages}(E,D,T)$

If the predicate representing the operation can be successfully inserted, that is, it is possible to satisfy all the constraints defined when deriving its effects, then an instance of the corresponding *executedO* will exist, which proves that the operation can be executed.

Redundancy of a Precondition

This test consists in checking whether the preconditions of the operations are really necessary, that is, they are not already guaranteed by the rest of the schema. This is tested by trying to find a state of the IB that violates a precondition. If this is not possible, the precondition is unnecessary. This test can be defined for each precondition *pre* of each operation.

redundantPre ← pre(T)

As an example, see the precondition of *newTeam*, which states that a department must exist in order to create an instance of *WorkingTeam*.

```
Op:     newTeam(t-name:  String,  insp:  Employee,  member:
        Employee, recruiter: Employee)
Pre:    Department.allInstances()->notEmpty()
Post:   WorkingTeam.allInstances()-> exists(t | t.oclIsNew()
        and t.name = t-name and t.inspector = insp and
        t.member->exists(m | m.oclIsNew() and
        m.employee=member and recruiter.member->exists(r |
        r.oclIsNew and r.working-team=t and m.recruiter=r)))
```

This is always true, since a team requires an inspector and two members, which must necessarily work in a department. Thus, we can remove this precondition and the behavior of the schema remains unchanged, while its modifiability is increased.

5.3.2 Is It the Right Conceptual Schema?

Once we know that the conceptual schema is right, we may wonder whether *it is the right schema* in the sense that it satisfies the requirements of the domain. As happens in the validation of the structural schema, some of the tests to check the external correctness can be automatically generated. Additionally, our approach allows also testing whether a certain desirable state that the designer may envisage is acceptable or not according to the schema, this time taking into account the operations defined. Such state may be defined either by means of a set of instances that classes and associations should contain at least; or by a derived predicate, which defines it declaratively.

In both cases, either for the automatically generated or the user-defined tests, once a test is executed, the designer should compare the obtained results to those expected according to the requirements and apply modifications to the conceptual schema if necessary.

The validation process works in the same way as explained for the structural schema. The same tests, both the automatically generated and the user-defined, can be applied to the complete schema, possibly with different results. In the following we exemplify it by means of some additional user-defined tests.

An interesting question that has not been answered yet is *"May an employee recruit a member for a team to which he does not belong?."* To test this situation, the designer should define the rule:

recruiterNotMember ← hasRecruited(R,M,T) ∧ member(M,E,W,T)
∧ ¬isMember(R,W,T)

isMember(R,W,T) ← member(R,E,W,T)

In this case, *recruiterNotMember* is satisfiable, as shown by the sample instantiation:

{newDept(sales, 1500, 2500, john, 1000, 1), hire(mary, 1500, sales, 2),

hire(susan, 1500, sales, 3), newTeam(team1, mary, susan, john, 4),

newTeam(team2, john, mary, susan, 5), hire(peter, 1500, sales, 6),

newMember(peter, team1, maryInTeam1, 7)}

That is, *Mary* is the inspector of *team1* and belongs only to *team2*. However, she has been able to recruit *peter* to *team1,* where she does not belong. This probably does not correspond to the reality, since it would be reasonable that recruitments are made within a team. Thus, the conceptual schema should forbid a member to recruit another one in a different team by defining an additional constraint in the structural schema or by strengthening the precondition of the operation *newMember*.

In our case, probably the best option would be to modify the constraint *OneRecruited* so that it ensures, not only that at least one of the members of each team is recruited by another member in the same team, but also that all the members are recruited by members of the same team. This means, in fact, removing *OneRecruited* and adding the following constraint instead:

```
context Member inv CorrectRecruitments:
     self.member->forall(m| m.workingTeam = self.workingTeam)
```

A side effect of this change is that the operation *newTeam* can be also simpler, since now a team only requires a single member to be created.

By studying the results of the previous tests, and with his knowledge about the requirements of the system to be built, the designer will be able to decide if the schema is correct, and perform the required changes if not.

5.4 A Reasoning Procedure

When we incorporate the behavior of operations in the translation of the schema, several levels of derivation appear. A limitation of the reasoning procedure that we have developed to validate the structural schema is that it may only be applied when the logic formalization of the schema has at most one level of derivation. Thus, this procedure cannot be used to validate a complete conceptual schema with operations.

As a consequence, in this case we are not able to ensure that any reasoning task will terminate, and we have to use an already existing method to perform the tests. The most appropriate method to perform the kind of integrity maintenance we require is the CQC-Method (Farré, Teniente et al. 2005). However, this method has had to be extended in order to incorporate a correct treatment of the time component of our atoms.

The CQC Method is a semidecision procedure for finite satisfiability and unsatisfiability. This means that it always terminates if there is a finite example or if the tested property does not hold. However, it may not terminate in the presence of solutions with infinite elements.

Roughly, the CQC Method is aimed at constructing a state that fulfills a goal and satisfies all the constraints in the schema. The goal to attain is formulated depending on the specific reasoning task to perform. In this way, the method requires two main inputs besides the conceptual schema definition itself. The *goal to attain*, which must be achieved on the state that the method will try to construct; and the set of *constraints to enforce*, which must not be violated by the constructed state.

Then, to check if a certain property holds in a schema, this property has to be expressed in terms of an initial goal to attain (G_0) and the set of integrity constraints to enforce (F_0), and then ask the CQC Method to attempt to construct a sample IB to prove that the initial goal G_0 is satisfied without violating any integrity constraint in F_0.

This means that, to perform our validation tests, we need to provide the CQC Method with the formalization of our schema, i.e. the derived predicates that represent classes and associations, the set of constraints of the schema as F_0 and the derived predicate formalizing the validation test to perform as G_0.

5.4.1 Variable Instantiation Patterns

The CQC Method performs its constraint-satisfiability checking tests by trying to build a sample state satisfying a certain condition. For the sake of efficiency the method tests only those variable instantiations that are relevant, without losing completeness. The method uses different *Variable Instantiation Patterns (VIPs)* for this purpose according to the syntactic properties of the schema considered in each test. The key point is that the VIPs guarantee that if the variables in the goal are instantiated using the constants they provide and the method does not find any solution, then no solution exists.

The VIP in which we are interested is the *discrete order VIP*. In this case, the set of constants is ordered and each distinct variable is bound to a constant according to either a former or a new location in the total linear order of constants maintained. The value of new variables is not always *static* (i.e. a specific numeric value), it can be a relative position within the linear ordering of constants. These are called *virtual constants*. For instance, in the ordering of constants {1, *d*, 6}, *d* is a virtual constant such that $1<d<6$. Then, its possible absolute values are 2 to 5. It may happen that the goal succeeds or fails without the need for further instantiations, and in this case *d* will never be bound to a concrete value.

To correctly instantiate the variables representing occurrence times that we have introduced in our translation of the conceptual schema, it has been necessary to add a *temporal VIP*. This new VIP has some similarities with the *discrete order VIP,* since they both deal with discrete values, order comparisons and negation, but it extends it to be able to bind a constant, either virtual or static, with its immediate successor. This is needed because our derivation rules require that preconditions hold exactly in the time immediately previous to the postcondition, not at any time before the postcondition. Then, we use a separate set of constants, with its own ordering, to deal with variables representing event times and we instantiate them with our *temporal VIP*.

For instance, assume we are attempting to derive an *Employee* which must hold at time *d,* being *d* a virtual constant and {1, *d,* 5} our set of temporal constants. According to the definition of the operation *hire*, which creates new employees, a department must exist at time *d-1*. Thus, since $1<d<5$, the time variable of the corresponding instance of *Department* must be instantiated either with 1 or with a virtual constant *f, f=d-1.* So, the relevant sets of constants are {[1, *d*], 5} and {1,[*f, d*],5}, where constants between brackets are tied so that no new constant can be ever placed between them.

The *temporal VIP* is formalized as follows. A variable instantiation step performs a transition from $(T\ \varnothing\ KT_i)$ to $(\varnothing\ \theta\ KT_{i+1})$ that instantiates the temporal variable T according to one of the VIP-rules, where θ is a ground substitution of T and KT_i is the set of temporal constants. Let d_i denote virtual constants, c_i denote static constants and k_i denote either static or virtual constants, and let G_c be the current goal. The *temporal VIP* consists of the VIP-rules of the *discrete order VIP*, extended by the following rules, that apply when instantiating a temporal constant T such that $T = k_i -1$, $k_i \in KT_i$:

Tmp1. $\theta = T / c_{prev}$ and $KT_{i+1}=KT_i$, where $c_{prev}=c_{suc}-1$, $\{c_{suc},c_{prev}\} \subseteq KT_i$, $\{T=c_{suc}-1\} \in G_c$

Tmp2. $\theta = T / k$ and $KT_{i+1} = KT_i$, where $\{k, k_{suc}\} \subseteq KT_i$, $\{T = k_{suc}-1\} \in G_c$, there is no constant k_{prev} such that $k<k_{prev}<k_{suc}$ and k is tied to k_{suc} in KT_{i+1}.

Tmp3. $\theta = T / c_{new}$ and $KT_{i+1} = KT_i \cup \{c_{new}\}$, where $c_{new}=c_{suc}-1$, $c_{new} \notin KT_i$, $c_{suc} \in KT_i$, $\{T = c_{suc}-1\} \in G_c$, there is no d_{prev} tied to c_{suc} in KT_i, and there is no $c_{prev} \in KT_i$ such that $c_{prev} < c_{suc}$ and $|\{d_i \mid d_i \in KT_i \text{ and } c_{prev} < d_i < c_{suc}\}| < |c_{suc} - c_{prev}| -1$.

Tmp4. $\theta = T / d_{new}$ and $KT_{i+1} = KT_i \cup \{d_{new}\}$, where $d_{new} \notin KT_i$, $d_{suc} \in KT_i$, $\{T = d_{suc}-1\} \in G_c$, there is no d_{prev} tied to d_{suc} in KT_i, d_{new} is tied to d_{suc} in KT_{i+1} and there are no $c_i, c_j \in KT_i$ such that $c_i < d_{suc} < c_j$, there is no c_m with $c_i < c_m < c_j$ and $|\{d_i \mid d_i \in KT_i \text{ and } c_i < d_i < c_j\}| < |c_j - c_i| -1$.

These rules mean that, when instantiating a temporal variable, the following relevant values must be tried. On the one hand, all the values provided by the discrete order VIP must be considered as alternatives to instantiate *T*:

- each one of the already existing constants
- a new virtual constant that is lower than all the existing ones
- a new virtual constant that is lower than the minimum static constant so far

- a new virtual constant between all the existing constants, in case it fits, i.e. there is a gap of at least 1 element between two consecutive constants
- a new virtual constant that is greater than the maximum static constant so far
- a new virtual constant that is greater than all the existing ones

Additionally, the values provided by the temporal VIP must be also considered:

- if k_i is static and k_i-1 belongs to KT_i, T is be instantiated with k_i-1 (Tmp1)
- if k_i is virtual, or if it is satic and it is tied to a virtual predecessor in KT_i, T is instantiated with its tied predecessor (Tmp2)
- if k_i is static and is not tied to its predecessor in KT_i, if the value k_i-1 fits in KT_i, T is instantiated with this value, which is added to KT_{i+1} (Tmp3)
- if k_i is virtual and is not tied to its predecessor in KT_i, if a new virtual constant d_{new} fits in KT_i, T is instantiated with d_{new}, which is tied to k_i in KT_{i+1} (Tmp4)

5.4.2 A Sample Execution

In the following we explain a sample execution of our extended CQC Method. The test consists in checking the liveliness of *Employee*, once the operation *newDept* has been redefined according to the contract in Figure 20. Recall that this contract specifies that *newDept* creates an instance *d* of *Department*, together with an *Employee e*, as well as instances of the associations *Manages* and *WorksIn* between *d* and *e* in order to satisfy all the constraints.

Since we assume this new definition of the operation, the following rule to derive an instance of *WorksIn* has to be added to the translation of the original schema:

addWorksIn(E,D,T) ← newDept(D,N,Min,Max,E,EN,ES,T) ∧ time(T)

The rest of the rules used in the example are defined in Appendixes A and B.

Figure 22 shows the steps needed to satisfy the goal.

The first goal to be achieved is an instance of the predicate *employee*. This is a derived predicate that is unfolded in the second step, by placing the body of its only derivation rule as the current goal (see Appendix B). The first literal of this body is selected in order to be treated. It corresponds also to a derived predicate that can be obtained in two different ways according to its derivation rules:

addEmployee(E,Name,Sal,T) ← hire(E,Name,Sal,Dep,T) ∧
department(Dep,DName, MinSal, MaxSal,T_{pre}) ∧
T_{pre} = T-1 ∧ time(T)

addEmployee(E,Name,Sal,T) ← newDept(D,N,Min,Max,E,Name,Sal,T) ∧ time(T)

We assume that the second rule is chosen, which will obtain the employee (a manager) by creating a new department.

Now, the literal *newDept* is selected. Since it is a base predicate, the goal will be satisfied if an instantiation of this predicate is added to the sample IB using the VIPs. We have chosen an instantiation that will lead to an IB satisfying all the constraints. In particular, we have given the value 0 to the terms representing the OID and the attribute *name* of the class *Department*, the value 1001 to the *minimumSalary* in order to satisfy the constraint *MinimumSalary,* and the value 1002 to the *maximumSalary* so that the constraint *CorrectSalaries* is not violated. Regarding the parameters representing attributes of the manager employee that is created, an 1 is assigned to the OID so that it is different from the one of the department, and the value 0 is given both to the name and the salary of the employee. Finally, the temporal variable is also instantiated with a 0, meaning that this operation occurs at time 0.

Current Goal	***Additions to the sample IB***
← <u>employee(E,N,S,T)</u>	
← <u>addEmployee(E,N,S,T2)</u> ∧ ¬deletedEmployee(E,T2,T) ∧ T2≤T ∧ time(T)	
← <u>newDept(D,ND,Min,Max,E,N,S,T2)</u> ∧ time(T2) ∧ ¬deletedEmployee(E,T2,T) ∧ T2≤T ∧ time(T)	
	{*newDept(0,0,1001,1002,1,0,0,0)*}
← <u>time(0)</u> ∧ ¬deletedEmployee(1,0,T) ∧ <u>0≤T ∧ time(T)</u>	
	{*time(0), time(1)*}
← <u>¬deletedEmployee(1,0,1)</u>	
[ ]	

Fig. 22. Goal satisfaction

In the next step, this instantiation is propagated to the rest of variables in the goal, and the literals selected to be treated are the ones corresponding to the time predicates. The temporal variable *T* must be instantiated so that it satisfies the conditions in the goal, and the value 1 is chosen.

The time predicates are added to the IB, and the part of the goal that remains to be satisfied consists in a negative literal which, in fact, is not achieved by means of additions to the IB, but by guaranteeing that the IB does not satisfy the fact *deletedEmployee(1,0,1).* It is clear that this fact does not hold in our IB since, according to its derivation rules:

deletedEmployee(E,T1,T2) ← delEmployee(E,T) ∧ T>T1 ∧ T≤T2 ∧ time(T1) ∧ time(T2)

delEmployee(E,T) ← fire(E,T) ∧ employee(E,Name,Sal,T_{pre}) ∧ ¬hasDep(E,T_{pre})
∧ time(T)

it requires an instance of the predicate *fire*, which does not belong to our sample instantiation. Thus, the goal has been achieved.

However, we must guarantee that all the constraints of the schema are satisfied by the sample IB. Figure 23 shows an example of how the constraints of the schema are guaranteed. We illustrate this process by means of the integrity constraint:

←department(D,N,Min,Max,T) ∧ ¬oneEmployee(D,T)

oneEmployee(D,T) ← worksIn(E,D,T)

which corresponds to the cardinality constraint 1..* of the association *WorksIn.*

Conditions to ensure
←department(D,N,Min,Max,T) ∧ ¬oneEmployee(D,T)
←addDepartment(D,N,Min,Max,T2) ∧ ¬deletedDepartment(D,T2,T) ∧ T2≤T ∧ time(T) ∧ ¬oneEmployee(D,T)
←newDept(D,N,Min,Max,E,MgrN,MgrS,T2) ∧ time(T2) ∧ ¬deletedDepartment(D,T2,T) ∧ T2≤T ∧ time(T) ∧ ¬oneEmployee(D,T)
← ¬deletedDepartment(0,0,T) ∧ 0≤T ∧ time(T) ∧ ¬oneEmployee(0,T)
← ¬deletedDepartment(0,0,1) ∧ ¬oneEmployee(0,1)

Fig. 23. Condition satisfaction

In the following, we omit the column showing the additions to the sample IB, since no new fact is added.

This second part of the process aims at ensuring that the IB constructed in the previous phase does not contain a department without any employee.

In the first step, the predicate *department* is going to be treated, by replacing it by its derivation rule:

department(D,Name,Min,Max,T) ← addDepartment(D,Name,Min,Max,T2) ∧ ¬deletedDepartment(D,T2,T) ∧ T2≤T ∧ time(T)

as shown in the second row of the figure. In turn, *addDepartment* is unfolded in the next step, according to its only derivation rule:

addDepartment(D,N,Min,Max,T) ← newDept(D,N,Min,Max,E,MgrN,MgrS,T) ∧ time(T)

Now, the selected literal corresponds to the operation *newDept*, which is unified with the facts included in the IB, in particular with {*newDept(0,0,1001,1002,1,0,0,0)*}. This instantiation is propagated to the rest of the rule, and the literal *time(T)* is selected, since it is the only positive one. It is also unified with the facts in the IB, and we show here the result for *T*=1.

In the last step, the condition states that it is not possible that the IB does neither entail *deletedDepartment (0,0,1)* nor *oneEmployee(0,1)*, i.e., the IB must entail *deletedDepartment (0,0,1)* or *oneEmployee(0,1)*. Thus, we return to the Goal Satisfaction phase in order to ensure that this new goal is satisfied, as shown in Figure 24. Note that only one of the literals in the condition is placed as a goal, since in case that the IB entails just one of them, then the condition is satisfied and the execution ends successfully.

Current Goal
← oneEmployee(0,1)
←worksIn(E,0,1)
← addWorksIn(E,0,T) ∧ ¬deletedWorksIn(E,0,T,1) ∧ T≤1 ∧ time(1)
← newDept(0,N,Min,Max,E,EN,ES,T) ∧ time(T) ∧ ¬deletedWorksIn(E,0,T,1) ∧ T≤1
← ¬deletedWorksIn(1,0,0,1)
[ ]

Fig. 24. Second phase of Goal Satisfaction

The predicate *oneEmployee* in the initial goal corresponds to the existance of an instance of *WorksIn* with the department 0, as can be seen in the second step. The third step consists in the unfolding of the predicate *worksIn* according to the translation of the schema:

worksIn(E,D,T) ← addWorksIn(E,D,T2) ∧ ¬deletedWorksIn(E,D,T2,T)
∧ T2≤T ∧ time(T)

Now the first literal is selected, and is unfolded according to the rule that we have added to repair the erroneous specification of the operation.

The predicate *newDept* is unified with the IB and time literals in the goal are also satisfied. What remains to be treated is a negative literal, which will be considered as a condition to ensure. We do not show this new consistency derivation, but intuitively it can be seen that the *deletedWorksIn(1,0,0,1)* does not hold in the IB, since the only operation that belongs to it is *newDept*, which does not belong to the derivation rules of *deletedWorksIn*.

Thus, the condition and the goal are satisfied, so we can conclude that *Employee* is lively. The IB constructed contains a sequence of operations that leads to such a state:

{*newDept(0,0,1001,1002,1,0,0,0)*}

6

Tool Implementation

In order to prove the feasibility of our approach, we have built a prototype that implements our method to validate the structural part of a conceptual schema. We have also implemented the reasoning procedure for the behavioral part as an extension of a Prolog implementation of the CQC Method, which is not shown in this chapter.

The input of our prototype to validate the structural schema is a UML class diagram, specified in Poseidon®, and a text file containing the OCL constraints. Then, the structural schema is automatically translated into logic, the decidability of reasoning is determined and, finally, the user is able to introduce the logic formalization of the tests to be performed, in case that reasoning on the schema results to be decidable. The tool answers to each test either with a sample instantiation satisfying the test, if possible, or with a negative answer if the test is not satisfiable.

We will use the example discussed in chapter 4 in order to illustrate how the user can validate a structural schema using our implementation. The steps to be followed are:

1. Specify the schema using Poseidon (see Figure 25), and the OCL constraints in a text file (Figure 26)
2. The logic representation of the schema is obtained (Appendix A)
3. The dependency graph is shown, together with the results of the analysis of decidability (Figures 30 and 31)
4. Perform the validation tests desired

6.1 Architecture

We have implemented the prototype of our method within the *EinaGMC* project (UPC and UOC 2008), which is being developed by the Conceptual Modeling Group (GMC) from the Technical University of Catalonia (UPC) and the Open University of Catalonia

(UOC). The aim of this project is to build an advanced environment to work with conceptual schemas specified in UML and OCL.

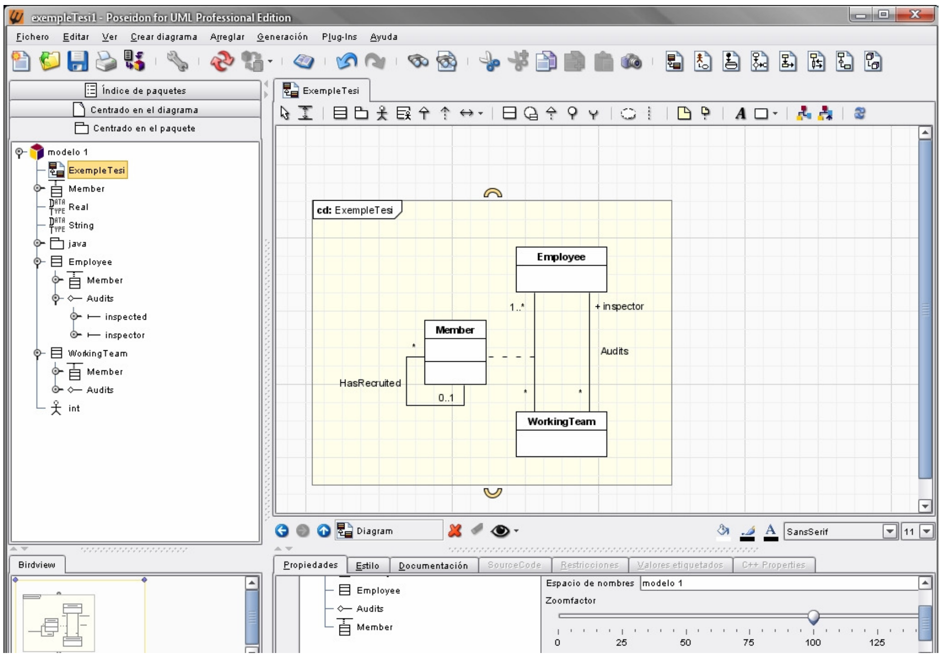
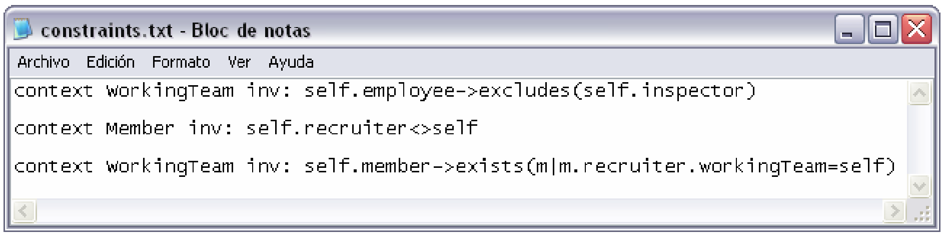


Fig. 25. The schema introduced in Poseidon


constraints.txt - Bloc de notas
Archivo Edición Formato Ver Ayuda

```
context WorkingTeam inv: self.employee->excludes(self.inspector)

context Member inv: self.recruiter<>self

context WorkingTeam inv: self.member->exists(m|m.recruiter.workingTeam=self)
```

Fig. 26. The OCL constraints corresponding to the schema in Figure 25

The core of *EinaGMC* is a Java library that implements the UML 2.0 (OMG 2007) and OCL 2.0 (OMG 2006) metamodels. Since a conceptual schema is an instantiation of these metamodels, the core of *EinaGMC* provides a set of Java objects and primitives that allow to manipulate the model (see Figure 27). In this context, by extending the *EinaGMC* core we can develop applications related to conceptual modeling in a Java framework.

The input UML and OCL schema of the *EinaGMC* is represented in XMI (XML Metadata Interchange). XMI is an OMG standard based on XML that intends to provide a standard way to exchange metadata by means of XML documents. In particular, it can be used to represent UML schemas.

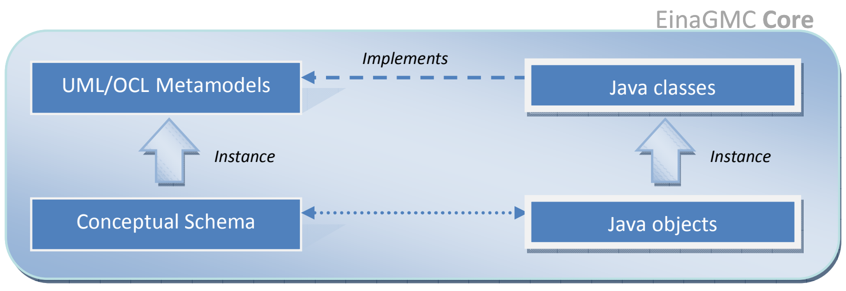
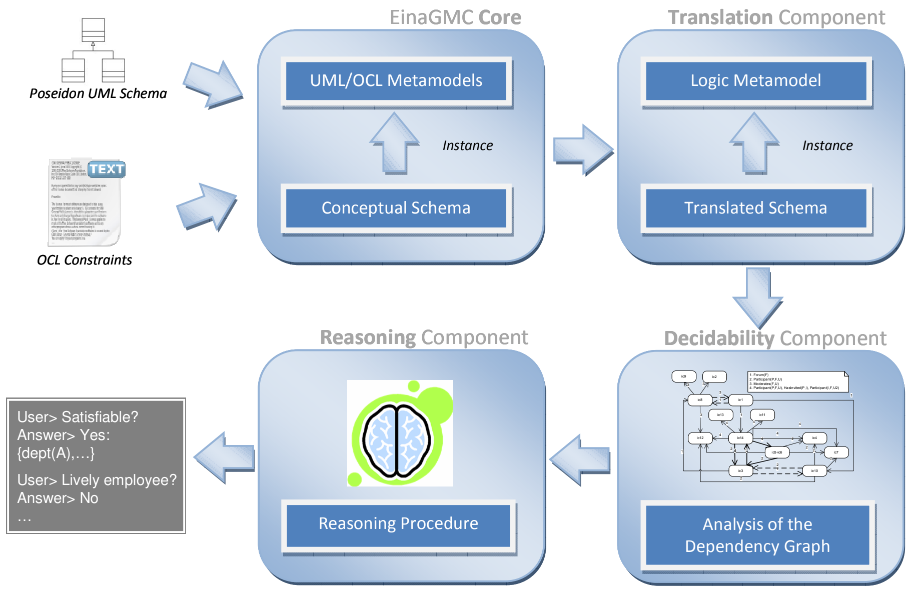


Fig. 27. *EinaGMC* Core

Since the XMI files generated by each different CASE tool are usually not compatible, the *EinaGMC* implements its own XMI. Thus, the files generated by the tools must be transformed to the *EinaGMC* XMI. Currently, a translator between Poseidon XMI and the XMI of *EinaGMC* is available, so the input models of our tool must be specified in Poseidon. However, there are some UML constructs that cannot be introduced in Poseidon, such as n-ary associations. Thus, in some cases, the *EinaGMC* XMI file has to be manually modified so that it represents the model to be validated.



```
User> Satisfiable?
Answer> Yes:
{dept(A),…}
User> Lively employee?
Answer> No
…
```

Fig. 28. General architecture of our tool

The general architecture of our tool is shown in Figure 28. As can be seen in the figure, the input of the tool consists in a UML schema specified in Poseidon, together with a text file containing the OCL specification of the constraints. This input is loaded as an instance of the classes in the *EinaGMC* core, which allows to manipulate them.

Our translation component is responsible for transforming the UML and OCL model into an instance of a logic metamodel, so that our method can be implemented using a logic representation of the schema. The decidability component is in charge of determining the decidability of reasoning on the particular schema to be validated and, finally, the reasoning component allows the user to perform the validation tests.

6.2 Translation Component

The translation component is aimed at expressing the UML and OCL schema, loaded as an instance of the *EinaGMC* core, as an instance of the logic metamodel in Figure 29.

This metamodel represents that a schema consists in a set of *NormalClauses* representing integrity constraints (*IC*) and derivation rules (*DR*), a set of *Atoms* and their definitions (*AtomDef*). The class *AtomDef* (which includes information about the predicate that it represents and its number of terms, among others) is used as a template to create atoms of a certain kind. Thus, there is an instance of *AtomDef* for each predicate of the schema.

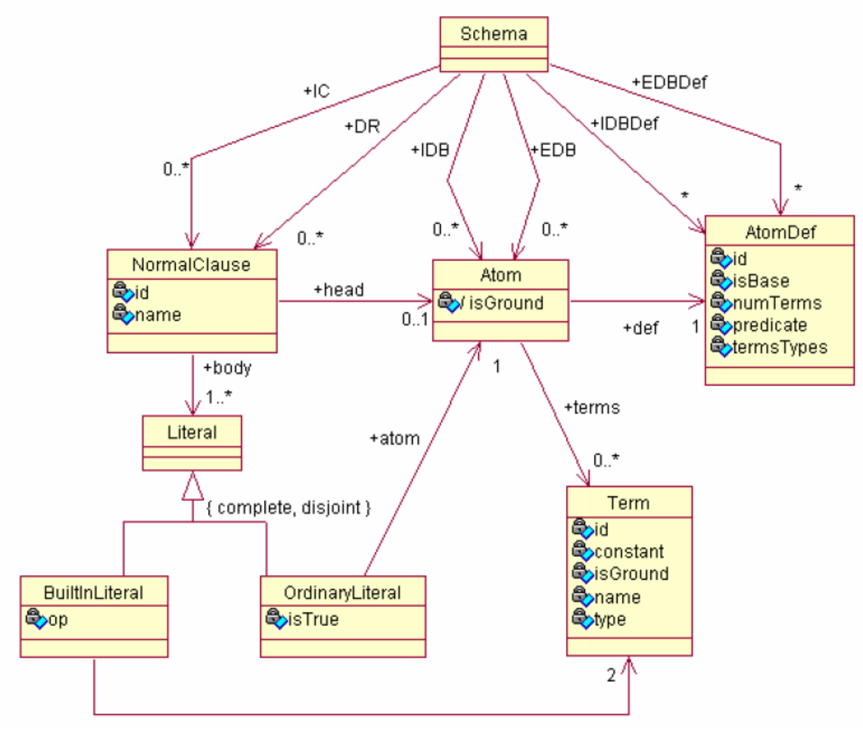


Fig. 29. Logic metamodel

A *NormalClause* may have an *Atom* as a head, and is composed by a set of *Literals*, which can be either *BuiltInLiterals* or *OrdinaryLiterals*. Built-in literals contain the comparison operator and the two terms compared, whereas ordinary literals correspond to an atom, which can be either negated or not (attribute *isTrue*).

Thus, once we have the schema loaded as an instance of this metamodel, we are able to manipulate it in order to implement the different steps of our method.

Additionally, this component returns a file containing the logic representation of the schema, which is included in the Appendix A.

6.3 Decidability Component

Once the schema is loaded as an instance of the logic metamodel, the decidability component is able to construct the dependency graph and analyze it in order to determine whether all the instances of the schema have a finite number of elements.

The dependency graph obtained from the constraints of the schema is shown in Figure 30. Superfluous arcs are omitted, and the cycles detected are grouped into nodes representing subgraphs. In this way, the reasoning component is able to order the repair of constraints so that each of them is treated the minimum number of times possible, as will be explained in the next section.

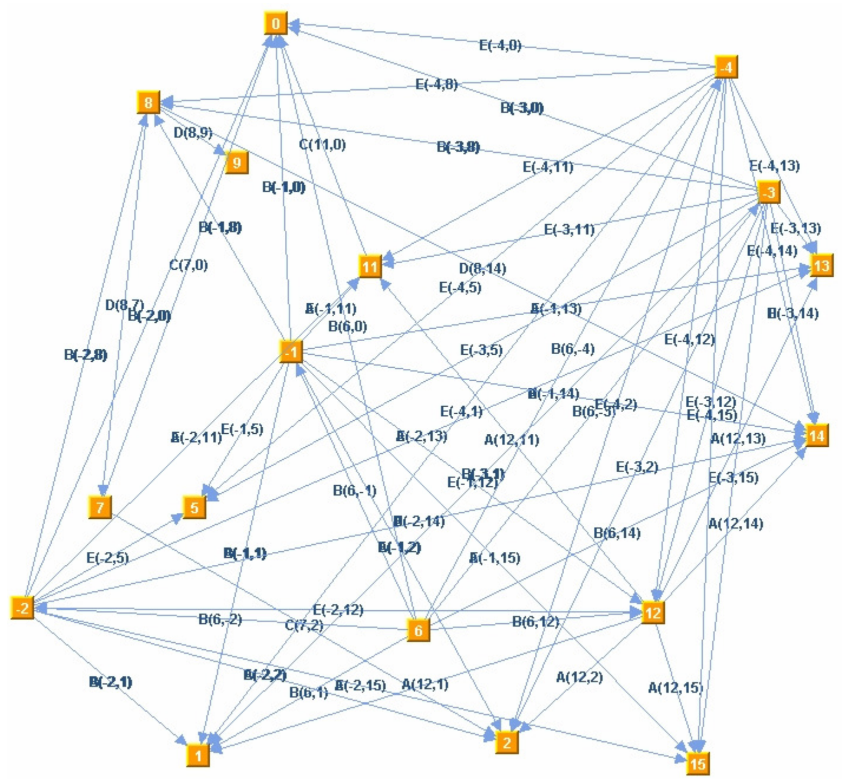


Fig. 30. Dependency graph obtained from the integrity constraints

As can be seen, this graph has more vertices and arcs than the one shown in section 4.1 (Figure 8). The reason is that in the automatic translation of the schema, constraints to ensure the uniqueness of OIDs are added, which where omitted in the explanation of chapter 4 for the sake of clarity. Additionally, the automatic translation of some OCL constraints introduces some literals that were also simplified in the example. This additional literals increase the number of potential violations of some constraints. For instance, the translation of the OCL constraint *InspectorNot Member* provided by our implementation is:

$\leftarrow$ workingTeam(T) $\land$ audits(E,T) $\land$ member(M,E,T)

instead of:

$\leftarrow$ audits(E,T) $\land$ member(M,E,T)

This simplification made in the explanation does not modify the results of the tests, since both logic representations are equivalent in our example (the term T of the predicates *audits* and *member* will always correspond to a *workingTeam* according to the referential constraints of the associations and, thus, *workingTeam(T)* is not needed in the logic formalization of the constraint).

This implies that the graph is more complicated, but the results of the cycle analysis are the ones expected. The results obtained as an output of this step can be seen in Figure 31.

As can be seen in the figure, four cycles are found. For each cycle, the repairs that cause the violation of the following constraint in the cycle are shown. For instance *WorkingTeam(10,16)* means that the repair *workingTeam(X)* of the constraint 10 obtained in the translation is a potential violation of constraint 16, which is the successor of 10 in the cycle.

```
Cycle 1: Finite
   WorkingTeam(10,16)
   WorkingTeam, Member, HasRecruited(16,3)
   Member(3,10)

Cycle 2: Finite
   WorkingTeam(10,16)
   WorkingTeam, Member, HasRecruited(16,10)

Cycle 3: Finite
   WorkingTeam, Member, HasRecruited(16,16)

Cycle 4: Finite
  WorkingTeam(10,16)
  WorkingTeam, Member, HasRecruited(16,4)
  Member(4,10)
```

Fig. 31. Results obtained from the analysis of the graph

Cycles 1, 2 and 4 correspond to the cycles explained in section 4.1. Cycle 2 corresponds to (*ic3 ic14*), while cycles 1 and 4 correspond to the cycles (*ic3 ic14 ic5*) and (*ic3 ic14 ic6*). The additional Cycle 3, which is also found finite, is due to the literals added in the automatic translation of the schema and corresponds to a recursive arc in the constraint *ic14*, since the predicate *WorkingTeam* is a potential violation that also belongs to its repair.

6.4 Reasoning Component

If the results obtained from the previous component determine that all the cycles found in the graph are finite, then the user is able to perform his tests with the guarantee that all of them will terminate. Currently, the tests must be introduced by the user according to the logic representation of the schema obtained from the translation component. However, our aim is to provide the user with a more intuitive interface, so that some tests are automatically executed without the need to specify them. Additionally, we also plan to provide the user with the ability of defining his own tests without need to be aware of the logic formalization of the schema.

The first step performed by the reasoning procedure is the initialization of the graph, which consists in marking all the nodes (and the subgraphs representing cycles) as not visited.

Then, the method starts with the Goal Satisfaction step, which consists in adding to the EDB under construction the positive literals of the goal, instantiated using one of the alternatives provided by the VIPs. After that, the Integrity Maintenance step must be performed as follows.

Using the order provided by the graph, the first constraint (node) to be considered is selected. The selection criteria is as follows:

1. the node is not visited
2. the node has no antecessor
3. the node has no antecessors that have not been visited yet, if the node does not belong to a cycle
4. if the current node belongs to a cycle, that is, if the node is a subgraph, the following node to be treated must also belong to the same subgraph, unless all the nodes belonging to the cycle have been already visited

When the node to be considered is selected, the treatment of the node consists in checking whether the corresponding constraint is violated by the current EDB:

- If it is not violated, the node is marked as visited, and the following node must be selected according to the selection criteria
- If it is violated, one of the possible repairs is chosen and applied, using one of the alternatives provided by the VIPs if the repair contains free variables. The successors of the node that are reached according to the repair applied are marked as not visited and, finally, the node is marked as visited and the following one is selected according to the criteria.

The procedure ends when all the nodes of the graph are marked as visited, which means that an EDB proving the test has been built. Otherwise, if all the alternatives provided by the VIPs and possible repairs have been taken, and some constraint cannot be successfully repaired the result is that the property tested cannot be satisfied.

In the following we show the results obtained when checking some of the tests that we have explained throughout de document. The constants used in the example constructed are always real numbers in order to facilitate the implementation. However, this does not affect the results of the tests.

We will try to check whether the associations *Audits* and *Member* require some kind of path inclusion or path exclusion constraint.

The first test corresponds to Test 7 to check path inclusion:

not-inclusionAuditsMember ← audits(E,T) ∧ ¬inMember(E,T)
inMember(E,T) ← member(M,E,T)

The result is:

```
Test:     audits(E,T), ¬inMember(E,T)
Answer:   Satisfiable
Facts:    Audits(1,0), WorkingTeam(0), Employee(1),
          Member(2,3,0), HasRecruited(2,4),
          Member(4,5,0), Employee(3), Employee(5)
Time:     1281 ms
```

Since an example has been found corresponding to the tested property, the meaning is that there is no path inclusion constraint in these associations.

If we now execute Test 8 to check path exclusion:

not-exclusionAuditsMember ← audits(E,T) ∧ member(M,E,T)

the result is:

```
Test:     audits(E,T), member(M,E,T)
Answer:   Unsatisfiable
Time:     67172 ms
```

That is, the test is unsatisfiable, which means that a constraint disallowing that the same instances of *Employee* and *WorkingTeam* are linked by both associations alredy exists.

Although our method allows that the tests are specified using constants instead of variables, note that in many cases this is not useful to check whether the schema satisfies or not a certain property. For instance, if we had executed the following test instead of the previous one:

```
Test:    audits(0,1), member(2,0,1)
Answer:  Unsatisfiable
Time:    3078 ms
```

the result obtained is the same (the property is unsatisfiable), but the tests do not have the same meaning. The former ensures that the same pair of instances of *Employee* and *WorkingTeam*, whichever they are, cannot belong to both associations, while the latter only ensures this for the pair *Employee(0)* and *WorkingTeam(1)*. This means that other instances, for example *Employee(2)* and *WorkingTeam(1),* may belong to both associations. As can be seen, the time spent in the execution of the second test is much lower, since the method does not have to guarantee that the property does not hold for any of the possible combinations of instances, but only for the given one.

7 Related Work

In this chapter we review the previous work on reasoning and validation of conceptual schemas. A brief explanation of each approach is included in section 1.5. However, here, we compare them in detail with our method, regarding the desirable properties that an approach of this kind should have.

We classify the approaches into those that deal with languages that are not UML and those that reason or validate schemas specified in UML. In both cases, we include a table with the following structure.

The column *Structural limitations* indicates those elements of the language under consideration, apart from integrity constraints, that are not supported by the approach.

Constraints handled specifies the kinds of constraints, which can be either graphical or textual, that are taken into account in the approach.

In *Kind of reasoning* we can see whether the approach is aimed at checking the internal or the external correctness of the schema, that is, whether it tests only structural properties of the schema or the designer is able to specify his own ad-hoc tests using his knowledge about the domain.

In the column *Definition of behavior* we indicate whether the behavioral schema is taken into account in some way.

The column *Reasoning with behavior* shows whether the definition of the behavior is taken into account, in the sense that the only changes that can be made on the IB are those specified in the behavioral schema.

The column *Complete* states whether the approach is such that when an example for a property exists, it will always be found.

Finally, in the column *Decidable* we can see if the approach guarantees termination in all the executions of the tests.

7.1 Non-UML Approaches

Table 4 includes those approaches dealing with validation and reasoning on schemas that are not specified in UML. We classify them in those that use ER and those that use other languages.

As can be seen in the table, the only approach that does not have any structural limitations is Alloy (MIT 2006), that uses a language similar to logic to specify the schemas. The rest of ER and non-ER approaches have some expressive limitation in order to guarantee completeness and decidability. The elements most commonly discarded are association classes, n-ary associations and subtyping, which are very important in UML schemas, especially when they are used for conceptual modeling, since schemas should be as expressive as possible. Thus, we find that the absence of these elements represents an important drawback for those approaches that do not consider them.

Another expressive feature are the *Constraints handled.* All the ER approaches, as well as (Baader, Lutz et al. 2005; Bekaert, Van Nuffelen et al. 2002; Formica 2003), are able to deal with the cardinality constraints. (MIT 2006) also considers them, since it can handle any kind of constraints that can be expressed in logic. Despite being so necessary in databases, we can see in the table that only the ER approach (Hartmann 2001) is able to deal with identifier (key) constraints, as well as (MIT 2006), since it deals with general constraints. (Baader, Lutz et al. 2005; Bekaert, Van Nuffelen et al. 2002) are the only approaches that deal with hierarchies and, thus, consider disjointness and covering constraints. Another kind of constraints are those that restrict the value of an attribute by means of some comparison operator, considered in (Formica 2003; Formica and Frank 2002).

Regarding the *Kind of reasoning* performed, all the ER approaches, as well as (Formica 2003; Formica and Frank 2002) only perform some specific tests to check internal correctness, the one most commonly addressed being satisfiability. (Baader, Lutz et al. 2005) also allows to test only internal correctness, this time regarding the correct definition of operations. In particular, the reasoning tasks that can be performed correspond to our definitions of applicability and executability of operations.

There are three methods that allow to test the external correctness (Bekaert, Van Nuffelen et al. 2002; Díaz, Paton et al. 1998; MIT 2006). (Bekaert, Van Nuffelen et al. 2002) is able to answer to the tests provided by the designer, but only on a semi-populated IB, since the method is not able to invent new values for the attributes. This means that, in some cases, it will not be able to find the example required and its answer will be that the tested property cannot be satisfied when, in fact, it may be with other values or by creating new instances. This is why this approach is not *Complete*.

Table 4. Comparison of the Non-UML approaches. The ones shadowed in gray allow the definition of behavior.

		Structural limitations	Constraints handled	Kind of reasoning	Definition of behavior	Reasoning with behavior	Complete	Decidable
ER	(Lenzerini and Nobili 1987)	Attributes, association classes, subtyping	Cardinalities	Internal	No	No	Yes	Yes
	(Hartmann 2001)	Association classes, subtyping	Cardinalities, Identifiers.	Internal	No	No	Yes	Yes
	(Vigna 2004)	Attributes, association classes, n-ary associations, subtyping	Cardinalities	Internal	No	No	Yes	Yes
Others	(Bekaert, Van Nuffelen et al. 2002)	N-ary associations	Cardinalities, Disjointness and covering	External	No	No	No	Yes
	(Formica 2003)	Association classes, n-ary associations, subtyping	Cardinalities, Restricted value comparisons	Internal	No	No	Yes	Yes
	(Díaz, Paton et al. 1998)	Subtyping	None	External	Yes, operations	Yes	Yes	Yes
	(Formica and Frank 2002)	Association classes, n-ary associations, subtyping	Restricted value comparisons	Internal	Yes, statecharts	No	Yes	Yes
	(Baader, Lutz et al. 2005)	Association classes, n-ary associations	Cardinalities, Disjointness and covering	Internal	Yes, operations	No	Yes	Yes
	(MIT 2006)	None	General	External	Yes, operations	No	No	Yes

Only four approaches take the behavior into account in some way. In (Formica and Frank 2002) the behavior is defined by means of statecharts, whereas (Baader, Lutz et al. 2005; Díaz, Paton et al. 1998) and (MIT 2006) specify the behavior by means of operation contracts. Only (Díaz, Paton et al. 1998) considers that the only changes allowed in a state of the IB are those specified by the operations and takes it into account when performing the validation, which means that the other two approaches may report as valid a state that is unreachable with the operations given.

Finally, as well as (Bekaert, Van Nuffelen et al. 2002) as we have seen, there is another approach that is not *Complete* (MIT 2006). This method is able to invent new instances but within a range of values specified by the user in order to guarantee termination. This means that, although a sample IB satisfying a certain property may exist, the method may not find it if it is beyond the bounds specified.

Thus, (MIT 2006) is the only approach that does not have any structural limitations and is able to deal with any kind of constraints, specified by means of a general logic expression. However, it is aimed at checking only the external correctness of the schema. Additionally, it has two important drawbacks. First, it does not take into account the meaning of operations in the validation process. Second, its lack of completeness represents a very important limitation, since when the method is not able to find an example of a certain property it cannot be guaranteed that one does not exist.

On the other hand, the only approach that considers the operations in the validation has important expressive limitations, since it cannot deal with subtyping and does not consider any kind of constraint. Moreover, it is aimed at checking only the external correctness by executing the models, so it cannot automatically detect semantic flaws such as satisfiability.

7.2 UML Approaches

In Table 5 we find those methods dealing with the reasoning and validation of UML conceptual schemas. We have classified them into two categories: the ones based in Description Logics (DL), and those using other approaches. The method proposed in this thesis belongs to this last group.

The basic idea of the DL approaches consists in translating a UML schema into a DL and then reason on this formalization, either by means of ad-hoc algorithms or using existing off-the-shelf reasoning tools. As can be seen from the table, none of the DL approaches deals with the behavioral part of the schema. Decidable DLs, which are the ones used in these approaches, are expressive enough to express any structural element of a UML schema (classes, attributes, class hierarchies, n-ary associations...) and also allow some kinds of constraints still guaranteeing termination.

The typical reasoning tasks performed in the DL approaches are those to check the internal correctness of the schema (satisfiability of the schema, liveliness of a class or association, class subsumption...). This can be done in finite time taking into account cardinality, disjointness and covering constraints, and without any structural limitation

(Berardi, Calvanese et al. 2005; Fillottrani, Franconi et al. 2006). Additionally (Fillottrani, Franconi et al. 2006) allows other kinds of constraints that must be expressed in a specific view definition language, not in OCL. The expressiveness is more limited in (Cadoli, Calvanese et al. 2007), since attributes, association classes and n-ary associations are not considered. The difference with the other DL approaches lies in the reasoning task performed, which consists in checking whether a class can be populated with a finite number of instances, which is a specific kind of satisfiability more interesting by its practical applicability.

Among the rest of UML approaches, the one that deals with more expressive schemas is (Cabot, Clarisó et al. 2008), since it is the only one that does not have any structural limitations and considers general OCL constraints. The rest of them, despite being able to deal with any kind of constraints, they do not consider association classes nor n-ary associations. From these ones, in (Brucker and Wolff 2006; Gogolla, Büttner et al. 2007) constraints are specified in OCL, whereas in (Dupuy, Ledru et al. 2000) they must be expressed in the Z language, and in (Snook and Butler 2006) constraints must be specified in B. We consider that this is a drawback, since OCL is the language recommended to formalize the constraints in UML schemas.

Regarding the behavioral schema, in (Brucker and Wolff 2006; Cabot, Clarisó et al. 2009; Dupuy, Ledru et al. 2000; Gogolla, Büttner et al. 2007), operations are specified by means of declarative contracts, with their preconditions and postconditions specified in OCL, while in (Snook and Butler 2006) operations are procedural and specified in the B language. All these approaches are able to test that the operations defined are applicable and executable, but none of them considers that the changes specified by the operations are the only ones allowed when checking the properties satisfied by the schema (satisfiability, liveliness...). Thus, a property can be reported as valid when, in fact, it is impossible to satisfy using the operations defined. We have seen several such examples in chapter 5. This also damages the results obtained when testing the applicability of operations, since the state that satisfies a precondition may not be obtained by means of the operations of the behavioral schema and, thus, the result of the test is not reliable. Additionally, operations are specified according to an extended semantics which, as discussed in section 5.1, has several drawbacks regarding modifiability, consistency and conciseness, derived of the required redundancy between preconditions and integrity constraints.

Completeness is not guaranteed by (Cabot, Clarisó et al. 2008; Gogolla, Büttner et al. 2007; Leuschel and Butler 2008), since they restrict the domains of the variables in order to guarantee decidability and they may fail to find an existing solution. On the contrary, (Brucker and Wolff 2006; Dupuy, Ledru et al. 2000) are complete but, in change, they do not guarantee termination in all cases.

Table 5. Comparison of the UML approaches. The ones shadowed in gray allow the definition of behavior.

		Structural limitations	Constraints handled	Kind of reasoning	Definition of behavior	Reasoning with behavior	Complete	Decidable
DL-UML	(Berardi, Calvanese et al. 2005)	None	Cardinalities, Disjointness and covering	Internal	No	No	Yes	Yes
	(Fillottrani, Franconi et al. 2006)	None	Cardinalities, Disjointness and covering, Identifiers, Inclusion and exclusion dep.	Internal	No	No	Yes	Yes
	(Cadoli, Calvanese et al. 2007)	Attributes, association classes, n-ary associations	Cardinalities, Disjointness and covering	Internal	No	No	Yes	Yes
UML	(Dupuy, Ledru et al. 2000)	N-ary associations	General Z constraints	Internal	Yes, operations	No	Yes	No
	(Brucker and Wolff 2006)	Association classes, n-ary associations	General	External	Yes, operations	No	Yes	No
	(Gogolla, Büttner et al. 2007)	Association classes, n-ary associations	General	External	Yes, operations	No	No	Yes
	(Snook and Butler 2006), (Leuschel and Butler 2008)	Association classes, n-ary associations	General B constraints	Internal	Yes, B operations	No	No	Yes
	(Cabot, Clarisó et al. 2008; 2009)	None	General	Internal	Yes, operations	No	No	Yes

Summarizing, and as happens with the non-UML approaches, the UML ones have three main drawbacks. The first one is that, in order to guarantee decidability, some of them renounce either to expressiveness by disallowing certain constructs or constraints, or to completeness by requiring finite domains. On the other hand, the methods that are not decidable, also disallow some UML constructs, such as n-ary associations or association classes.

The second drawback is that all of them are able to perform only one kind of reasoning, either to check the internal or the external correctness of the schema, but not both.

Finally, the third drawback, also shared by all the approaches dealing with behavior in some sense, is that the semantics of operations is not taken into account in the validation, thus reporting as valid states that are, in fact, impossible to construct.

As a conclusion, we can say that our method overcomes the limitations we find in the previous approaches. In this way, our work represents a significant contribution to the field of validation and reasoning on UML and OCL schemas, with and without operations.

In particular, the expressiveness of the schemas that can be handled by our method is very high, which is one of our main goals. We can deal with any structural element of UML class diagrams that is useful in conceptual modeling, and also with general OCL constraints. However, we are not able to deal with those OCL expressions that include arithmetic operations, since they cannot be expressed in the logic representation we use to reason. This is why we have marked this cell with an asterisk.

As far as the kind of reasoning is concerned, our approach is the only one that, using a single method, is able to check any internal property (satisfiability, liveliness, non-redundancy) and also any ad-hoc test that the designer may wish in order to check the correspondence of the schema with the requirements.

Additionally, among the UML approaches dealing with operations, ours is the only one that takes into account the concept of operation by only allowing the changes specified in them in order to construct the sample IBs.

Our approach is also complete, which we find essential, and besides, we can guarantee decidability in many cases, as we have seen in section 4.1. The fact that our method is not decidable in all cases is the reason for the asterisk in this cell. However, it is worth to note that we identify the cases in which the reasoning may not terminate, so the designer is aware of this fact before performing the tests on each particular schema.

8 Conclusions

To ensure the quality of an information system, it is essential that the conceptual schema that represents the knowledge about its domain and the functions it has to perform is semantically correct.

The correctness of a conceptual schema can be seen from two different perspectives. On the one hand, from the point of view of its definition, determining the internal correctness of a conceptual schema consists in answering to the question "*Is the conceptual schema right?*". This can be achieved by determining whether the schema fulfills certain properties, such as satisfiability, non-redundancy or operation executability.

On the other hand, from the perspective of the requirements that the information system should satisfy, not only the conceptual schema *must be right*, but it also *must be the right one*. In order to determine this external correctness, the designer must be provided with some kind of help and guidance during the validation process, so that he is able to understand the exact semantics of the schema and see whether it corresponds to the requirements to be formalized.

Both kinds of correctness have been addressed in the literature. However, current methods and tools can be improved in several ways.

From the point of view of the expressiveness they can deal with, the most expressive approaches do not guarantee termination of the reasoning procedure. In contrast, those that guarantee termination, only allow some predefined constructs in the conceptual schema, or fail to detect certain erroneous situations.

Regarding the validation of conceptual schemas with a behavioral part, the fact that only the changes defined in the operations can be performed is not taken into account by most approaches.

In this thesis we have presented an approach to help the designer in the validation of a complete UML conceptual schema, with general constraints and operations formalized in OCL. Our approach allows to validate the conceptual schema both from the point of view of its definition and of its correspondence to the requirements.

To guarantee the internal correctness of a schema, a set of tests is provided. Some of the tests correspond to well-known properties whereas others, which are an original contribution of this thesis, are automatically generated from the concrete schema to be validated.

To check the external correctness, some additional tests are automatically provided to the designer. Moreover, the designer can define his own tests in order to complete the validation process.

All the validation tests are formalized in such a way that they can be treated uniformly, regardless the specific property they allow to test.

Our approach can be either applied to a complete conceptual schema or only to its structural part. In case that only the structural part is validated, we provide a set of conditions to determine whether any validation test performed on the schema will terminate. For those cases in which these conditions of termination are satisfied, we also provide a reasoning procedure that takes advantage of this situation and works more efficiently than in the general case. This approach allows the validation of very expressive schemas and ensures completeness and decidability at the same time.

When the schema contains operations, or when the conditions of termination are not guaranteed, any reasoning procedure that is able to deal with negation of derived predicates can be used to perform the validation tests we have formalized. The formalization of the complete schema ensures that a test is satisfied only if the tested property is reachable using the operations defined.

In order to show the feasibility of our approach, we have built a prototype that implements the complete validation process for a structural schema. Additionally, for the validation of a conceptual schema with a behavioral part, we have extended the implementation of an existing reasoning procedure so that it can correctly deal with operations.

As a conclusion, we have provided an approach which improves the results of previous proposals to validate both the internal and the external correctness of a conceptual schema, with or without operations. The validation is performed by means of a set of tests that are applied to the schema, including automatically generated tests and ad-hoc tests defined by the user. The expressiveness allowed for the schema to be validated is high, and our reasoning procedure is complete for a conceptual schema with operations, and also decidable for the structural part.

As further work to be started from this thesis, the efficiency of our reasoning procedure can be addressed. Since our procedure is efficient only for those schemas for which the reasoning is found decidable, a direct research direction to be followed is to improve the efficiency of our method for the rest of schemas. In particular, this includes the schemas containing operations. Additionally, the efficiency in the validation of the structural schema can also be increased to some extent, for instance by considering the conditions specified by those constraints that cannot be repaired when trying to find a solution.

Also, we believe that we can increase the number of conditions that allow determining the decidability of reasoning on a schema. In this way, a larger number of schemas can be benefited from our more efficient algorithm that can be used in these cases.

We would also like to extend our results to increase the expressiveness of the conceptual schemas to be validated. On the one hand, we could minimize the restrictions on the OCL expressions we can deal with. On the other hand, we plan to extend our approach so that it can deal with derived information defined in the conceptual schema.

Another interesting direction to follow is to extend our reasoning procedure so that it can provide explanations of the problems found in the schema. To help the designer to fix an error detected in the schema, it is useful to provide him with the cause of that error, which consists in the set of constraints that are responsible for the non-satisfaction of the property being tested.

Finally, since our algorithm constructs sample instantiations to prove the desired properties, it could also be extended in order to generate test data to be used when validating the final implementation of the specification.

Regarding technological issues, the implementation of the translation of a complete conceptual schema with operations remains to be done. Also, we would like to develop a tool that integrates the complete validation process of a conceptual schema, with and without operations, with an interface that proactively provides interesting information to the designer and facilitates the definition of the user-defined tests.

References

Adrion, W. R., M. A. Branstad, et al. (1982). "Validation, Verification and Testing of Computer Software." ACM Comput. Surv. **14**(2): 159-192.

Baader, F., D. Calvanese, et al. (2003). The Description Logic Handbook: Theory, Implementation and Applications, Cambridge University Press.

Baader, F., C. Lutz, et al. (2005). Integrating Description Logics and Action Formalisms: First Results. AAAI 2005, MIT Press**:** 572-577.

Bekaert, P., B. Van Nuffelen, et al. (2002). On the Transformation of Object-Oriented Conceptual Models to Logical Theories. Conceptual Modeling - ER 2002, LNCS 2503**:** 152-166.

Berardi, D., D. Calvanese, et al. (2005). "Reasoning on UML Class Diagrams." Artificial Intelligence **168**(1-2): 70-118.

Boman, M., J. A. Bubenko, et al. (1997). Conceptual Modelling. Upper Saddle River, NJ, USA, Prentice-Hall, Inc.

Borgida, A. (1995). "Description Logics in Data Management." IEEE Transactions on Knowledge and Data Engineering **7**(5): 671-682.

Borgida, A. and R. J. Brachman (2003). Conceptual Modeling with Description Logics. The Description Logic Handbook: Theory, Implementation and Applications. F. Baader, D. Calvanese, D. McGuiness, D. Nardi and P. Patel-Schneider, Cambridge University Press**:** 359-381.

Borgida, A., M. Lenzerini, et al. (2003). Description Logics for Data Bases. The Description Logic Handbook: Theory, Implementation and Applications. F. Baader, D. Calvanese, D. McGuiness, D. Nardi and P. Patel-Schneider, Cambridge University Press**:** 472-494.

Borgida, A., J. Mylopoulos, et al. (1995). "On the frame problem in procedure specifications." IEEE Transactions on Software Engineering **21**(10): 785-798.

Bowers, D. S. (2003). "Detection of Redundant Arcs in Entity Relationship Conceptual Models." ER 2003 Ws **LNCS 2784**: 275-287.

Brucker, A. D. and B. Wolff (2006). The HOL-OCL Book, Swiss Federal Institute of Technology (ETH).

Bubenko, J. A. (1986). Information System Methodologies - A Research View. Proc. of the IFIP WG 8.1 Working Conference on Information Systems Design Methodologies: Improving the Practice, North-Holland Publishing Co.

Cabot, J., R. Clarisó, et al. (2008). Verification of UML/OCL Class Diagrams Using Constraint Programming. Workshop on Model Driven Engineering, Verification and Validation (MoDEVVa 2008).

Cabot, J., R. Clarisó, et al. (2009). Verifying UML/OCL Operation Contracts. Integrated Formal Methods.

Cadoli, M., D. Calvanese, et al. (2007). Finite Model Reasoning on UML Class Diagrams via Constraint Programming. AI*IA 2007: Artificial Intelligence and Human-Oriented Computing**:** 36-47.

Calvanese, D., M. Lenzerini, et al. (1998). Description Logics for Conceptual Data Modeling. Logics for Databases and Information Systems. J. Chomicki and G. Saake, Kluwer**:** 229-263.

Ceri, S., P. Fraternali, et al. (1994). "Automatic Generation of Production Rules for Integrity Maintenance." ACM Transactions on Database Systems **19**(3): 367-422.

Console, L., M. L. Sapino, et al. (1995). "The Role of Abduction in Database View Updating." J. Intelligent Information Systems **4**(3): 261-280.

Costal, D., C. Gómez, et al. (2008). "Improving the Definition of General Constraints in UML." Software and System Modeling **7**(4): 469-486.

Costal, D., M. R. Sancho, et al. (2002). Understanding Redundancy in UML Models for Object-Oriented Analysis. Advanced Information Systems Engineering: 14th International Conference, CAiSE 2002 Proceedings, LNCS 2348**:** 659-674.

Costal, D., E. Teniente, et al. (1996). "Handling Conceptual Model Validation by Planning." 8th Conference on Advanced Information Systems Engineering - CAiSE'96 **LNCS 1080**: 255-271.

D'Souza, D. F. and A. C. Wills (1998). Objects, Components and Frameworks with UML: The Catalysis Approach, Addison-Wesley.

Davis, A. M. (1993). Software Requirements: Objects, Functions and States. Englewood Cliffs, Prentice Hall.

Decker, H., E. Teniente, et al. (1996). "How to Tackle Schema Validation by View Updating." Proc. of the 5th Conference on Extending DataBase Technology - EDBT'96 **LNCS 1057**: 535-549.

Devos, F. and E. Steegmans (2005). "Specifying Business Rules in Object-Oriented Analysis." Software and Systems Modeling **4**(3): 297-309.

Díaz, O., N. W. Paton, et al. (1998). "Formalizing and Validating Behavioral Models through the Event Calculus." Information Systems **23**(3/4): 179-196.

Dupuy, S., Y. Ledru, et al. (2000). An Overview of RoZ: A Tool for Integrating UML and Z Specifications. Conference on Advanced Information Systems Engineering - CAiSE 2000, LNCS 1789**:** 417-430.

Farré, C., E. Teniente, et al. (2005). "Checking Query Containment with the CQC Method." Data and Knowledge Engineering **53**(2): 163-223.

Fillottrani, P. R., E. Franconi, et al. (2006). The New ICOM Ontology Editor. International Workshop on Description Logics (DL2006).

Formica, A. (2002). "Finite Satisfiability of Integrity Constraints in Object-Oriented Database Schemas." IEEE Transactions on Knowledge and Data Engineering **14**(1): 123-139.

Formica, A. (2003). "Satisfiability of Object-Oriented Database Constraints with Set and Bag Attributes." Information Systems **28**: 213-224.

Formica, A. and H. Frank (2002). "Consistency of the Static and Dynamic Components of Object-Oriented Specifications." Data and Knowledge Engineering **40**: 195-215.

Gogolla, M., J. Bohling, et al. (2005). "Validating UML and OCL Models in USE by Automatic Snapshot Generation." Software and System Modeling **4**(4): 386-398.

Gogolla, M., F. Büttner, et al. (2007). "USE: A UML-based Specification Environment for Validating UML and OCL." Science of Computer Programming **69**(1-3): 27-34.

Gogolla, M. and M. Richters (2002). Expressing UML Class Diagrams Properties with OCL. Object Modeling with the OCL. T. Clark and J. Warmer, LNCS 2263**:** 85-114.

Hartmann, S. (1998). "On the Consistency of Int-cardinality Constraints." 17th International Conference on Conceptual Modeling - ER'98 **LNCS 1507**: 150-163.

Hartmann, S. (2001). "Coping with Inconsistent Constraint Specifications." 20th International Conference on Conceptual Modeling - ER 2001 **LNCS 2224**: 241-255.

Hoare, C. A. R. (1972). "Proof of Correctness of Data Representations." Acta Informatica **1**(4): 271-281.

ISO/TC97/SC5/WG3 (1982). Concepts and Terminology for the Conceptual Schema and the Information Base. J. J. van Griethuysen.

Larman, C. (2004). Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, Prentice Hall PTR.

Lenzerini, M. (1999). "Description Logics and their Relationships with Databases." Proc. of the 7th International Conference on Database Theory - ICDT'99 **LNCS 1540**: 32-38.

Lenzerini, M. and P. Nobili (1987). On the Satisfiability of Dependency Constraints in Entity-Relationship Schemata. 13th International Conference on Very Large Databases - VLDB'87, Morgan Kaufmann.

Leuschel, M. and M. Butler (2008). "ProB: An Automated Analysis Toolset for the B Method." Software Tools for Technology Transfer **DOI: s10009-007-0063-9**.

Lobo, J. and G. Trajcevski (1997). "Minimal and Consistent Evolution in Knowledge Bases." J. Applied Non-Classical Logics **7**(1-2): 117-146.

Maraee, A. and M. Balaban (2007). Efficient Reasoning about Finite Satisfiability of UML Class Diagrams with Constrained Generalization Sets. European Conference on Model Driven Architecture - Foundations and Applications - ECMDA-FA 2007, LNCS 4530**:** 17-31.

Martin, J. and J. J. Odell (1999). Object-Oriented Methods. A Foundation. Englewood Cliffs, New Jersey, P T R Prentice Hall.

Mayol, E. and E. Teniente (2003). "Consistency Preserving Updates in Deductive Databases." Data and Knowledge Engineering **47**(1): 61-103.

Meyer, B. (1992). "Applying 'Design by Contract'." Computer **25**(10): 40-51.

Meyer, B. (1997). Object-Oriented Software Construction. New York, Prentice Hall.

MIT. (2006). "The Alloy Analyzer." from http://alloy.mit.edu.

Moerkotte, G. and P. C. Lockemann (1991). "Reactive Consistency Control in Deductive Databases." ACM Transactions on Database Systems **16**(4): 670-702.

Olivé, A. (2004). Definition of Events and their Effects in Object-Oriented Conceptual Modeling Languages. Conceptual Modeling - ER 2004, LNCS 3288.

Olivé, A. (2005). "Conceptual Schema-Centric Development: A Grand Challenge for Information Systems Research." 17th Int. Conf. on Advanced Information Systems Engineering (CAISE'05) **LNCS 3520**: 1-15.

Olivé, A. (2007). Conceptual Modeling of Information Systems, Springer.

Olivé, A. and M. R. Sancho (1996). "Validating Conceptual Specifications through Model Execution." Information Systems **21**(2): 167-186.

Oliver, I. and S. Kent (1999). Validation of Object Oriented Models using Animation. EUROMICRO Conference, IEEE Computer Society.

OMG (2006). Object Constraint Language Specification, Version 2.0. http://www.omg.org/technology/documents/formal/ocl.htm.

OMG (2007). UML 2.1 Superstructure Specification. http://www.omg.org/spec/UML/2.1.2/.

Pressman, R. S. (2004). Software Engineering: A Practitioner's Approach, McGraw-Hill.

Queralt, A. and E. Teniente (2006a). Reasoning on UML Class Diagrams with OCL Constraints. Conceptual Modeling - ER 2006, LNCS 4215**:** 497-512.

Queralt, A. and E. Teniente (2006b). "Specifying the Semantics of Operation Contracts in Conceptual Modeling." Journal on Data Semantics **JoDS VII**: 33-56.

Queralt, A. and E. Teniente (2008a). Decidable Reasoning in UML Schemas with Constraints. Advanced Information Systems Engineering- CAiSE'08, LNCS 5074**:** 281-295.

Queralt, A. and E. Teniente (2008b). Validation of UML Conceptual Schemas with Operations. CAiSE'08 Forum.

Rumbaugh, J., I. Jacobson, et al. (2004). The Unified Modeling Language Reference Manual, Addison Wesley Professional.

Schewe, K. D. and B. Thalheim (1999). "Towards a Theory of Consistency Enforcement." Acta Informatica **36**(2): 97-141.

Snook, C. and M. Butler (2006). "UML-B: Formal Modeling and Design Aided by UML " ACM Trans. on Soft. Engineering and Methodology **15**(1): 92-122.

Thalheim, B. (2000). Entity-Relationship Modeling. Foundations of Database Technology, Springer.

UPC and UOC. (2008). "EinaGMC." from http://guifre.lsi.upc.edu/eina_GMC.

Utting, M. and B. Legeard (2006). Practical Model-Based Testing, Morgan Kauffman.

Various Authors (1998). IEEE Recommended Practice for Software Requirements Specifications (IEEE Std 830-1998).

Vigna, S. (2004). Reachability Problems in Entity-Relationship Schema Instances. Conceptual Modeling - ER 2004, LNCS 3288**:** 96-109.

Warmer, J. and A. Kleppe (2003). The Object Constraint Language: Getting Your Models Ready for MDA, Addison-Wesley Professional.

Wieringa, R. (1998). "A Survey of Structured and Object-Oriented Software Specification Methods and Techniques." ACM Comput. Surv. **30**(4): 459-527.

Appendix A

We provide here the whole translation of the example in Figures 1 and 2, according to the method explained in section 3.1.

Translation of classes and associations

department(D)

departmentName(D,Name)

employee(E)

employeeName(E,Name)

boss(B)

bossPhone(B,Phone)

worksIn(E,D)

manages(E,D)

worksFor(E1,E2)

workingTeam(T)

audits(E,T)

member(M,E,T)

hasRecruited(R,M)

Constraints for OIDs

←employee(X) ∧ department(X)

←employee(X) ∧ workingTeam(X)

←employee(X) ∧ member(X,E,T)

←department(X) ∧ workingTeam(X)

←department(X) ∧ member(X,E,T)

←workingTeam(X) ∧ member(X,E,T)

Constraints for hierarchies

←boss(E) ∧ ¬employee(E)

Referential constraints for associations

←worksIn(E,D) ∧ ¬employee(E)

←worksIn(E,D) ∧ ¬department(D)

←manages(E,D) ∧ ¬employee(E)

←manages(E,D) ∧ ¬department(D)

←worksFor(S,E) ∧ ¬employee(S)

←worksFor(S,E) ∧ ¬employee(E)

←audits(E,T) ∧ ¬employee(E)

←audits(E,T) ∧ ¬workingTeam(T)

←member(M,E,T) ∧ ¬employee(E)

←member(M,E,T) ∧ ¬workingTeam(T)

←hasRecruited(R,M) ∧ ¬isMember(R)

←hasRecruited(R,M) ∧ ¬isMember(M)

isMember(M) ← member(M,E,T)

Constraints for uniqueness of association classes

←member(M,E,T) ∧ member(M2,E,T) ∧ M<>M2

Constraints for association cardinalities

←department(D) ∧ ¬oneEmployee(D)

oneEmployee(D) ← worksIn(E,D)

←employee(E) ∧ ¬oneDepartment(E)

oneDepartment(E) ← worksIn(E,D)

←worksIn(E,D1) ∧ worksIn(E,D2) ∧ D1<>D2

←department(D,N) ∧ ¬oneManager(D)

oneManager(D) ← manages(E,D)

←manages(E1,D) ∧ manages(E2,D) ∧ E1<>E2

←manages(E,D1) ∧ manages(E,D2) ∧ D1<>D2

←worksFor(S1,E) ∧ worksFor(S2,E) ∧ S1<>S2

←workingTeam(T) ∧ ¬oneInspector(T)

oneInspector(T) ← audits(E,T)

←audits(E,T) ∧ audits(E2,T) ∧ E<>E2

←workingTeam(T) ∧ ¬oneMember(T)

oneMember(T) ← member(M,E,T)

←hasRecruited(R,E) ∧ hasRecruited(R2,E) ∧ R<>R2

Referential constraints for attributes

$\leftarrow$departmentName(D,N) $\wedge$ $\neg$department(D)

$\leftarrow$departmentMinSal(D,M) $\wedge$ $\neg$department(D)

$\leftarrow$departmentMaxSal(D,M) $\wedge$ $\neg$department(D)

$\leftarrow$employeeName(E,N) $\wedge$ $\neg$employee(E)

$\leftarrow$employeeSalary(E,S) $\wedge$ $\neg$employee(E)

$\leftarrow$bossPhone(B,P) $\wedge$ $\neg$boss(B)

$\leftarrow$workingTeamName(T,N) $\wedge$ $\neg$workingTeam(T)

Constraints for attribute cardinalities

$\leftarrow$department(D) $\wedge$ $\neg$oneDepartmentName(D)

oneDepartmentName(D) $\leftarrow$ departmentName(D,N)

$\leftarrow$departmentName(D,N1) $\wedge$ departmentName(D,N2) $\wedge$ N1<>N2

$\leftarrow$department(D) $\wedge$ $\neg$oneDepartmentMinSal(D)

oneDepartmentMinSal(D) $\leftarrow$ departmentMinSal(D,M)

$\leftarrow$departmentMinSal(D,M1) $\wedge$ departmentMinSal(D,M2) $\wedge$ M1<>M2

$\leftarrow$department(D) $\wedge$ $\neg$oneDepartmentMaxSal(D)

oneDepartmentMaxSal(D) $\leftarrow$ departmentMaxSal(D,M)

$\leftarrow$departmentMaxSal(D,M1) $\wedge$ departmentMaxSal(D,M2) $\wedge$ M1<>M2

$\leftarrow$employee(E) $\wedge$ $\neg$oneEmployeeName(E)

oneEmployeeName(E) $\leftarrow$ employeeName(E,N)

$\leftarrow$employeeName(E,N1) $\wedge$ employeeName(E,N2) $\wedge$ N1<>N2

$\leftarrow$employee(E) $\wedge$ $\neg$oneEmployeeSalary(E)

oneEmployeeSalary(E) $\leftarrow$ employeeSalary(E,S)

$\leftarrow$employeeSalary(E,S1) $\wedge$ employeeSalary(E,S2) $\wedge$ S1<>S2

$\leftarrow$boss(B) $\wedge$ $\neg$onebossPhone(B)

onebossPhone(B) $\leftarrow$ bossPhone(B,p)

$\leftarrow$bossPhone(B,P1) $\wedge$ bossPhone(B,P2) $\wedge$ P1<>P2

$\leftarrow$workingTeam(T) $\wedge$ $\neg$oneWorkingTeamName(T)

oneWorkingTeamName(T) $\leftarrow$ workingTeamName(T,N)

$\leftarrow$employeeName(E,N1) $\wedge$ employeeName(E,N2) $\wedge$ N1<>N2

OCL constraints

context Department **inv** UniqueDep**:**

Department.allInstances()->select(d | Department.allInstances()->select(d2 | d<>d2 and d2.name=d.name))->size()=0

$\leftarrow$department(D) $\wedge$ departmentName(D,N) $\wedge$ department(D2) $\wedge$
departmentName(D2,N2) $\wedge$ D<>D2 $\wedge$ N2=N

context Employee **inv** UniqueEmp**:**

Employee.allInstances()->select(e | Employee.allInstances()->select(e2 | e<>e2 and e2.name=e.name))->size()=0

$\leftarrow$employee(E) $\wedge$ employeeName(E,N) $\wedge$ employee(E2) $\wedge$ employeeName(E2,N2) $\wedge$
E<>E2 $\wedge$ N2=N

context WorkingTeam **inv** UniqueTeam**:**

WorkingTeam.allInstances()->select(t | WorkingTeam.allInstances()->select(t2 | t<>t2 and t2.name=t.name))->size()=0

$\leftarrow$workingTeam(T) $\wedge$ workingTeamName(T,N) $\wedge$ workingTeam(T2) $\wedge$
workingTeamName(T2,N2) $\wedge$ T<>T2 $\wedge$ N2=N

context Department **inv** MinimumSalary :

self.minSalary > 1000

$\leftarrow$department(D) $\wedge$ departmentMinSal(D,M) $\wedge$ M<=1000

context Department **inv** CorrectSalaries:

self.minSalary < self.maxSalary

$\leftarrow$department(D) $\wedge$ departmentMinSal(D,Min) $\wedge$ departmentMaxSal(D,Max) $\wedge$
Min>=Max

context Department **inv** ManagerIsWorker:

self.worker->select(e | e=self.manager)->size()>0

$\leftarrow$ department(D) $\wedge$ $\neg$aux1(D)

aux1(D) $\leftarrow$ department(D) $\wedge$ worksIn(E,D) $\wedge$ manages(E2,D) $\wedge$ E=E2

context Department **inv** ManagerHasNoSuperior:
self.manager.superior->size()=0

$\leftarrow$department(D) $\wedge$ manages(E,D) $\wedge$ worksFor(S,E)

context Boss **inv** BossIsManager:
self.managed-dep->size()>0

$\leftarrow$boss(B) $\wedge$ $\neg$aux2(B)
aux2(B) $\leftarrow$ boss(B) $\wedge$ manages(B,D)

context Boss **inv** BossHasNoSuperior:
self.superior->size()=0

$\leftarrow$ boss(B) $\wedge$ worksFor(S,B)

context Boss **inv** SuperiorOfAllWorkers:
self.managed-dep.worker->select(E | self.employee->select(E2 | E2=E)->size()=0)
    ->size()=0

$\leftarrow$boss(B) $\wedge$ manages(B, D) $\wedge$ worksIn(E, D) $\wedge$ $\neg$aux3(E,B)
aux3(E,B) $\leftarrow$ boss(B) $\wedge$ worksFor(B, E2) $\wedge$ E2=E

context WorkingTeam **inv** InspectorNotMember:
self.employee->select(e | e=self.inspector)->size()=0

$\leftarrow$workingTeam(T) $\wedge$ member(M,E,T) $\wedge$ audits(I,T) $\wedge$ E=I

context Member **inv** NotSelfRecruited:
self.recruiter<>self

$\leftarrow$member(M,E,T) $\wedge$ hasRecruited(R,M) $\wedge$ R=M

context WorkingTeam **inv** OneRecruited:
self.member->select(m | m.recruiter.workingTeam=self)->size()>0

$\leftarrow$workingTeam(T) $\wedge$ $\neg$aux4(T)
aux4(T) $\leftarrow$ workingTeam(T) $\wedge$ member(M,E,T) $\wedge$ hasRecruited(R,M) $\wedge$ member(R,E2,T)

Appendix B

Translation of the behavioral part of the schema, as explained in section 5.2, according to the operation contracts specified in Figure 3. We do not include here the translation of the strcutrual schema, which is similar to the one included in the Appendix A, but taking into account now that the predicates representing classes also include terms representing their attributes.

Base predicates

newDept(D,Name,Min,Max,E,MgrN,MgrS,T)

removeDept(D,T)

hire(E,Name,Sal,Dept,T)

fire(E,T)

newTeam(W,Name,Insp,T)

newMember(M,Emp,WorkT,Rec,T)

time(T)

Deriving instances from operations

Employee

employee(E,Name,Sal,T) ← addEmployee(E,Name,Sal,T2) ∧ ¬deletedEmployee(E,T2,T) ∧ $T2 \leq T$ ∧ time(T)

deletedEmployee(E,T1,T2) ← delEmployee(E,T) ∧ $T>T1$ ∧ $T \leq T2$ ∧ time(T1) ∧ time(T2)

addEmployee(E,Name,Sal,T) ← hire(E,Name,Sal,Dep,T) ∧ department(Dep,DName, MinSal, MaxSal,T_{pre}) ∧ $T_{pre} = T-1$ ∧ time(T)

addEmployee(E,Name,Sal,T) ← newDept(D,N,Min,Max,E,Name,Sal,T) ∧ time(T)

delEmployee(E,T) ← fire(E,T) ∧ employee(E,Name,Sal,T_{pre}) ∧ ¬hasDep(E,T_{pre}) ∧ $T_{pre}=T-1$ ∧ time(T)

hasDep(E,T) ← worksIn(E,D,T) ∧ time(T)

Boss

$boss(E,Name,Sal,Phone,T) \leftarrow addBoss(E,Name,Sal,Phone,T2) \wedge \neg deletedBoss(E,T2,T) \wedge T2 \leq T \wedge time(T)$

$deletedBoss(E,T1,T2) \leftarrow delBoss(E,T) \wedge T>T1 \wedge T \leq T2 \wedge time(T1) \wedge time(T2)$

$addBoss(E,Name,Sal,Phone,T) \leftarrow promote(E,Phone,T) \wedge employee(E,Name,Sal,T_{pre}) \wedge \neg isManager(E,T_{pre}) \wedge T_{pre} = T\text{-}1 \wedge time(T)$

$isManager(E,T) \leftarrow manages(E,D,T) \wedge time(T)$

$delBoss(E,T) \leftarrow delEmployee(E,T) \wedge time(T)$

Department

$department(D,Name,Min,Max,T) \leftarrow addDepartment(D,Name,Min,Max,T2) \wedge \neg deletedDepartment(D,T2,T) \wedge T2 \leq T \wedge time(T)$

$deletedDepartment(D,T1,T2) \leftarrow delDepartment(D,T) \wedge T>T1 \wedge T \leq T2 \wedge time(T1) \wedge time(T2)$

$addDepartment(D,Name,Min,Max,T) \leftarrow newDept(D,Name,Min,Max,E,MgrN,MgrS,T) \wedge time(T)$

$delDepartment(D,T) \leftarrow removeDept(D,T) \wedge department(D,Name,Min,Max,T_{pre}) \wedge T_{pre}=T\text{-}1 \wedge time(T)$

WorksIn

$worksIn(E,D,T) \leftarrow addWorksIn(E,D,T2) \wedge \neg deletedWorksIn(E,D,T2,T) \wedge T2 \leq T \wedge time(T)$

$deletedWorksIn(E,D,T1,T2) \leftarrow delWorksIn(E,D,T) \wedge T>T1 \wedge T \leq T2 \wedge time(T1) \wedge time(T2)$

$addWorksIn(E,D,T) \leftarrow hire(E,Name,Sal,D,T) \wedge department(D,DName, MinSal, MaxSal,T_{pre}) \wedge T_{pre} = T\text{-}1 \wedge time(T)$

$delWorksIn(E,D,T) \leftarrow delEmployee(E,T) \wedge worksIn(E,D,T_{pre}) \wedge T_{pre}=T\text{-}1 \wedge time(T)$

$delWorksIn(E,D,T) \leftarrow delDepartment(D,T) \wedge worksIn(E,D,T_{pre}) \wedge T_{pre}=T\text{-}1 \wedge time(T)$

Manages

$manages(E,D,T) \leftarrow addManages(E,D,T2) \wedge \neg deletedManages(E,D,T2,T) \wedge T2 \leq T \wedge time(T)$

$deletedManages(E,D,T1,T2) \leftarrow delManages(E,D,T) \wedge T>T1 \wedge T \leq T2 \wedge time(T1) \wedge time(T2)$

$addManages(E,D,T) \leftarrow newDept(D,DName,MinS,MaxS,E,MgrN,MgrS,T_{pre}) \wedge T_{pre} = T\text{-}1 \wedge time(T)$

$delManages(E,D,T) \leftarrow delEmployee(E,T) \land manages(E,D,T_{pre}) \land T_{pre}=T-1 \land time(T)$

$delManages(E,D,T) \leftarrow delDepartment(D,T) \land manages(E,D,T_{pre}) \land T_{pre}=T-1 \land time(T)$

WorkingTeam

$workingTeam(W,Name,Insp,T) \leftarrow addWorkingTeam(W,Name,Insp,T2) \land$
$T2 \leq T \land time(T)$

$addWorkingTeam(W,Name,Insp,T) \leftarrow newTeam(W,Name,Insp,T) \land$
$employee(Insp,InspName, Sal, T_{pre}) \land T_{pre}= T-1 \land time(T)$

Audits

$audits(Insp,W,T) \leftarrow addAudits(Insp,W,T2) \land \neg deletedAudits(Insp,W,T2,T) \land$
$T2 \leq T \land time(T)$

$deletedAudits(Insp,W,T1,T2) \leftarrow delAudits(Insp,W,T) \land$
$T>T1 \land T \leq T2 \land time(T1) \land time(T2)$

$addAudits(Insp,W,T) \leftarrow newTeam(W,Name,Insp,T) \land$
$employee(Insp,InspName, Sal, T_{pre}) \land T_{pre}= T-1 \land time(T)$

$delAudits(E,W,T) \leftarrow delEmployee(E,T) \land audits(E,W,T_{pre}) \land T_{pre}=T-1 \land time(T)$

Member

$member(M,Emp,WorkTeam,T) \leftarrow addMember(M,Emp,WorkTeam,T2)$
$\land \neg deletedMember(M,T2,T) \land T2 \leq T \land time(T)$

$deletedMember(M,T1,T2) \leftarrow delMember(M,T) \land T>T1 \land T \leq T2 \land time(T1) \land time(T2)$

$addMember(M,Emp,WorkTeam,T) \leftarrow newMember(M,Emp,WorkTeam,Rec,T) \land$
$employee(Emp,EmpName, Sal, T_{pre}) \land$
$workingTeam(WorkTeam,WTName,T_{pre}) \land$
$T_{pre}= T-1 \land time(T)$

$delMember(M,T) \leftarrow delEmployee(E,T) \land member(M,E,W,T_{pre}) \land T_{pre}=T-1 \land time(T)$

HasRecruited

$hasRecruited(Rec,Mem,T) \leftarrow addHasRecruited(Rec,Mem,T2) \land$
$\neg deletedHasRecruited(Rec,Mem,T2,T) \land$
$T2 \leq T \land time(T)$

$deletedHasRecruited(Rec,Mem,T1,T2) \leftarrow delHasRecruited(Rec,Mem,T) \land$
$T>T1 \land T \leq T2 \land time(T1) \land time(T2)$

$addHasRecruited(Rec,Mem,T) \leftarrow newMember(Mem,Emp,WT,Rec,T) \wedge$
$employee(Emp,EmpName, Sal, T_{pre}) \wedge$
$workingTeam(WT,WTName,T_{pre}) \wedge T_{pre} = T\text{-}1 \wedge time(T)$
$delHasRecruited(Rec,Mem,T) \leftarrow delMember(Rec,T) \wedge$
$hasRecruited(Rec,Mem,T_{pre}) \wedge T_{pre} = T\text{-}1 \wedge time(T)$
$delHasRecruited(Rec,Mem,T) \leftarrow delMember(Mem,T) \wedge$
$hasRecruited(Rec,Mem,T_{pre}) \wedge T_{pre} = T\text{-}1 \wedge time(T)$

Constraints generated

← newDept(D,N,Mi,Ma,T) ∧ newDept(D2,N2,Mi2,Ma2,T) ∧ D<>D2
← newDept(D,N,Mi,Ma,T) ∧ newDept(D2,N2,Mi2,Ma2,T) ∧ N<>N2
← newDept(D,N,Mi,Ma,T) ∧ newDept(D2,N2,Mi2,Ma2,T) ∧ Mi<>Mi2
← newDept(D,N,Mi,Ma,T) ∧ newDept(D2,N2,Mi2,Ma2,T) ∧ Ma<>Ma2
← removeDept(D,T) ∧ removeDept(D2,T) ∧ D <>D2
← hire(E,N,S,D,T) ∧ hire(E2,N2,S2,D2,T) ∧ E<>E2
← hire(E,N,S,D,T) ∧ hire(E2,N2,S2,D2,T) ∧ N<>N2
← hire(E,N,S,D,T) ∧ hire(E2,N2,S2,D2,T) ∧ S<>S2
← hire(E,N,S,D,T) ∧ hire(E2,N2,S2,D2,T) ∧ D<>D2
← fire(E,T) ∧ fire(E2,T) ∧ E<>E2
← newTeam(W,N,I,T) ∧ newTeam(W2,N2,I2,T) ∧ W<>W2
← newTeam(W,N,I,T) ∧ newTeam(W2,N2,I2,T) ∧ N<>N2
← newTeam(W,N,I,T) ∧ newTeam(W2,N2,I2,T) ∧ I<>I2
← newMember(M,E,W,R,T) ∧ newMember(M2,E2,W2,R2,T) ∧ M<>M2
← newMember(M,E,W,R,T) ∧ newMember(M2,E2,W2,R2,T) ∧ E<>E2
← newMember(M,E,W,R,T) ∧ newMember(M2,E2,W2,R2,T) ∧ W<>W2
← newMember(M,E,W,R,T) ∧ newMember(M2,E2,W2,R2,T) ∧ R<>R2
← newDept(D,N,Mi,Ma,T) ∧ removeDept(D2,T)
← newDept(D,N,Mi,Ma,T) ∧ hire(E,N2,S,D2,T)
← newDept(D,N,Mi,Ma,T) ∧ fire(E,T)
← newDept(D,N,Mi,Ma,T) ∧ newTeam(W,N2,I,T)
← newDept(D,N,Mi,Ma,T) ∧ newMember(M,E,W,R,T)
← removeDept(D,T) ∧ hire(E,N,S,D,T)
← removeDept(D,T) ∧ fire(E,T)
← removeDept(D,T) ∧ newTeam(W,N,I,T)
← removeDept(D,T) ∧ newMember(M,E,W,R,T)

$\leftarrow$ hire(E,N,S,D,T) $\wedge$ fire(E2,T)

$\leftarrow$ hire(E,N,S,D,T) $\wedge$ newTeam(W,N,I,T)

$\leftarrow$ hire(E,N,S,D,T) $\wedge$ newMember(M,E2,W,R,T)

$\leftarrow$ fire(E,T) $\wedge$ newTeam(W,N2,I,T)

$\leftarrow$ fire(E,T) $\wedge$ newMember(M,E2,W,R,T)

$\leftarrow$ newTeam(W,N,I,T) $\wedge$ newMember(M,E,W2,R,T)