UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

ALBERT TORT PUGIBET

PHD THESIS

TESTING AND TEST-DRIVEN DEVELOPMENT OF CONCEPTUAL SCHEMAS

ADVISED BY:
DR. ANTONI OLIVÉ RAMON
DR. MARIA-RIBERA SANCHO SAMSÓ

Barcelona, 2012

A thesis presented by Albert Tort Pugibet
in partial fulfillment of the requirements for the degree of
Doctor per la Universitat Politècnica de Catalunya

Acknowledgements

"A dream you dream alone is only a dream.
A dream you dream together is reality."
John Lennon

Foremost, I would like to thank Antoni Olivé and Maria-Ribera Sancho for his support, rigor and guidance. I am really grateful to them for the confidence placed in me. It has been a pleasure to have the opportunity of learning from them. They will always be more than my PhD advisors.

I would also like to thank Prof. Ernest Teniente, Dr. Horacio Rodríguez, Prof. Oscar Pastor, Prof. John Krogstie and Prof. Esteban Zimányi for accepting to be members of the examination panel.

Thanks to my colleagues in the *Grup de Recerca en Modelització Conceptual* and the *Department of Service and Information System Engineering* for their interesting and useful comments on my research work and for their friendship.

I am also especially grateful to those people who know that they are part of my life.

Finally, thanks to my family for their support and for being always close to me.

This work has been partly supported by the Spanish Ministerio de Ciencia y Tecnología and FEDER under the project TIN2008-00444, Grupo Consolidado.

Abstract

The traditional focus for Information Systems (IS) quality assurance relies on the evaluation of its implementation. However, the quality of an IS can be largely determined in the first stages of its development. Several studies reveal that more than half the errors that occur during systems development are requirements errors. A requirements error is defined as a mismatch between requirements specification and stakeholders' needs and expectations.

Conceptual modeling is an essential activity in requirements engineering aimed at developing the conceptual schema of an IS. The conceptual schema is the general knowledge that an IS needs to know in order to perform its functions. A conceptual schema specification has semantic quality when it is valid and complete. Validity means that the schema is correct (the knowledge it defines is true for the domain) and relevant (the knowledge it defines is necessary for the system). Completeness means that the conceptual schema includes all relevant knowledge. The validation of a conceptual schema pursues the detection of requirements errors in order to improve its semantic quality.

Conceptual schema validation is still a critical challenge in requirements engineering. In this work we contribute to this challenge, taking into account that, since conceptual schemas of IS can be specified in executable artifacts, they can be tested. In this context, the main contributions of this Thesis are (1) an approach to test conceptual schemas of information systems, and (2) a novel method for the incremental development of conceptual schemas supported by continuous test-driven validation. As far as we know, this is the first work that proposes and implements an environment for automated testing of UML/OCL conceptual schemas, and the first work that explores the use of test-driven approaches in conceptual modeling.

The testing of conceptual schemas may be an important and practical means for their validation. It allows checking correctness and completeness according to stakeholders' needs and expectations. Moreover, in conjunction with the automatic check of basic test adequacy criteria, we can also analyze the relevance of the elements defined in the schema. The testing environment we propose requires a specialized language for writing tests of conceptual schemas. We defined the Conceptual Schema Testing Language (CSTL), which may be used to specify automated tests of executable schemas specified in UML/OCL. We also describe a prototype implementation of a test processor that makes feasible the approach in practice.

The conceptual schema testing approach supports test-last validation of conceptual schemas, but it also makes sense to test incomplete conceptual schemas while they are developed. This fact lays the groundwork of Test-Driven Conceptual Modeling (TDCM), which is our second main contribution. TDCM is a novel conceptual modeling method based on the main principles of Test-Driven Development (TDD), an extreme programming method in which a software system is developed in short iterations driven by tests. We have applied the method in several case studies, in the context of Design Research, which is the general research framework we adopted. Finally, we also describe an integration approach of TDCM into a broad set of software development methodologies, including the Unified Process development methodology, MDD-based approaches, storytest-driven agile methods and goal and scenario-oriented requirements engineering methods.

Contents

1 Introduction

Throughout history, testing has been a widely used technique in order to increase confidence about the quality of human-developed artifacts.

In many scientific and technical fields that have been traditionally a base for social progress, such as medical research, civil engineering or aeronautics, testing is a critical activity to enhance confidence on quality.

Over the last decades, **software has become an intrinsic part of business and society**. All developed economies depend on software. In the United States, the *National Institute of Standards and Technology* reported in 2002 that software errors cost the U.S. economy an estimated $59.5 billion annually (RTI International 2002).

Furthermore, social and technological progress has increased the complexity and the diversity of domains in which software is expected to contribute. Therefore, in the 60s, software engineering was born to face up the complexity of software development.

Software engineering is a discipline that promotes software engineers to "adopt a systematic and organized approach to their work and use appropriate tools and techniques depending on the problem to be solved, the development constraints and the resources available" (Sommerville 2010). Several software engineering methods and techniques have been proposed in order to develop software according to user expectations, quality criteria, estimated costs and planned delivery time.

In the context of software engineering, the need for and the importance of **software testing** are undisputed (Glass 2008). We adopt the precise and concise definition of testing proposed by Meyer: "To test a program is to try to make it fail", from which the goal of testing becomes "to uncover faults by triggering failures" (Meyer 2008).

In the last decades, most work on software testing has been focused on testing software code. In this direction, several testing languages, methods and techniques have been proposed to test software implementations (Myers et al. 2004). However, the quality of an information system is largely determined early in the development cycle, especially during the requirements engineering stage. **Requirements engineering** comprises the elicitation, specification and validation of the expected functions and quality constraints of the system (Van Lamsweerde 2009, Pohl 2010).

Empirical studies show that more than half the errors that occur during systems development are requirements errors (Moody 2005). A requirements error is defined as a mismatch between requirements specification and stakeholders' requirements. Moreover, requirements errors are usually much more expensive to correct than errors introduced during design or implementation (Endres et al. 2003).

These observations justify the necessity of making quality assurance efforts in the requirements engineering stage, aimed at the **early detection of software errors.**

A big range of software engineering methods supports the development of **information systems** by considering requirements engineering as an essential activity. An information system is aimed at managing the information of a domain by performing the following functions (1) Maintaining a consistent representation of the state of the domain of interest, (2) Providing information about the state, and (3) performing actions to change the state of the domain (Boman et al. 1997).

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).

Most of the existing information systems development methods include conceptual modeling activities in requirements engineering stages. The purpose of **conceptual modeling** is to determine and define the conceptual schema of an information system, which must include all relevant static and dynamic aspects of its domain.

Correctness and completeness are two fundamental semantic **quality** properties of conceptual schemas (Lindland et al. 1994). Correctness means that the defined knowledge is true for the domain, and completeness means that all relevant knowledge is defined in the schema, according to the needs and expectations of stakeholders. The validation of these properties is a research challenge, which is open to new contributions.

If conceptual schemas are specified in a formal conceptual modeling language, then they may be executable. Therefore, the following question arises: Can we test conceptual schemas as a requirements validation and quality assurance technique, in order to enhance the early detection of errors in information systems development?

In this thesis, we present two main contributions:

- An approach to **test conceptual schemas** in order to validate them according to stakeholders' needs and expectations.
- A method for performing **test-driven development of conceptual schemas** by continuous validation of its semantic quality.

In this introductory chapter, we explain the essentials of these two main contributions. Firstly, we introduce conceptual modeling (Section 1.1), which is the activity aimed to develop conceptual schemas. Secondly, we state the problem of conceptual schema validation (Section 1.2). After that, we present the research questions addressed in this Thesis (Section 1.3). Finally, in Section 1.4, we introduce Design Science Research, which is the research approach adopted for the development of this Thesis. We also point out the Thesis contributions in the context of this research framework.

1.1 On Conceptual Modeling

In the context of software engineering, a model is "an artifact formulated in a modeling language describing a system" (Kühne 2005). Most software development methods include specification and design of models in their activities and artifacts. OMG's Model-Driven Development (MDD) approach (Mellor et al. 2002, Pastor et al. 2007) conceives software development as a sequence of evolution of models, from Platform-Independent Models (PIM) to Platform-Specific Models (PSM).

A conceptual schema defines the general knowledge that an information system needs to know in order to perform its functions (Olivé 2007). A conceptual schema can be represented as a PIM which consists of a structural (sub)schema and a behavioral (sub)schema. The structural schema specifies the static knowledge of the system, which determines the valid states of the domain. The behavioral schema specifies the events of the system, which determine the valid state changes and queries. Conceptual modeling is an essential requirements engineering activity. The *principle of necessity* of conceptual schemas (Olivé 2007) states that it is not possible to develop a system without considering its conceptual schema. The main purpose of conceptual modeling is eliciting and defining the conceptual schema of an information system.

Requirements engineering is the branch of software engineering concerned with the real-world goals for, functions of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software behavior, and to their evolution over time and across software families. The main result of a requirements engineering process is a set of artifacts that describe the system the users require and that the designers have to design and build (Van Lamsweerde 2009, Pohl 2010). One of these artifacts is the specification of the conceptual schema (in the desired level of formalism), which defines the functions that the system needs to perform and the required structural knowledge about the domain.

Conceptual schemas can be formally specified by using the Unified Modeling Language (UML) (Object Management Group (OMG) 2009, Booch et al. 2005), and the Object Constraint Language (OCL) (Object Management Group (OMG) 2010b). UML is the *de facto* standard for representing conceptual schemas graphically. Some schema elements such as integrity constraints, derivation rules or the effect of events cannot be graphically represented, but they can be formally defined in OCL.

Fig. 1 shows a simple fragment of the structural schema of a meeting scheduler system. The conceptual schema specifies the event *MeetingRequest*, which creates a non-scheduled meeting with its subject, the initiator of the meeting, the set of possible dates and the set of invited participants.

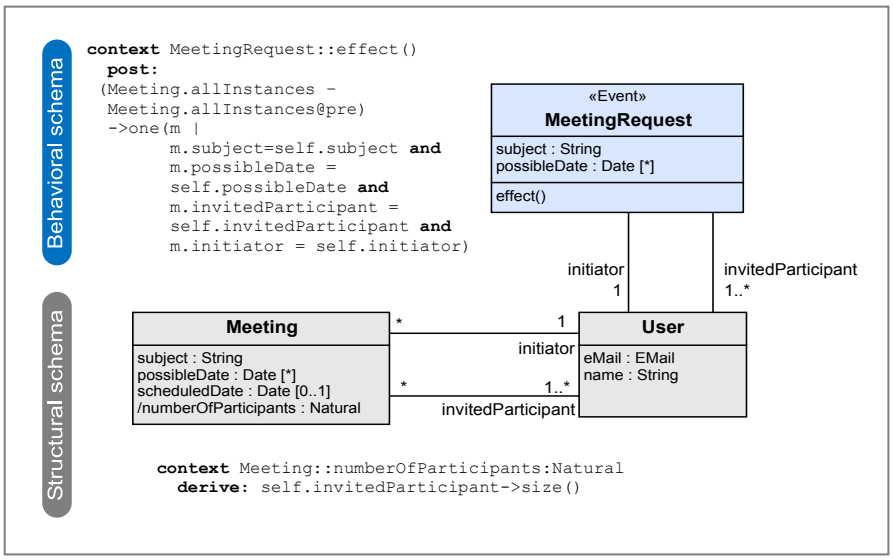


Fig. 1. Conceptual schema fragment of a meeting scheduler system

1.2 On Conceptual Schema Validation

According to (IEEE 1998b), "validation concerns the process of examining a product to determine conformity with user needs". In this context, the main purpose of conceptual schema validation is to ensure correctness and completeness (Lindland et al. 1994, Moody et al. 2003) according to stakeholders' needs and expectations.

A big range of techniques has been already proposed in order to achieve (at different degrees) correct and complete conceptual schemas. Some popular techniques are paraphrasing in natural language, generation of abstractions and abstracts, explanation generation, constraint acquisition, simulation and animation, etc. These techniques are reviewed in the state of the art presented in Chapter 3.

There has been also a lot of work on automated reasoning procedures for checking internal schema properties such as schema satisfiability, liveliness of entity types and relationship types, contradictions between integrity constraints, etc. (see Section 3.2.7). Satisfying these internal properties is a necessary (but not sufficient) condition for correctness. These verification techniques do not take into account the needs and expectations of stakeholders. The conceptual schema testing approach proposed in this work may be used with other existing verification and validation techniques, in order to achieve higher levels of completeness and correctness by considering expected scenarios.

As an example, the schema fragment of Fig. 1 is syntactically correct (it complies with UML/OCL language rules) and satisfiable (the schema admits at least one non-empty state of the information base), and no inconsistent integrity constraints are found on it. However, if we take into account the expectations of some stakeholders shown in Fig. 2, the schema is incomplete and incorrect. The reason is that the schema does not have the knowledge to satisfy the expectations highlighted in bold because (1) there is no constraint to prevent two users with the same email; (2) there is no event type to schedule a meeting in a date; and (3) the derivation rule *numberOfParticipants* does not take into account that the initiator of a meeting is also a participant, as it is expected by one of the expectations of Fig. 2.

The conceptual schema testing approach and the Test-Driven Conceptual Modeling method proposed in this Thesis are aimed to enhance validation of the semantic quality (correctness, relevance and completeness) by considering the needs and expectations of stakeholders involved in the system under development.

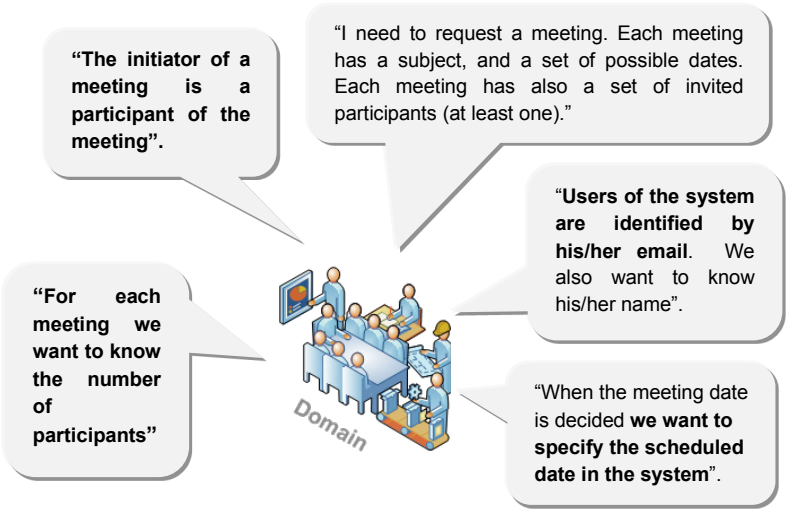


Fig. 2. Stakeholders' needs and expectations

1.3 Research Questions

In this section, we introduce the addressed research questions, which are the base for the contributions of this Thesis. The contributions are explained in detail in the following chapters of this document.

1.3.1 Testing Conceptual Schemas

Most work has been devoted to test software implementations (code). Code testing assumes that the System Under Test (SUT) consists of programs (objects, components) that provide only a set of operations, and testing a SUT means calling those operations with appropriate context and input parameters, and checking that they return the expected outputs. For example, the recent UML Testing Profile (UTP) is based on this assumption (Object Management Group (OMG) 2005, Baker et al. 2008) and the same happens in popular testing frameworks like JUnit (Gamma et al. 1999).

If a conceptual schema were like an ordinary program, then its testing would not be remarkably different from testing a program. However, a conceptual schema is knowledge or, more precisely, it is the general knowledge that an information system needs to know about the domain and about the functions it has to perform (Olivé 2007). Consequently, we may find some similarities between testing a program and testing a conceptual schema, but there are significant differences. In contrast with lines of code, conceptual schemas explicitly define concepts, relationships between these concepts, integrity constraints, derived information, domain events, queries, etc.

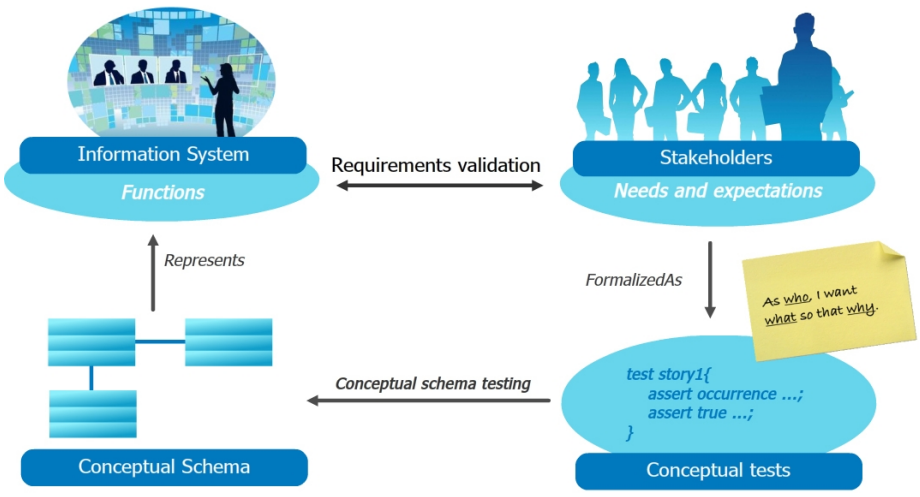


Fig. 3. Conceptual schema testing overview

Currently, most work in conceptual modeling assumes that conceptual schemas can be executable (Mellor et al. 2002, Insfrán et al. 2002, Olivé 2005). Then, a first research question naturally arises:

Main Research Question 1.
Can we test conceptual schemas?

In order to answer *Research Question 1,* in this Thesis we explore what it means to test conceptual schemas, and we present a language and a testing processor for writing and running automated tests of conceptual schemas. The purpose of conceptual schema testing is enhancing its validation according to stakeholders' needs and expectations.

As shown in Fig. 3, conceptual schema testing contributes to requirements validation. Conceptual test cases formally represent expected user stories, and these test cases are checked on the conceptual schema, which formally represents the functional requirements of the system.

In order to answer *Research Question 1,* it is required to address the following specific research questions:

Research Question 1.1.
What it means to test conceptual schemas?

Research Question 1.2.
Which kinds of tests are required to test conceptual schemas?

Research Question 1.3.
Why do we want to test conceptual schemas?

Research Question 1.4.
Which are the requirements of an environment for conceptual schema testing?

Research Question 1.5.
How can we determine the suitability of a test for testing a conceptual schema?

1.3.2 Test-Driven Development of Conceptual Schemas

Test-Driven Development (TDD) (Janzen et al. 2005, Beck 2003) is an extreme programming development method (Beck et al. 2001) in which a software system is implemented in short iterations. In each iteration the developer: (1) Writes a test for the next bit of functionality that he wants to add; (2) Writes the functional code until the test passes; and (3) Refactors both new and old code to make it well structured.

If conceptual schema testing is feasible as considered in the first contribution of this research work (Section 1.3.1), then another main research question arises:

Main Research Question 2.
Can we develop conceptual schemas using a TDD-based method?

In this Thesis, we present a TDD-based method to elicit and define conceptual schemas of information systems. We name it Test-Driven Conceptual Modeling (TDCM). In TDCM, conceptual schema development is driven by test cases that formally specify concrete needs and expectations about the functions of the system. Furthermore, TDCM integrates and fosters continuous validation during conceptual modeling, by taking into account that the quality of a conceptual schema should not be considered as an afterthought and it should be aimed for in each step of the conceptual modeling process.

In order to answer *Research Question 2,* it is required to address the following specific research questions:

Research Question 2.1.
Which are the activities and the process that define TDCM?

Research Question 2.2.
How TDCM may contribute to the quality of conceptual schemas?

Research Question 2.3.
Can TDCM be integrated in existing requirements engineering and software development methods?

Research Question 2.4.
Which are the requirements of an environment to support TDCM?

1.4 The Research Approach

The overall framework of the research presented in this Thesis is that of Design Science (Hevner et al. 2004). In this section, we introduce the Design Science Research framework, and we present the main contributions of the Thesis in the context of this research approach.

1.4.1 Design Science Research

According to (Brinkkemper 1996), it is convenient to adopt a “discipline to design, construct and adapt methods, techniques and tools for the development of information systems”.

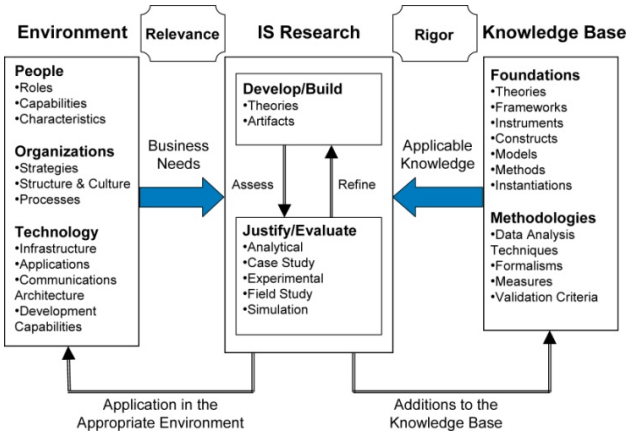


Fig. 4. Design Science Research framework

The Association of Information Systems (AIS) states that Design Science Research (DSR) is "another lens or set of analytical techniques and perspectives for performing research in IS" (Association for Information Systems (AIS) 2009). DSR is a problem-solving paradigm based on the creation and evaluation of artifacts intended to solve identified organizational problems.

(Hevner et al. 2004) argues that "the design-science paradigm has its roots in engineering and the sciences of the artificial". However, in DSR, "artifacts are not exempt from natural laws of behavioral theories. To the contrary, their creation relies on existing kernel theories". This idea is the base of the *Information Systems Research Framework* proposed by (Hevner et al. 2004). Fig. 4 shows the schema of this research framework. Its main principles are the base of our research approach.

Several guidelines are proposed in order to perform Design Science Research:

- *Design as an artifact*: DSR must produce a viable artifact in the form of a construct, a model, a method, or an instantiation.
- *Problem relevance*: The objective of DSR is to develop technology-based solutions to significant and relevant business problems.
- *Design evaluation*: The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods.
- *Research rigor*: DSR relies upon the application of rigorous methods in both the construction and evaluation of the design artifact.

- *Design as a search process*: The search for an effective artifact requires using available means to reach desired ends while satisfying laws in the problem environment.

- *Communication of research*: DSR research must be presented effectively both to technology-oriented as well as management-oriented audiences.

Additionally to the popular design research framework proposed by (Hevner et al. 2004), other papers have contributed to DSR. (Peffers et al. 2007) argued that a framework with a set of guidelines is not sufficient in order to provide a DSR methodology (DSRM). Consequently, a process model was proposed in order to suggest how to apply DSR.

Fig. 5 illustrates the process model proposed in (Peffers et al. 2007), which can be adapted in each particular research process. This model suggests that a research process should start by identifying the problem, its motivation and its relevance. Then, by inference, an initial definition of the research objectives should be defined. After that, the design and development of an artifact for solving the problem may be performed. This task may imply the refinement of the research objectives iteratively. Once we reach a stable artifact, it is important to demonstrate that it is able to solve the problem in a suitable context. Furthermore, an evaluation of the solution should be carried out in order to determine its effectiveness and efficiency. Again, the evaluation of the artifact may suggest refinements to the artifact design and to the research objectives. Finally, the achieved solution needs to be communicated to the research community. The feedback obtained by the community members may be useful to improve the presented solutions, to refine the research objectives or to identify relevant future work.

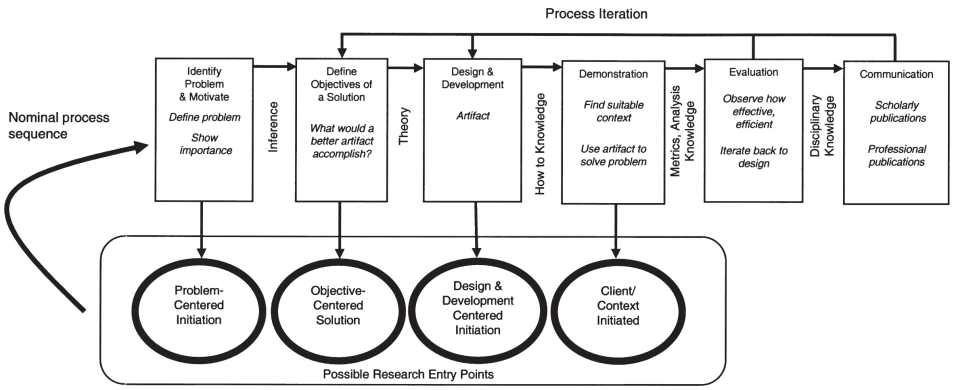


Fig. 5. DSRM Process Model

The DSRM Process model has been taken into account during the planning and development of this Thesis.

Finally, for the development of the Test-Driven Conceptual Modeling method, we also took into account that three perspectives should be considered when developing an engineering method (Lings et al. 2004): the *method-in-concept* (the formal definition of the method, consisting of the main activities and its dependencies), the *method-in-practice* (how the *method-in-concept* should be used to make it effective) and the *method-in-tool* (which are the requirements of a tool to support the *method-in-practice*). The three perspectives are dependent, and they are usually in tension between them.

1.4.2 Contributions of the Thesis in the Context of Design Science Research

In the following, we state the contributions of this Thesis by taking into account the Design Science Research Framework explained in Section 1.4.1.

This Thesis is aimed to contribute to the development of correct, relevant and complete conceptual schemas. The problem is significant because (1) each information system development project requires the development of its conceptual schema according to the principle of necessity of conceptual schemas, which states that "developers need to know the conceptual schema in order to develop an information system", and (2) that conceptual schemas must be correct, relevant and complete (Olivé 2005). On the other hand, it is widely recognized that errors at the conceptual level must be detected and corrected as soon as possible.

In this Thesis, we present an approach for testing conceptual schemas and a novel method for the development of conceptual schemas that we call Test Driven Conceptual Modeling (TDCM), which is a variant of the popular Test-Driven Development (TDD).

As far as we know, this is the first work that proposes and implements an environment for automated testing of UML/OCL executable conceptual schemas and the first work that explores the use of TDD in conceptual modeling.

From the research questions, the problem context and the objectives introduced in Chapter 1, we developed a testing environment (Chapter 4) and the TDCM method (Chapter 8 and 10) (*research contributions*) in order to contribute to conceptual schema validation in Information Systems development (Chapter 2) (*problem relevance*).

We also developed a tool (Chapter 5) to put into practice both the testing approach and the method. The proposed testing environment, the TDCM method and the associated tool are artifacts which have been incrementally designed as a search process in the context of Design Research (*design as an artifact*). The research work was evaluated by applying the testing approach and the method in several case studies (Chapters 6 and 9) by using the associated tool under development (*design evaluation*).

The contributions of this Thesis are based on the state of the art related to conceptual schema validation (Chapter 3) and Test-Driven Development fundamentals (Chapter 7) (*research rigor)*.

We also made new assumptions based on the state of the art and according to the thesis objectives. Then, we incrementally experienced them by using a tool (Chapter 5) for analyzing its effectiveness and obtaining feedback to drive the research process (*design as a search process*).

Finally, we have published the main contributions in research publications aimed to communicate the research results (*communication of research)*.

2

The Challenge of Conceptual Schema Validation

This Thesis contributes to the challenge of conceptual schema validation by the use of testing in conceptual modeling.

In this chapter, we review basic concepts on conceptual modeling, and we describe the elements that constitute the conceptual schema specifications considered in this Thesis (Section 2.1). We also review the main quality properties of conceptual schemas according to well-known conceptual modeling quality frameworks, focusing on those properties addressed in this work (Section 2.2). Finally, we state the problem of validation of conceptual schemas (Section 2.3).

2.1 Basic Concepts on Conceptual Modeling

Every information system embodies a conceptual schema. A conceptual schema defines the general knowledge that an information system needs to know in order to perform its functions (Olivé 2007). Conceptual modeling is an essential requirements engineering activity, the main purpose of which is the development of the conceptual schema of an information system.

We focus on conceptual schemas that comprise the structural and the behavioral knowledge of the system under development. The structural knowledge allows maintaining a consistent representation of the state of the domain. An information

system maintains the representation of the state of the domain in the Information Base (IB) at each moment of the system's lifetime. The IB is an instantiation of the general structural knowledge defined in the conceptual schema. The behavioral knowledge of the conceptual schema corresponds to the functions of the system, which may be specified as a set of events. Each event defines general knowledge related to valid changes in the IB state or queries about the state of the domain.

According to the *principle of conceptualization* (International Standards Organization (ISO) 1982), conceptual schemas "should only include conceptually relevant aspects, both static and dynamic, of the domain, thus excluding all aspects of data representation, physical data organization and access as well as aspects of particular external user representation such as message formats, data structures, etc."

In the context of Model-Driven Development (MDD) (Mellor et al. 2002, Pastor et al. 2007), conceptual schemas are Platform-Independent Models (PIM) that can be manually or automatically transformed into Platform-Specific Models (PSM). When the transformation is automatic, a conceptual schema becomes the last description (of the domain and data management layers) of the information system that needs to be created in its development (Insfrán et al. 2002, Olivé 2005). Moreover, conceptual schemas written in formalized languages like UML/OCL are software artifacts that can be executed to a varying degree (Mellor et al. 2002, Insfrán et al. 2002, Olivé 2005).

We adopt UML/OCL as the conceptual modeling language. In the last years, the UML (Unified Modeling Language) (Object Management Group (OMG) 2009, Booch et al. 2005) has become a *de facto* standard in conceptual modeling. We use UML in conjunction with OCL (Object Constraint Language) (Object Management Group (OMG) 2010b) in order to formally specify integrity constraints, derivation rules and the effect of the events, which cannot be graphically represented by using UML.

In this section, we review the main concepts and notation we have used to define conceptual schemas in this Thesis.

2.1.1 UML/OCL Conceptual Schemas

In this thesis, we deal with conceptual schemas that consist of a structural (sub)schema and a behavioral (sub)schema.

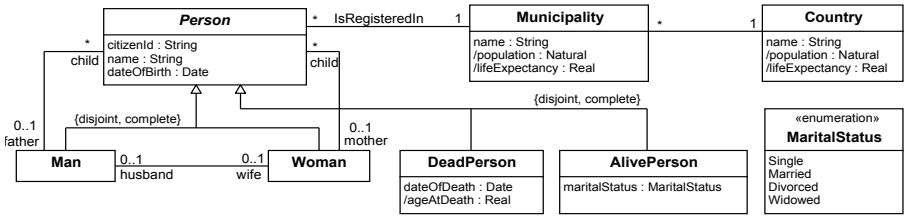
Fig. 6 shows the structural schema of a civil registry domain that will be used as an example in this section. The civil registry records information about the birth and death of the people registered in municipalities. The marital status and the marriage relationships of the inhabitants are also maintained. The main purpose of civil registration systems is computing demographic information such as the population, the life expectancy, etc.

Fig. 7 shows an example of the domain event *Marriage*, in order to illustrate the specification of an event in the behavioral schema.

The Structural Schema

The structural schema consists of a taxonomy of entity types (a set of entity types with their generalization/specialization relationships and the taxonomic constraints), a set of relationship types (either attributes or associations), the cardinality constraints of the relationship types and a set of other constraints formally defined in OCL (OMG 2006).

An *entity type* is a concept whose instances at a given time are identifiable individual objects that are considered to exist in the domain at that time. The instances of an entity type are called *entities*. Entity types are specified in UML as classes, and entities can be represented as UML objects.



OCL Derivation rules

```
context Municipality::population:Natural
derive: self.person->select(oclIsTypeOf(AlivePerson))->size()

context Municipality::lifeExpectancy:Real
derive: let deadPeople:Set(Person)=self.person->select(oclIsTypeOf(DeadPerson))
        in  if deadPeople->size()>0 then
                deadPeople.oclAsType(DeadPerson).ageAtDeath->sum() / deadPeople->size()
            else 0.0
            endif

context Country::population:Natural
derive: self.municipality.population->sum()

context Country::lifeExpectancy:Real
derive: let deadPeople:Set(Person)=
            self.municipality.person->select(oclIsTypeOf(DeadPerson))->asSet()
        in  if deadPeople->size()>0 then
                deadPeople.oclAsType(DeadPerson).ageAtDeath->sum() / deadPeople->size()
            else 0.0
            endif

context DeadPerson::ageAtDeath:Real
 derive: dateOfDeath-dateOfBirth
```

OCL Integrity constraints

```
context Country inv identifiesInhabitantsByCitizenId:
 self.municipality.person->isUnique(citizenId)
```

Fig. 6. Structural schema fragment of a civil registry system

We deal with schemas that may also include data types and enumerations, which are particular kinds of entity types. A *data type* consists of a set of values and a set of literals. The set of values is the population of the data type and is called the *value space* of the type. The set of lexical representations is called the *lexical space* of the type. Each value in the value space is denoted by one or more literals of the lexical space. Values are represented in an information base by means of one of their literals. The value space of a data type does not change over time. We consider the following set of predefined data types:

- *String*. The values are finite-length sequences of characters.
- *Boolean*. The values are {true, false}, represented by the literals {true, false, 1, 0}.
- *Decimal*. The value space is the set of values $i \times 10^{-n}$, where i and n are integers and $n \geq 0$. The lexical space is the set of finite sequences formed by the set of digits {0..9} with a point, optionally preceded by a sign.
- *Integer*. Decimal values without digits in the fractional part. The space of values is the set {…, -2, -1, 0, 1, 2, …}.
- *PositiveInteger*. The value space is the subset {1, 2, …} of integers.
- *NonNegativeInteger*, which we shall call *Natural*. These are the positive integers and zero.
- *Date*. The space of values is the set of dates in the Gregorian calendar.
- *Time*. The value space is the set of times in a day, starting from midnight.
- *DateTime*. The values are specific instants. The value space comprises all valid combinations of Date and Time.

Enumerations are particular kinds of data types whose values are enumerated in the model as enumeration literals.

In the example of Fig. 6, there are seven entity types (*Person*, *Man*, *Woman*, *DeadPerson*, *AlivePerson*, *Country* and *Municipality*). In this example, it has been also defined the enumeration *MaritalStatus*, which allows the enumeration literals *Single, Married, Divorced* and *Widowed*.

A *relationship type* is a concept whose instances at a given time are identifiable individual relationships between entities that are considered to exist in the domain at that time. Each related entity is a *participant* of the corresponding relationship type.

Relationship types may be specified in UML as *associations* or *attributes*. When a relationship type is represented as an association, its instances are represented in UML as *links*. In this work, we allow the use of *n-ary* relationships. An *n-ary* relationship is a relationship between *n* participants, where $n \geq 2$. In the structural schema of Fig. 6, there are two binary ($n = 2$) relationship types (*IsRegisteredIn* and the relationship type between *Municipality* and *Country*). Relationship types have *cardinality constraints,* which constraint their population. Cardinality constraints can be graphically specified in UML. In Fig. 6, the cardinality constraints state that each *Person* must be registered in one *Municipality* and that municipalities belong to one *Country*.

Our conceptual schemas also include the concept of *attribute* of an entity type. Attributes are particular kinds of binary relationship types that allow a subordination of one relationship participant to another (usually a relationship between an entity type and its particular characteristics). The schema reproduced in Fig. 6 shows several examples of attributes. In UML, by default, its multiplicity (the cardinality constraint of the attribute) is 1. For example, a *Person* must have a *citizenId*, a *name*, and a *dateOfBirth.* If the *Person* is an *AlivePerson*, we also need to know its *maritalStatus*.

Other constraints that can be graphically specified in UML are the *taxonomic constraints*. A generalization set satisfies the *covering constraint* if the instances of the *superclass* must be instances of at least one of the *subclasses*. The generalization sets that satisfy the *covering constraint* are called *complete*; otherwise, they are called *incomplete*. Additionally, a generalization set satisfies the *disjointness constraint* if each instance of the *superclass* is an instance of at most one *subclass*. The generalization sets that satisfy the *disjointness constraint* are called *disjoint;* otherwise, they are called *overlapping*. In the schema of Fig. 6, each instance of the entity type *Person* must be either a *Man* or a *Woman*, and also either a *DeadPerson* or an *AlivePerson*.

Those constraints that cannot be represented graphically in the schema can be included as OCL *invariants*. Those IB states which do not satisfy the constraints defined in the schema are considered to be inconsistent. Fig. 6 includes an OCL integrity constraint (*Country::identifiesInhabitantsByCitizenId)* in order to ensure that the inhabitants of a country are identifiable by a unique citizen identifier.

The *Information Base (IB) state* is the set of instances of the entity and relationship types of the conceptual schema at a given time. We assume that an entity may be an instance of several entity types not related by *IsA* relationships. This characteristic is called *multiple classification* and UML admits it. Otherwise, we would say that our IB only admits *single classification* of entity types.

Entity and relationship types may be base or derived. The population of the base entity and relationship types is explicitly represented in the Information Base (IB). If they are derived, there is a formal *derivation rule* in OCL that defines their population in terms of the population of other types. Fig. 6 includes the derivation rules of five derived attributes (*DeadPerson::ageAtDeath*, *Municipality::population*, *Municipality::lifeExpectancy, Country::population* and *Country::lifeExpectancy)*.

The Behavioral Schema

The behavioral schema consists of a set of *event types*. An *event entity* is an instance of an event type. There are two main kinds of event types:

- *Domain events*: A state change that consists of a nonempty set of *structural events* that are perceived or considered a single change in the domain. A s*tructural event* is an elementary change in the population of an entity or relationship type. There are four kinds of structural events: entity insertion, entity deletion, relationship insertion and relationship deletion.

- *Queries*: An external request whose effect is to provide some information (answer) about the domain.

We adopt the view that an event can be modeled as a special kind of entity, which we call *event entity* (Olivé et al. 2006). The main advantage of this method is the uniform treatment given to event and entity types. By this way, event types may have constraints and derived characteristics, additionally to its effect. Moreover, when events types are specified as entity types, specialization allows the incremental definition of new event types, as refinements of their supertypes (Olivé et al. 2006).

The characteristics of an event are the set of relationships (attributes or associations) in which it participates. The constraints are the conditions that events must satisfy in order to occur. An event constraint involves the characteristics and the state of the IB before the event occurrence. An event may occur in the state *S* of the IB if *S* satisfies all constraints, and the event satisfies its event constraints. Each event type has an operation called *effect*() that gives the effect of an event occurrence. The effect is declaratively defined by the postcondition of the operation. We define both the event constraints and the postcondition in OCL.

Given that there is a direct correspondence between events and invocations of system operations, the adaptation of our work to languages that view events as invocations of system operations is straightforward (Larman 2005).

For domain event types, the postcondition defines the state of the IB after the event occurrence. It is assumed that the state of the IB after the event occurrence also satisfies all constraints defined over the IB. Therefore, the effect of a domain event is a state that satisfies the postcondition and all IB constraints. Note that the OCL expressions used in constraints, derivation rules and pre/post conditions are without side-effects. These expressions are evaluated over the IB and their evaluation cannot change the IB.

Given that we want to deal with executable conceptual schemas, we also need a procedural specification of the method of the *effect*() operation. An execution of an *effect*() operation can change the IB. A method is correctly specified if the result it produces always satisfies the postcondition and the IB constraints. UML does not include any particular language for writing methods (Booch et al. 2005). In the work reported here, we write the methods of the *effect*() operations using a subset of the Conceptual Schema Testing Language (CSTL) developed in this Thesis. However, we envision the use of standard languages for writing actions in UML schemas, such as the recent *Action Language for Foundational UML (Alf*) (Object Management Group (OMG) 2010a) proposed by the Object Management Group (OMG), as soon as they become mature and associated compilers are developed.

Fig. 7 shows the complete and formal specification in UML/OCL of the domain event *Marriage* including its initial integrity constraint (*marriageIsAuthorized*), its postcondition and the method of its *effect()* operation.

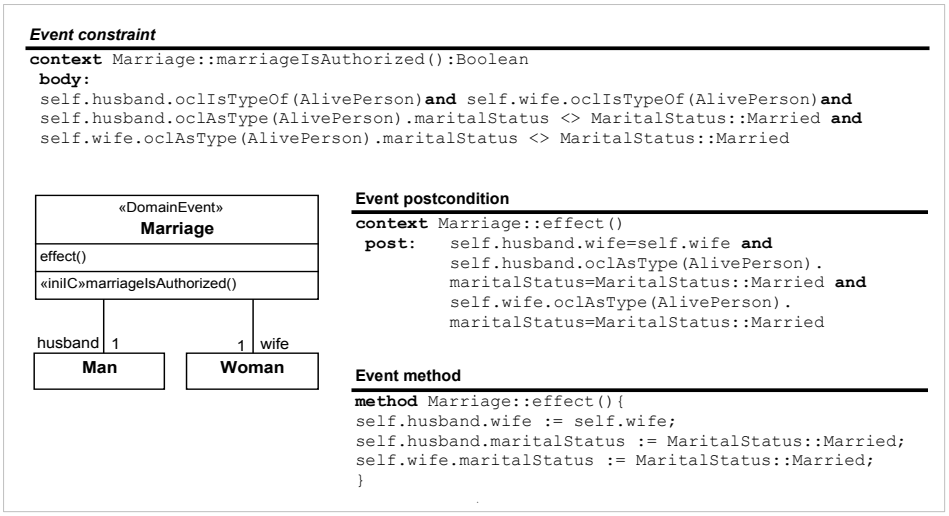


Fig. 7. Marriage domain event example

2.2 Quality of Conceptual Schemas

In general, the quality of a product is the degree to which a set of quality properties is present. Some conceptual modeling quality frameworks have been proposed in the literature in order to state quality properties for conceptual schemas. In (Moody 2005), several existing quality frameworks are reviewed. However, although there is an international standard for evaluating the quality of software products (International Standards Organization (ISO) 2000), there is no specific standard for evaluating the quality of conceptual schemas.

(Moody 2005) argues that, in the absence of a standard, practitioners continue to evaluate conceptual schemas in an *ad hoc* and subjective way, based on common sense and experience. A possible explanation for the non-existence of a conceptual modeling quality standard is that "it is easier to evaluate the quality of a finished product than a logical specification which needs to be aligned with the expectations of people involved in the system under development" (Vliet 2000). In this Thesis, we face the challenge of evaluating fundamental semantic quality properties of conceptual schemas by testing.

Conceptual modeling naturally belongs as a sub discipline of Requirements Engineering in Software Engineering. Therefore, (Moody 2005) suggests that any conceptual quality analysis should comply with both *ISO 9000* and *ISO/IEC 9126*. *ISO 9000* defines a framework of quality concepts, terminology, principles and processes that apply to all software products and services (a conceptual schema is a particular type of product) (International Standards Organization (ISO) 2000). *ISO/IEC 9126* defines a framework for evaluating the quality of software products and covers the software development lifecycle (conceptual schemas exists as models of information systems).

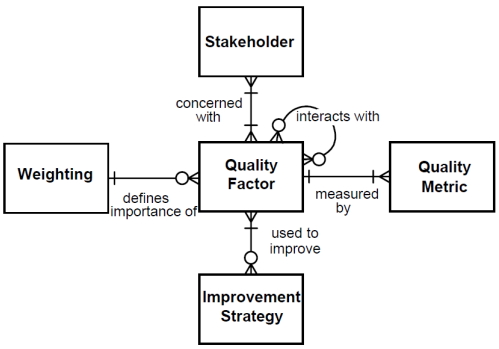


Fig. 8. Moody and Shanks data quality framework

In software development, ISO 9000 defines software quality as "the totality of features and characteristics of a product or service that bear on its ability to satisfy stated or implied needs" (International Standards Organization (ISO) 2000). Similarly, (Moody 2005) adapts this definition to conceptual modeling as "the quality of features and characteristics of a conceptual schema that bear on its ability to satisfy stated or implied needs". Again, this definition makes clear that the quality of a conceptual schema needs to be checked according to the needs and expectations of the people involved in the domain.

Relevant frameworks have been proposed in the literature in order to evaluate the quality of data models such as the one proposed in (Moody et al. 1994) which was refined and extended in (Moody et al. 2003) (Fig. 8). This framework proposes a set of quality factors (completeness, correctness, simplicity, flexibility, integration, understandability and implementability), its relationships with stakeholders, its contribution to the improvement strategy, its importance, and quality measures to evaluate them. The completeness factor analyzes if the model contains all relevant requirements. The quality framework used in this thesis provides a similar definition. However, the correctness factor is not related to stakeholders' expectations and it is defined as the absence of errors when using the modeling language.

There also exist quality frameworks that are focused on the quality of the modeling process. Fig. 9 shows the framework proposed by (Wand et al. 1996). In this framework, the *application domain* is conceptualized by analysts in a conceptual modeling language (*Information system)* by using a grammar (a set of *ontological constructs* that are used to represent the real world*)*. The domain is observed by the stakeholders (*users*), and the conceptual schema is interpreted.

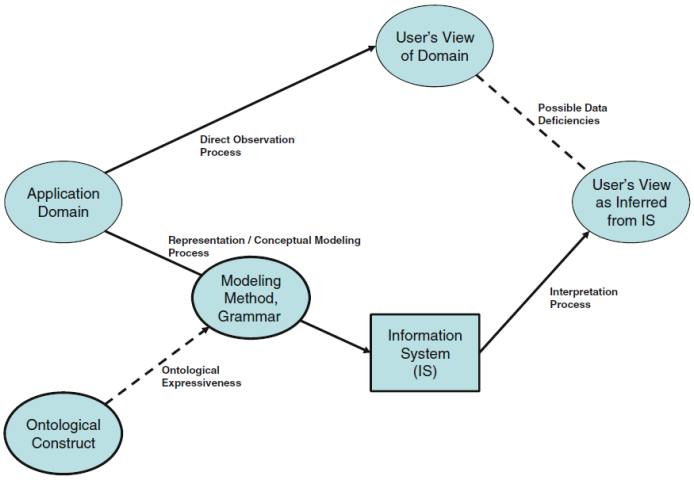


Fig. 9. Wand and Wang's conceptual modeling quality framework

Note that the semantic quality defined in Section 2.2.1 corresponds to the alignment between the interpreted knowledge of the schema and the knowledge of the domain, which is directly observed by the stakeholders. The non-continuous line of Fig. 9 (possible data deficiencies) represents this quality relationship. The conceptual schema testing environment and the TDCM method proposed in this thesis contribute to this quality relationship.

Finally, it is necessary to remark that new frameworks that combine both existing product and process quality frameworks have been also recently published (Nelson et al. 2011).

For the purpose of contextualizing the quality properties considered in this Thesis, we adopt the popular conceptual modeling quality framework proposed by (Lindland et al. 1994) and the extension of this framework presented by (Krogstie et al. 2006).

According to (Lindland et al. 1994), there are three main conceptual schema quality goals: Syntactic quality, semantic quality and pragmatic quality. The *Lindland framework* was extended by the SEQUAL framework (Krogstie et al. 2006). These frameworks consider that conceptual schemas are defined as sets of statements in a modeling language, and they have their foundations in the theory of semiotics (Posner 1987). Semiotics consists of a theory of codes and signs, which are used to convey meaning about things in the world. Semiotics is closely related to the field of linguistics and includes the evaluation of codes and signs based on three main points of view: syntactic (relations among signs in formal structures), semantic (relations between signs and the things to which they refer) and pragmatic (relation between signs and the effects they have on the people who use them).

SEQUAL identifies eight *quality types* and the *quality goals* to be achieved in each type. Table 1 is an overview of the conceptual schema quality properties proposed in the SEQUAL framework.

The main quality properties addressed in this Thesis are described in detail in Section 2.2.1, according to the SEQUAL quality framework. The conceptual schema testing approach and the Test-Driven Conceptual Modeling method presented in this work are focused on *semantic quality* validation, which is pursued by means of *perceived semantic* quality. Semantic quality includes validity and completeness. Validity comprises correctness and relevance. In Section 2.2.2, we briefly describe other quality properties that are not directly addressed in this Thesis. Some of these other properties have an impact on *perceived semantic quality* and, consequently, they have an effect on the quality of the conceptual schema testing process. Therefore, we briefly analyze them, and we explore its relationship with the validation of semantic quality by conceptual schema testing.

Quality type	Goals	Description
Physical quality	Externalization	The conceptual schema is available as a physical artifact, representing the knowledge of some social actor using statements of the modeling language.
	Internalizability	The conceptual schema is available and persistently enabling the model audience to interpret it.
Empirical quality	Minimal error frequency	The conceptual schema can be evaluated looking only at the schema itself, comprising understandability matters such as layout for graphs and readability indexes for text.
Syntactic quality	Syntactic correctness	All statements in the model are according to the syntax and vocabulary of the modeling language.
Semantic quality	Feasible validity	The knowledge in the schema is sufficiently correct and relevant to the problem.
	Feasible completeness	The model contains all valuable statements that would be correct and relevant for the problem domain.
Pragmatic quality	Feasible comprehension	The model can be understood by the audience.
Perceived semantic quality	Perceived validity and completeness	The model is valid and complete according to actors' interpretation of the schema and their current knowledge about the domain.
Social quality	Feasible agreement	Implies resolving inconsistencies by choosing alternatives where benefits of choosing exceed the costs of working out consensus.
Organizational quality	Modeling goals satisfaction	The model satisfies the modeling goals, which define why the conceptual modeling process is undertaken according to the tasks to be performed by each stakeholder in the organization.
Knowledge quality	Feasible knowledge validity and completeness	Validity and completeness taking into account the audience knowledge about the domain.

Table 1. SEQUAL quality framework

2.2.1 Semantic quality properties

Semantic Quality

The contributions of this Thesis are focused on the validation of (feasible and perceived) semantic quality properties. A conceptual schema of an information system has semantic quality when it is valid and complete. *Validity* means that the schema is correct and relevant.

A conceptual schema is *correct* if the knowledge it defines is true for the domain, and it is *relevant* if the knowledge it defines is necessary for the system. *Completeness* means that the conceptual schema includes all relevant knowledge.

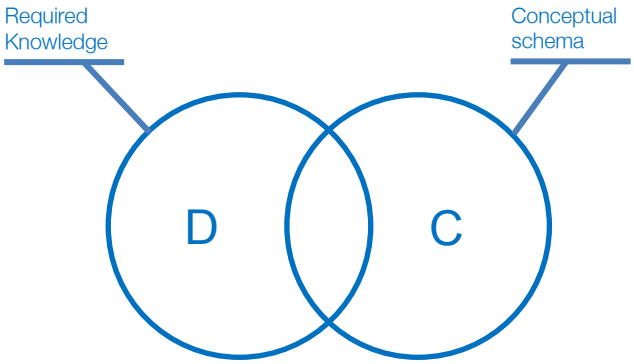


Fig. 10. Relationship between completeness and correctness

Ensuring that a conceptual schema has semantic quality implies checking that the knowledge the system requires to know in order to perform its functions is the same as the knowledge defined by the conceptual schema. Fig. 10 shows the relationship between completeness and correctness. Circle *D* represents the domain knowledge that the system needs to perform its functions. Circle *C* represents the knowledge defined in the conceptual schema. In a complete conceptual schema, *D* is a subset of *C*. In a correct conceptual schema, *C* is a subset of *D*. In a complete and correct conceptual schema, *D* = *C*.

Perceived Semantic Quality

(Krogstie et al. 2006) states that "the primary goal of semantic quality is a correspondence between the externalized model and the domain. However, this

correspondence can neither be established nor checked directly: to build the model, one has to go through the participants' knowledge regarding the domain, and to check the model one has to compare this with the participants' interpretation of the externalized model". According to this analysis, we need to achieve semantic quality by considering *perceived semantic quality,* which is based on the alignment between stakeholders' interpretation of the conceptual schema and their current knowledge of the domain.

Feasible Semantic Quality

(Krogstie et al. 2006) affirms that for anything but extremely simple and highly inter-subjectively agreed domains total validity and completeness cannot be achieved. (Lindland et al. 1994) states that "attempts to do so would require spending unlimited amounts of time and money, which is unacceptable". For this reason, the SEQUAL quality framework introduces a relaxed and realistic kind of validity and completeness (*feasible validity* and *feasible completeness*). Feasible validity and completeness are achieved "not when the model is perfect (which will never happen) but when it has reached a state where further modeling is less beneficial than applying the model in its current state". Considering the terminology of this framework, we may observe that when talking about *semantic quality* checking, we are, in fact, referring to *feasible semantic quality* checking.

The proposed conceptual schema testing approach fosters the validation of *feasible semantic quality,* by specifying the needs and expectations of the people involved in the domain as test cases that are executed. Conceptual test cases formally define concrete user stories that represent the knowledge that is expected to be relevant and correct as perceived by stakeholders. These test cases may be checked against a formally defined and executable conceptual schema.

2.2.2 Other quality properties

Fig. 11 shows the main concepts and relationships considered in the conceptual modeling quality framework that we adopt in this Thesis. As explained in Section 2.2.1, we may observe that semantic quality corresponds to the alignment between the *Model externalization* (the specification of the conceptual schema) and the *Modeling domain* (the domain of interest for the system under development).

Since we cannot directly check this relationship and that we need to interpret the domain and the needs and expectations in accordance with stakeholders (Social and technical actor interpretation), what we observe in quality control is not the actual semantic quality of the schema, but a *perceived semantic quality*.

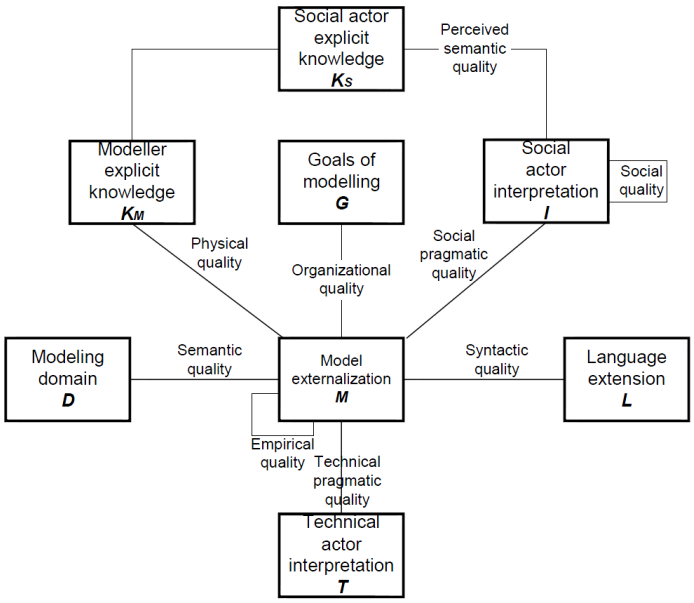


Fig. 11. SEQUAL quality framework

Although the present work is focused on the validation of semantic quality properties, conceptual schema quality comprises other properties that we briefly review in this section. It is important to note that these properties may have an impact on semantic quality, as analyzed in the following.

Physical Quality

Conceptual schemas may be explicitly specified by using conceptual modeling languages such as UML/OCL. These schemas can be stored and distributed in physical supports (paper, disk, etc.).

Two basic quality features determine the physical quality of a conceptual schema:

- *Externalization*: The knowledge of participants is physically represented (externalized) in the conceptual schema.
- *Internalizability*: The conceptual schema representation is persistent (protected against loss or damage) and available (the interested audience can access it).

In this Thesis, we consider formally specified conceptual schemas in an executable form, which are saved as electronic files. These files can be stored in physical memory and they can be easily accessible.

Empirical Quality

Empirical quality deals with the distinction of schema elements, the error frequencies when schemas are written or read, coding aspects such as the graphical shapes being used, and ergonomics (layout mechanisms, aesthetics, etc.). Therefore, empirical quality influences the comprehension of the model. These quality aspects are not addressed in this Thesis.

Syntactic Quality

Syntactic quality refers to the correspondence between the conceptual schema representation (the model) and the language in which the model is written.

Given that conceptual schemas are represented in a language, all statements in the model should be according to the language syntax. In the case of conceptual schemas formally defined in UML and OCL, schemas should be valid instances of UML and OCL metamodels (Object Management Group (OMG) 2009, Object Management Group (OMG) 2010b).

In the context of this Thesis, syntactic quality is a *sine qua non* condition. The reason is that conceptual schema specifications are interpreted by testing software that assumes (and checks) that conceptual schema specifications comply with the syntax of UML and OCL modeling languages.

Pragmatic Quality

Pragmatic quality is defined as the effect that the model has on the participants and the world. The main pragmatic quality goal in conceptual modeling is *comprehension*. Conceptual schemas are functional requirements specifications and, consequently, all concerned parties should understand it.

This quality property is not directly addressed by the main contributions of the thesis, but it is positively influenced when refactoring is performed during TDCM. Moreover, our testing environment and the TDCM method take pragmatic quality into account, because the resultant schema is systematically based on test cases that promote to preserve the names used for concepts and relationships in the user stories acquired through stakeholders.

Social Quality

The main goal for social quality is (feasible) agreement in the knowledge required by the system, and, therefore, in the knowledge to be defined in the conceptual schema. Achieving this quality level may require negotiation, conflicts resolution and capability of reaching agreements on the defined requirements.

This quality property is not directly addressed in this Thesis, because it relies on social capabilities. Nevertheless, both the conceptual schema testing approach and the Test-Driven Conceptual Modeling method presented in this Thesis are able to detect test cases that contain contradictory expectations. Therefore, the present work fosters the detection and resolution of conflicts, although the decisions to solve them require a social process, which is out of the scope of this work.

Organizational Quality

Organizational quality corresponds to the alignment between the conceptual schema representation and the modeling goals. Modeling goals ask the question of why the conceptual modeling process is undertaken according to the tasks performed by the stakeholders involved in the organization. Modeling goals are more related to have an up to date, tailored model to support the individual users in their actual tasks, and is less directly linked to the overall goals of the organization.

This quality property is implicitly addressed in this Thesis, because the resultant conceptual schema obtained by testing is based on a set of test cases, which in turn, are based on stories the source of which are the stakeholders involved in the organization. Therefore, the conceptual schema is influenced by this source and, consequently, the schema may better help users in their organizational tasks.

Knowledge Quality

Knowledge quality corresponds to the relationship degree between the audience knowledge and the domain knowledge. Usually, not all the stakeholders have the same valid knowledge about the domain. Therefore, careful participant selection and stakeholder identification is required.

This quality property is not addressed in this Thesis, although it is necessary to be considered when defining the testing strategy in our conceptual schema testing processes. Obviously, the *quality* of stakeholders' expectations, which are the source for conceptual schema test cases, relies on their knowledge about the domain and, consequently, it must be taken into account.

2.3 Validation of Conceptual Schemas

In software engineering, information systems are developed to satisfy the needs and the expectations of the people involved in a domain. A key activity for the development of an information system is identifying these needs and expectations, as a base for determining the features of the system we are going to develop. The requirements of an information system are the features that the system must have in order to satisfy the stakeholders' needs and expectations.

In general, a requirement is "a condition or capability needed by a user to solve a problem or achieve an objective" (IEEE 1998a). In the context of information systems development, there are two kinds of software requirements: functional requirements and quality requirements. Functional requirements define the functional effects that the software is required to have. The conceptual schema specifies the functional requirements of an information system (Olivé 2007). Quality requirements define constraints on the way the software-to-be should satisfy functional requirements.

Requirements engineering "is the branch of software engineering concerned with the real-world goals for, functions of, and constraints on software systems" (Zave 1997). Requirements engineering is a complex process, because it involves different actors who may all have different views, needs, and interests. Requirements engineering involves requirements elicitation, requirements specification and requirements validation. These processes are far from trivial. (Van Lamsweerde 2009) explains that "We must correctly understand and define what problem needs to be solved. This seems common sense at first sight. However, […] we need to discover, understand, formulate, analyze and agree on the problem to be solved".

Several requirements elicitation techniques are proposed in the literature (data collection, background studies, questionnaires, interviews, storyboards, prototypes, knowledge reuse, etc.) (Van Lamsweerde 2009). Once a requirement is elicited, it is convenient to be specified. After that, requirements specifications need to be verified and validated for ensuring their quality.

2.3.1 The Need for Software Validation

In the context of information systems development, software artifacts may be verified and validated according to a set of quality criteria.

In the literature, we may find several definitions of *verification* and *validation*. For notation purposes, it is important to clarify the difference between both activities. In this Thesis, we adopt the general definitions of these terms proposed in (IEEE 1998b),

which are illustrated in Fig. 12. As explained in Section 2.3.2, in our work we focus on validation techniques applied to conceptual schemas.

According to (IEEE 1998b), validation is the "confirmation by examination and provisions of objective evidence that the particular requirements for a specific intended use are fulfilled". In software development, "validation concerns the process of examining a product to determine conformity with user needs". In software engineering, all software artifacts (requirements specifications, design artifacts, implementation modules, etc.) should satisfy requirements and, consequently, they may be validated.

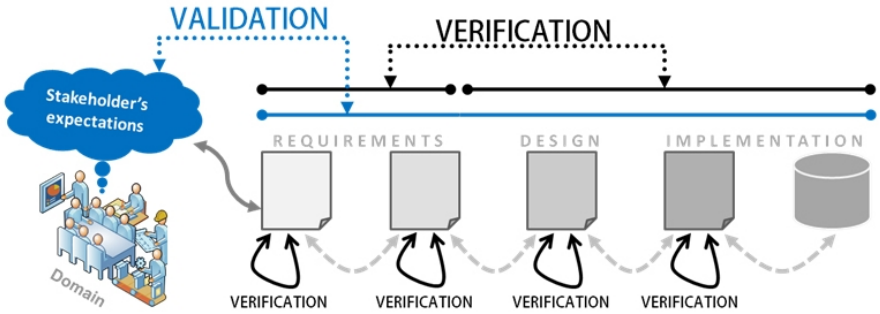


Fig. 12. Verification and Validation

Verification is the "confirmation by examination and provisions of objective evidence that specified requirements have been fulfilled" (IEEE 1998b). *Verification* asks the question "are the requirements right?" and *validation* asks the question "are the right requirements?" (Pohl 2010). As shown in Fig. 12, verification includes those techniques for checking internal defects (inconsistencies, syntax errors, unsatisfied correctness properties, etc.). Internal defects in software artifacts imply that the specified software requirements cannot be valid. However, the non-existence of internal defects does not imply that software requirements are valid (we also need to check that these are the "right requirements"). It is essential to note that, in contrast with verification, validation techniques need to check the expectations and needs of stakeholders.

In general, validation and verification techniques can be complementary used. Verification&Validation (V&V) is "the process of determining whether the requirements for a system are complete and correct and the final system complies with specified requirements" (IEEE 1990).

2.3.2 Conceptual Schema Validation in Requirements Engineering

Software systems are always validated by their users once the system is delivered. The use of the implemented system is, in fact, a validation-by-use process in which requirements defects may be identified directly by users.

However, since software artifacts are usually based on other artifacts created in previous development stages, validation can also be performed in artifacts prior to the implementation. If the validated artifact is a requirements specification, then the validation process is called *requirements validation*. More specifically, if the artifact under validation is a conceptual schema specification, then the process is called *conceptual schema validation*.

The contributions of this Thesis are aimed at enhancing conceptual schema validation in the context of requirements engineering. (Pohl 2010) affirms that requirements validation "require additional effort during requirements engineering. However, it also reduces the cost and the risks caused by requirements defects in later development phases".

According to (Van Lamsweerde 2009), requirements can be specified in different kinds of artifacts and in different levels of formalization (in unrestricted natural language, in disciplined documentation or in formal notation). The application of techniques aimed at validating requirements may depend on the formalization level of the requirements specifications. In particular, conceptual schemas defined in UML/OCL are formal specifications of functional requirements.

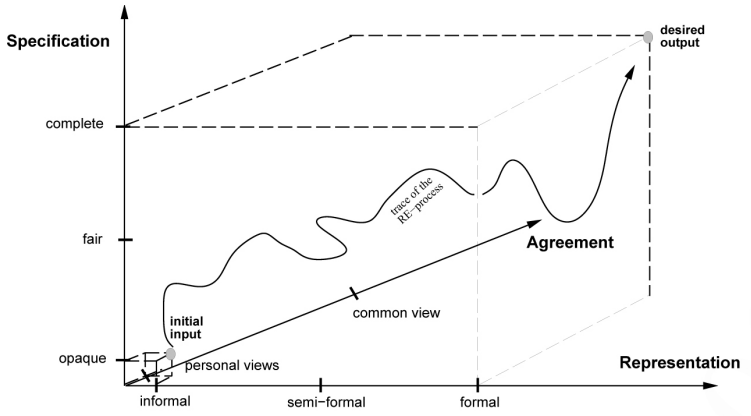


Fig. 13. Trace of a RE process (Pohl's quality framework)

(Pohl 2010) explains that a formal representation of the requirements is regarded as desirable in order to reach higher levels of quality (see Fig. 13). The main reason is that when using formal specifications of a conceptual schema, then more validation opportunities are feasible. In this Thesis, we propose using formally specified conceptual schemas (see Section 2.1) in order to support our validation approach.

Several techniques have been proposed in the literature in order to perform validation of conceptual schemas (see Section 3) as a quality control process to check those semantic quality properties explained in Section 2.2.1. However, conceptual schema validation is still a challenge open to new contributions, which would be aimed at enhancing the existing techniques.

We propose testing of explicitly and formally defined conceptual schemas (Chapter 4), in order to enhance the validation of *feasible completeness* and *validity* quality properties. Moreover, given that the quality of a conceptual schema should not be considered as an after-thought (it should be aimed for in each step of the conceptual modeling process), the second contribution of the Thesis is a conceptual modeling method based on continuous validation by testing (Chapter 8).

3

Related Work on Conceptual Schema Validation

In Chapter 2, we explained the general challenge of software validation and, in particular, the problem of conceptual schema validation in requirements engineering.

In this chapter, we review a representative set of existing software validation techniques. We focus on the validation of functional requirements specifications and, in particular, on the work related to conceptual schema validation, which is a fundamental challenge addressed by the main contributions of this Thesis.

Additionally, we also review some relevant validation techniques which are aimed at checking stakeholders' expectations on implementation artifacts, because some of their principles are related to our conceptual schema testing approach.

In Section 3.1, we present *prototyping* and *software testing* as popular validation techniques which have been traditionally applied to software implementations. Our approach brings the principles of code testing to conceptual modeling, which is performed in initial stages of the development.

In section 3.2, we analyze a set of representative validation techniques applicable to requirements specifications, focusing on those that can be used for conceptual schema validation.

3.1 Validation of Software Implementations

Programs that implement information systems consist of lines of code, which are written in a programming language. This code may contain different kinds of defects. On the one hand, code must conform to the programming language syntax. Language compilers usually include syntactic analysis in order to perform automatic detection of syntactic defects in the code. On the other hand, other kinds of execution errors can be detected while running a program. Nevertheless, even if the code "is right", it does not imply that this code implements "the right requirements". Therefore, code needs to be validated.

Several techniques have been proposed in the literature in order to verify and validate software implementations. Verification&Validation (V&V) of code is aimed at preventing runtime errors and checking if the implementation is according to the expected functionalities.

The main techniques to validate code are *testing* and *prototyping*. We review them in sections 3.1.1 and 3.1.2. In Section 3.1.3, we summarize *code inspections* and *reviews*, which are verification techniques that may contribute to code V&V.

3.1.1 Testing

Testing Techniques

Testing is commonly-used to verify and validate software. Most of the work in software testing assumes that the System Under Test (SUT) consists of programs that provide only a set of operations, and testing a SUT means calling those operations with appropriate context and input parameters, and checking that they return the expected outputs. For example, the recent UML Testing Profile (UTP) is based on this assumption (Object Management Group (OMG) 2005, Baker et al. 2008).

Based on the purpose and the scope of tests, testing techniques can be classified as follows (Myers et al. 2004, Beizer 2002):

- *Unit testing* is a verification technique focused on testing basic units of software. A basic unit is the smallest testable piece of software.
- *Integration testing* is a verification technique focused on testing sets of tested software units, which interact to constitute larger program structures.
- *System testing* is performed to test the functional (and some non-functional) requirements of the system. It is performed to test the end to

end quality of the entire system. If the requirements have been previously specified, then system testing becomes a verification technique to check that the implementation satisfies the specified requirements. In contrast, some agile methodologies do not assume the existence of previous requirements artifacts and the tests itself act as specifications of stakeholders expectations. In this case, system testing is a validation technique.

- *Acceptance testing* is a validation technique aimed at checking that the program satisfies the current needs and expectations of its end users. It is usually performed by customers and end users.

Testing techniques can also be classified into *black-box* and *white box* techniques. *Black-box* testing (also named functional testing) tests the specified functions of the system without knowing its internal implementation. White-box testing (also named structural testing) assumes that the internal workings of a program are known by the tester. White-box tests are conducted to assure that all internal components have been exercised (Beizer 2002).

Testing Languages

Tests programs written in a testing language allows the specification of test cases. A test case is a set of specified conditions under which a tester is able to determine whether an information system is working as expected or not.

Test cases can be written in the same language that is used to implement the software under test. However, tests may also be written in specialized testing languages. We focus on testing languages that allow *automated tests*. Test automation is a basic property of the Conceptual Schema Testing Language (CSTL) proposed in this Thesis.

Test automation is "the use of software to control the execution of tests, the comparison of actual outcomes to predicted outcomes, the setting up of test preconditions, and other test control and test reporting functions" (Janzen et al. 2005, British Computer Society 2009).

Several testing frameworks that allow test automation are known collectively as *xUnit* frameworks. *xUnit* frameworks include built-in constructs for the formalization of *test assertions*. *Test assertions* formally specify expectations that can be automatically checked. Assertions make tests automatically executable as many times as needed. Such frameworks are based on SUnit (Beck 1994), designed by Kent Beck and originally implemented for Smalltalk. The main ideas of SUnit have been ported to many programming languages and development platforms. Some

widely-used examples (Gamma et al. 1999, Ostroff et al. 2005) are JUnit (for testing programs written in Java), CppUnit (for C++), NUnit (for .NET), XMLUnit (for XML specifications), E-Tester (for Eiffel), etc.

```
public class TestProgram extends TestCase
{
    public void testAddition (){
            assertEquals(3+2, 5);
    }
}
```

Fig. 14. Example of a simple JUnit test case

Other research works have been devoted to specifying validation test cases in more understandable forms for domain stakeholders. *Fit* is a representative framework that promotes the use of *Fit tables* to express storytests (Mugridge 2008). *Fit tables* specify user expectations in a form which is assumed to be more understandable for business people than test cases written in a programming language. *Fit* was proposed by *Ward Cunningham* (Mugridge et al. 2005) for programs developed in Java. Since 2005, versions for Java, C#, Python, Perl, PHP and Smalltalk have been developed. Several tools to support *Fit tables* have also been proposed (the most popular is *Fitness* (Myers et al. 2004, Mugridge 2008, Murphy-Hill et al. 2008, Martin et al. 2008)). These tables are executed on the software implementation in order to check for compliance.

Discount Availability	
type	price
full price	$95.00
student	$80.00
senior	$80.00
child	$32.00 *expected* $31.99 *actual*

Fig. 15. Example of a Fit table checked by the Fitness tool

MDD and Software Testing

In the context of Model-Driven Development (Mellor et al. 2002), there has been some work that relates testing to modeling. (Hierons et al. 2009) provides a thorough survey on the state of the art regarding the many relationships between formal specifications and testing. One such relationship which attracted a lot of work is that specifications can be used for deriving, automatically, test cases of the final system and/or of some intermediate artifact (Pickin et al. 2002, Gargantini et al. 1999, Nebut et al. 2006, Santos Neto et al. 2005, Hartman et al. 2004, Briand et al. 2001, Zhang 2004, Pilskalns et al. 2007, Javed et al. 2007).

It is important to note that, in these verification approaches, the artifact under test is software code. Moreover, the source models are assumed to be valid. These approaches contrast with those explained in Section 3.2, in which the artifact under test is a requirements specification or a system model. Our testing approach (Chapter 4) assumes that the artifact under test is an executable conceptual schema. In contexts in which the design and the implementation of the system are not automatically obtained and correct-by-construction from the conceptual schema, then the reviewed testing generation approaches (Model-Driven Testing) could support the automatic generation of design and implementation test cases from those specified when testing the conceptual schema by using our approach.

Testing Strategies

A testing strategy's most valuable property is producing failures revealing faults as fast as possible (Meyer 2008). Not all possible tests equally contribute to this objective. Therefore, it is necessary to plan a test strategy, according to the testing objectives and available resources.

(Goodenough et al. 1975) pointed out in the 70s that a central question in software testing was analyzing what constitutes an adequate test. This analysis may be performed by means of defining adequacy test criteria.

A great number of such criteria have been proposed and investigated. Test adequacy criteria allow measuring the coverage of a set of tests, i.e. which parts of the code are not exercised enough. (Zhu et al. 1997) surveys the research work on test coverage metrics and test adequacy criteria.

A large variety of test adequacy criteria have been proposed in the literature. For example, the statement coverage criterion requires that each statement of the program is executed at least once by a test set. Statement coverage is neutral to control structures. Other criteria have been proposed, such as decision coverage (requires each boolean expression tested in control structures to be evaluated to both true and false), condition coverage (partitions each boolean sub expression and requires to be evaluated to both true and false), etc. (Whalen et al. 2006) proposes a set of adequacy criteria for software testing, based on requirements formalized in logics. Several tools have been developed to measure coverage (Yang et al. 2007).

Each criterion implies a different level of testing and, therefore, a different testing effort. Defining the testing strategy requires choosing the most adequate testing criterion, or a combination of them.

In Chapter 11, we complement the conceptual schema testing approach described in Chapter 4 with a set of basic adequacy criteria for testing conceptual schemas.

3.1.2 Prototyping

(Vonk 1990) defines a *prototype* as a "working model of (parts of) an information system, which emphasizes specific aspects of that system". Prototypes can be used to elicit and validate requirements (Beynon-Davies et al. 1999).

Prototypes may implement in detail some particular functionalities of a system (vertical prototyping) or consider a wider range of functionalities without entirely developing them (horizontal prototyping) (Beynon-Davies et al. 1999). In horizontal prototyping, it is common to use prototypes that only implement the user interface. Either vertical and horizontal prototyping (or a combination of them) can be used and evaluated by stakeholders. By this way, prototypes promote software validation and contribute to the elicitation of unrevealed requirements. Prototypes are especially useful to elicit and validate requirements for innovative software projects in which goals are not clear enough (Doke et al. 1995).

Prototypes are based on a partial implementation of the system, and their validation requires manual experimentation and observation. In contrast, conceptual schema validation by testing, as proposed in this Thesis, simulates the execution of the conceptual schema. Prototyping may be used in conjunction with conceptual schema testing, since prototypes may be automatically generated from conceptual schemas (Pastor et al. 2007).

3.1.3 Code inspections and reviews

Code *inspections* and *reviews* are verification techniques which are applied to code that implements an information system.

Inspections and reviews may be used cooperatively in an *inspection&review* process. Depending on the configuration of the *inspection&review* process, these techniques may also contribute to software code validation. Inspections and reviews consist of analyzing the code without executing them: "For many years, most of us in the programming community worked under the assumption that programs are written solely for machine execution and are not intended for people to read, and that the only way to test a program is to execute it on a machine. This attitude began to change in the early 1970s through the efforts of program developers who first saw the value in reading code as part of a comprehensive testing." (Myers et al. 2004).

Inspections and reviews are known to be quite effective for source code (Van Lamsweerde 2009). Code inspections may be more or less structured. *Walkthroughs* are internal inspections involving members of the project team. *Walkthroughs* are the simplest and less structured verification approach to find defects in code. The

inspection process can be made more formal through having external reviewers, meetings with specific agendas, and inspection reports (Van Lamsweerde 2009, Pohl 2010). *Fagan* proposed in 1976 a well-known example of a structured inspection & review process (Fagan 1976, Fagan 1986). *Reviews* may be performed in free mode; based on defect-based, quality-specific or domain-specific checklists (Jaffe et al. 1991, Lutz 1993, Sommerville et al. 1998); or based on a specific process to follow for defect search (Porter et al. 1995). If the *Inspection&Review* process pursues domain-specific defects, then it also contributes to validation.

3.2 Requirements Validation

In Section 3.1, we have seen that software code may be validated in order to check if it implements the expected requirements. Nevertheless, in software engineering, requirements are usually elicited and specified before implementing them. Therefore, requirements can be validated earlier in the development. In this section, we analyze how requirements specifications can be validated.

According to (Van Lamsweerde 2009), requirements can be specified in different kinds of artifacts and in different levels of formalization (in unrestricted natural language, in disciplined documentation or in formal notation). The application of techniques aimed at validating requirements may depend on the formalization level of its specification. In particular, conceptual schemas defined in UML/OCL are formal specifications of functional requirements. Its validation is the main objective of the conceptual schema testing approach and the Test-Driven Conceptual Modeling method proposed in this document (Sections 4 and 8).

In the following, we review requirements validation techniques that may be applicable to conceptual schemas (sections 3.2.1 to 3.2.6). Some of them are general techniques that can be applied to other kinds of requirements specifications. Others have been specifically proposed to validate conceptual schemas. Conceptual schema testing belongs to the category of *simulation & animation* reviewed in Section 3.2.6. In Section 3.2.7, we briefly review a representative set of verification approaches which can be used in conjunction with validation techniques to enforce the V&V process, as explained in Section 2.3.1.

3.2.1 Inspections and Reviews

In Section 3.1.3, we have seen that, since 70s, inspections and reviews are used for code *Verification&Validation*. In requirements engineering, most of the requirements specifications can also be inspected and reviewed.

Some studies in the literature reveal the effectiveness of inspections and reviews when applied to requirements documents, in terms of high defect detection rates, quality benefits and cost savings (Doolan 1992). *Fagan*'s inspection & review process (originally designed for code inspection) can also be applied to requirements specifications (Van Lamsweerde 2009). If the process checks for specific defects related to domain needs and expectations, then it contributes to validation. The process consists of an iterative cycle of four activities:

- *Inspection planning:* In this phase, the members of the inspection team, the timing of the process, the schedule and scope of the review meetings and the format of the inspection documentation are determined.
- *Individual reviews:* Each inspector analyzes the requirements specifications or parts of it individually to look for defects. Reviews can be based on checklists or specific defect-search processes.
- *Defect evaluation at review meetings:* The defects found by each inspector are collected and discussed by the meeting participants in order to reach an agreement.
- *Requirements document consolidation:* The requirements document is revised to address all concerns revealed in the previous phases.

Inspections do not require formal requirements specifications. However, when semi-formal or formal specifications are the requirements artifacts under inspection, the process may be more clear, structured and traceable.

Templates can be used to organize the elicitation and specification of requirements. The *IEEE Std-830* template (IEEE 1984) and the *Volere* template (Robertson et al. 1997) are well known examples of requirements templates, which are structured documents to be inspected and reviewed.

Specific kinds of requirements can be specified in other types of structured specifications. Functional requirements, for example, can be informally described as use cases (Cockburn 1999). Scenarios are concrete instances of use cases, which "are captured as text narratives, sketches and informal media. As analysis progresses, informal representations may be replaced by models" (Sutcliffe 1998). Software development methodologies, such as the popular Unified Process (Larman 2005) include models to specify scenarios. This fact lays the ground for scenario-based inspections (Leite et al. 2005, Leite et al. 1997, Regnell et al. 2000, Regnell et al. 1995), which are an adaptation of the *Fagan*'s process aimed to inspect scenario models.

Conceptual schemas specify functional requirements and they can also be inspected and reviewed. This is why understandability is a fundamental property of

conceptual schema representations (see Section 2.2.2). If conceptual schemas are formally defined, some defects that may be pursued by inspections and reviews can be automatically detected, as explained in the following.

3.2.2 Validation by Reasoning

(Grüninger et al. 1995) state that conceptual schemas must be able to answer competency questions, which are requirements specified in the form of questions.

(Queralt 2009, Queralt et al. 2009, Queralt et al. 2008) propose to translate conceptual schemas into logic. Then, user-defined properties can be specified as questions, which are formalized as logical predicates. This approach reasons on the logical translation of the schema, trying to construct a state of the Information Base (IB) in which the specified property holds.

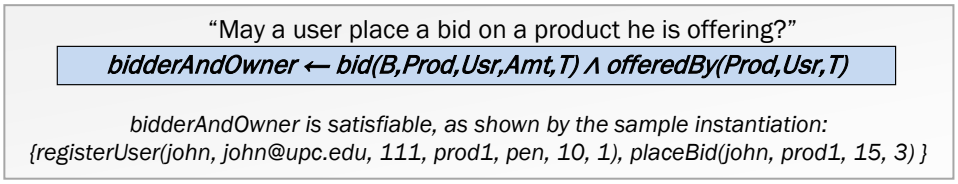


Fig. 16. Example of a user-defined question and the provided feedback

It is well known that the problem of reasoning with integrity constraints and derivation rules in its full generality is undecidable. Therefore, the available procedures are restricted to certain kinds of constraints and derivation rules or domains, or they may not terminate in some circumstances (Queralt 2009).

In contrast, the conceptual schema testing technique that we propose in this Thesis can be applied to conceptual schemas that include any kind of OCL constraint or derivation rule and, if there is not an infinite loop in a derivation rule or method, test cases usually terminate in a very short time. Moreover, tests are significantly different from property questions: tests are formalizations of concrete scenarios (that is instantiations of the conceptual schema) and questions are formalizations of properties. The results obtained by testing are not as strong as those obtained by automated reasoning procedures when they are applicable. Like it has been observed in the general field of software (Hierons et al. 2009), testing and automated reasoning procedures are complementary and can be used in conjunction in order to enhance validation.

Many reasoning techniques have been also proposed in order to verify general properties of conceptual schemas. We briefly reference them in Section 3.2.7. These techniques can also be used in combination with validation techniques in order to contribute to the whole V&V process.

3.2.3 Paraphrasing

The acquisition of requirements is achieved through language manipulation (communication with stakeholders). However, it is usually convenient to specify these requirements in models, such as conceptual schemas. As we have seen, some V&V techniques require semi-formal or formal models to be applied. Moreover, models are usually used for specification purposes and as a base for design and implementation (including automatic generation of code).

Several works propose the use of *paraphrasing* in natural language for making easier the validation of conceptual schemas. According to (Rolland et al. 1992), domain analysts "are able to correctly use concepts of a conceptual schema but have difficulties to abstract reality in order to represent it through concepts". In order to address this issue, paraphrasing techniques generate accurate natural language descriptions of a conceptual schema, which can be easily understood by domain people who performs validation.

Paraphrasing is usually manually performed by system engineers when communicating with stakeholders. However, (Dalianis 1992) states that "it would be convenient if it could be carried out automatic, so the domain expert himself could validate the requirements model without having deeper knowledge in the formalism".

Many researches focused their work on paraphrasing techniques in the 90s (Rolland et al. 1992, Dalianis 1992). Nowadays, some authors point out paraphrasing challenges, arguing that "the main techniques proposed for semantic matching are ontology based. […] Numerous proposals can be found. However, not any one is really convincing" (Métais 2002).

Some recent work proposes paraphrasing in structured languages. (Cabot et al. 2010) proposes an approach for automatically paraphrasing UML/OCL conceptual schemas to SBVR to facilitate its comprehension and validation. SBVR (Semantics of Business Vocabulary and Business Rules) is a language conceptualized optimally for business people, and it is assumed to be better understood for domain people than UML/OCL. There also exist approaches to generate abstracts of model specifications (such as documentation in natural language, user manuals, etc.) (Jesus et al. 1992).

Paraphrasing efforts are closely related to many attempts to verbalize facts from the universe of discourse in order to automatically support the conceptual modeling activity. The main purpose of these research efforts is deriving a conceptual schema from a set of sample sentences verbalized in a language that is better understandable by the domain experts.

Some work has been done in this direction for Object-Role Modeling (ORM) (Halpin et al. 2008, Halpin 2001). Object-Role Modeling (ORM) is a method for modeling and querying an information system at the conceptual level. ORM diagrams represent the domain as objects (entity types), the relationships (fact types) between them, the roles that the objects play in those relationships and constraints within the problem domain. (ter Hofstede et al. 1997) proposes techniques in order to elaborate verbalization of sample facts from the universe of discourse, in order to be the input to derive ORM representations. Moreover, these verbalizations also provide means to paraphrase the resulting model in order to be validated by domain experts.

3.2.4 Explanation Generation

Explanation generation is another research direction to contribute to the validation of conceptual schemas. (Gulla et al. 1993) describes that "explanation generation components can form a question-answer facility, where explanations are user tailored and may include graphical model views in addition to the textual descriptions".

Explanation generation techniques provide explanations about the behavior of conceptual schemas from an explanation request. In contrast with paraphrasing, explanation generation techniques usually take into account states of the Information Base (IB).

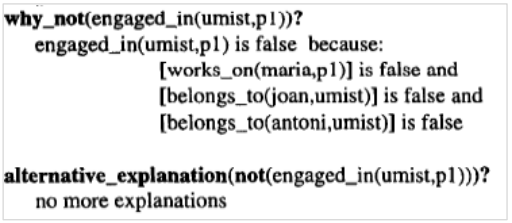


Fig. 17. A question-answer example

(Olivé et al. 1996) propose an explanation generation framework in which domain people ask catalogued questions to the system and analyzes if the provided explanations satisfy their expectations. This work provides explanations about the contents of the information base (why fact?, why_not fact?), the changes to the IB (why_inserted fact?, why_deleted fact?, how_not fact?), the historical evolution of the IB (why fact at state?, why_not fact at state?, what_known_about fact type at state?, when fact?, when update?) and about hypothetical past updates (what_if update at state on fact?, what_if_not update at state on fact?).

3.2.5 Constraint Acquisition

Requirements validation consists in ensuring that the system satisfies stakeholders' expectations. It is important to note that these expectations are not only about what the system is able to perform, but also about what invalid states the developed system should prevent. Invalid states are those that violate the constraints defined in a conceptual schema.

Constraint acquisition is an approach specifically aimed to assist the elicitation and validation of constraints in Entity-Relationship models. (Hartmann et al. 2009) defines this technique as "an iterative process of inspection: some participant suggests the significance of some integrity constraint φ for the application domain (i.e., φ becomes a candidate constraint that might be specified)." Then, object sets for the model related to the candidate constraint are generated. Object sets satisfy Armstrong database principles (Armstrong 1974). A database instance is defined to be an Armstrong database for a constraint set Σ, if the database instance satisfies an integrity constraint φ (the candidate constraint), precisely when φ is implied by Σ. If no object sets can be generated, then the candidate constraint may be redundant. Otherwise, if an object set is acceptable, then the constraint should be discarded. If all generated object sets are not acceptable, then the constraint can be added to the constraints set of the model.

3.2.6 Simulation & Animation

Formal requirements specifications (like conceptual schemas defined in a formal modeling language) can be validated by using testing techniques that execute them through animation (Mellor et al. 2002, Zhang 2004, Ostroff et al. 2007). In general, animation facilities allow users to execute operations of the specification with user supplied parameters, thereby calculating the value of the output parameters and the new system state (Bicarregui et al. 1997). The method we propose to test conceptual schemas belongs to this category of validation techniques.

The idea of animating conceptual schemas for testing purposes dates back to the mid-80's. (Dignum et al. 1987) describes a conceptual language (CPL) and a tool that generates a prototype from a CPL schema, which can be tested. The generated prototype makes it possible to build an IB state, perform consistency checks and ask questions about the contents of the IB. A similar approach was taken in (Lindland et al. 1993) and (Gogolla et al. 1995), with the PPP and TROLL light language and environment, respectively. Others, such as (Trong et al. 2005) are focused on animating design models.

The most recent work, and the one most closely related to our approach, is the USE tool (Gogolla et al. 2007, Gogolla et al. 2005, Richters et al. 2000). USE (Fig. 18) makes it possible to define snapshots (instantiations) of structural schemas expressed in a subset of UML/OCL and checking if they are valid instantiations of the schema. (Seybold et al. 2006) is a similar approach focused on behavior, which allows simulation of scenarios (specified as message sequence chars) of use cases modeled in the ADORA language.

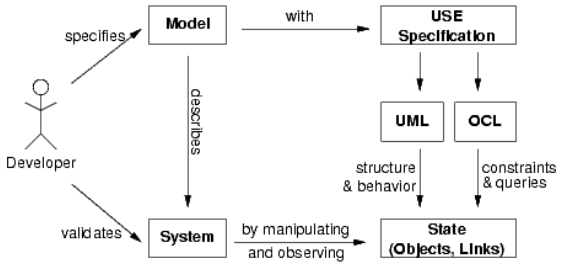


Fig. 18. USE validation framework

The information processor that allows the execution of conceptual schemas in our tool (Chapter 5) is based on the USE core, although it has been extended to deal with additional language features (taking into account advanced structural schema elements and the behavioral schema).

In contrast with the USE tool, in this Thesis we specify conceptual schema tests in an xUnit-styled conceptual schema testing language (Section 4.4). This language allows specifying assertions (formalizations of expectations) that make possible to run automated tests as many times as needed. By this way, the modeler is alerted only in case of failing assertions. Moreover, in this Thesis we present the TDCM method (Section 8) in order to use these tests to drive the conceptual modeling activity and a complete environment to support the application of the method in practice.

3.2.7 Conceptual Schema Verification Techniques

Although the contributions of this Thesis are related to conceptual schema validation, it is important to remark that formal requirements specifications (such as conceptual schemas) also admit verification checks which are usually automated by tools (Van Lamsweerde 2009). Our conceptual schema testing approach (Chapter 4) and our Test-Driven Conceptual Modeling method (Chapter 8) are validation techniques that can be used in conjunction with existing procedures aimed to verify conceptual schemas.

There has been a lot of work on automatic verification procedures applicable to conceptual schemas. A representative set of recent papers is (Pilskalns et al. 2007, Pilskalns et al. 2007, Queralt et al. 2009, Queralt et al. 2008, Formica 2003, Berardi et al. 2005, Jarrar 2007, Cabot et al. 2008, Kalyanpur et al. 2005, Glinz 2000)). Their general objective is to automatically check that a conceptual schema has a set of properties. Some of these properties are general and must be satisfied by all conceptual schemas, while others apply only to a particular conceptual schema. The most studied general properties are integrity constraints satisfiability, liveliness of an entity or relationship type, non-redundancy of integrity constraints and operation executability.

4

An Approach to Test Conceptual Schemas

In this chapter, we present the first main contribution of this Thesis: an approach to test conceptual schemas. The fundamental aspects of this contribution are published in (Tort et al. 2010).

This chapter presents the fundamentals of conceptual schema testing (Section 4.1), the set of elementary test kinds for testing conceptual schemas (Section 4.2), and an overview of the proposed testing environment (Section 4.3). After that, we present the Conceptual Schema Testing Language (CSTL), a specialized language for writing tests of executable conceptual schemas defined in UML/OCL (Section 4.4).

4.1 Fundamentals of Conceptual Schema Testing

As we have seen in Section 2.1, a conceptual schema defines the general knowledge that an information system needs to know in order to perform its functions. Conceptual modeling is the activity aimed to elicit, specify and validate conceptual schemas of information systems.

According to well-known conceptual modeling quality frameworks (Section 2.2), correctness (i.e. the knowledge defined in the schema is true for the domain) and completeness (i.e. all relevant knowledge is defined) are fundamental semantic quality properties of conceptual schemas. The validation of these properties is a key challenge as explained in Section 2.3.

The main objective of our approach is checking these properties by executing and checking concrete user stories formally defined as test cases. These test cases may be collected and automatically executed as many times as needed. Our approach to test conceptual schemas promotes continuous refinement and evolution of the schemas under test, with the aim of pursuing higher levels of semantic quality.

The presented approach can be used in conjunction with the existing validation and verification techniques reviewed in Chapter 3. In conjunction with an associated test processor capable to interpret and execute CSTL test cases (described in Chapter 5), our approach allows the automation of tests (Janzen et al. 2005) of conceptual schemas, that is, the use of software to control the execution of tests, the comparison of actual outcomes to predicted outcomes, the setting up of test preconditions, and other test control and test reporting functions (British Computer Society 2009).

Our approach to test conceptual schemas is based on three main observations:

1. The validation of the functional requirements defined in a conceptual schema is a challenge that admits new contributions (Section 3.2). These contributions should be aimed at achieving higher semantic quality levels.

2. In the programming field, testing is a widely-used technique in order to address the validation of software implementations (Section 3.2). Since (1) conceptual schemas may be executable and (2) software testing is being successfully used in programming, we could adopt many basic principles of software testing in conceptual schema testing.

3. In contrast with code testing, in which the artifact under test is a system implementation, in conceptual schema testing the conceptual schema specification is the artifact under test. Therefore, the objective of conceptual schema testing is to early reveal requirements errors that may be detected at the conceptual modeling stage.

Validation of Conceptual Schemas: An Open Challenge

As we have seen in the state of the art on conceptual schema validation presented in Section 2.2, several existing techniques are used to achieve correct and complete conceptual schemas in different degrees. These include paraphrasing in natural language (Rolland et al. 1992, Dalianis 1992, Métais 2002, Frederiks et al. 2006), generation of abstractions and abstracts (Jesus et al. 1992), explanation generation (Gulla et al. 1993, Olivé et al. 1996, Gulla 1996), constraint acquisition (Hartmann et al. 2009) or simulation and animation (Van Lamsweerde 2009, Lindland et al. 1994, Dignum et al. 1987, Gogolla et al. 1995, Gogolla et al. 2007).

The first rationale of conceptual schema testing is that conceptual schema validation is a research area that admits new methods and techniques, with the purpose of improving semantic schema quality assurance.

The approach to conceptual schema testing described in this chapter belongs to the category of simulation and animation, and it can be used in conjunction with other existing techniques (see Chapter 3).

Application of Software Testing Principles in Conceptual Modeling

In professional practice, software testing is a dominant technique which is widely used for validating implementations of information systems (Glass 2008, Myers et al. 2004, Beizer 2002).

However, the development of information systems in most software engineering methods is the result of the evolution of different artifacts. In particular, in the OMG's Model-Driven Development (MDD) approach (Mellor et al. 2002, Pastor et al. 2007), software development is viewed as an evolution of models, from platform-independent models (PIM) to platform-specific models (PSM). A Conceptual schema is a PIM that defines the functions that the systems under development need to perform and the required structural knowledge about the domain required by these functions.

Since a conceptual schema is a formal specification of the functional requirements of a system, it is possible to develop techniques for the early detection of errors in software developments and, in particular, in conceptual modeling.

Moreover, conceptual schemas specified in a formal language can be executable (Mellor et al. 2002, Insfrán et al. 2002, Olivé 2005). Executability may be achieved by an automatic and complete transformation of the conceptual schema into software components (including the database schema) written in the languages required by the production environment, or by the use of a virtual machine that interprets the modeling language and simulates the execution of the schema.

The second rationale of our approach is that if conceptual schemas can be executable, they are, in fact, software artifacts and, therefore, they can be tested. Additionally, conceptual schemas may contain requirements errors, and these errors can be early detected in the schema. Then, it seems feasible to define a testing environment in order to enhance the early validation of the functional requirements defined in conceptual schemas. This is the main purpose of the approach for testing conceptual schemas presented in this chapter.

The third rationale of our approach is that if testing is a widely used technique in programming, some basic principles and techniques which have been successfully applied in software testing can be adapted to conceptual modeling. In this sense, our approach applies modern techniques developed in the software testing field to conceptual schemas. We adopt the definition of testing proposed by Meyer: "To test a program is to try to make it fail" from which the goal of testing becomes "to uncover faults by triggering failures" (Meyer 2008).

If a conceptual schema were like an ordinary program, then its testing would not be remarkably different from testing a program. However, a conceptual schema is knowledge or, more precisely, it is the general knowledge that an information system needs to know about the domain and about the functions it has to perform (Olivé 2007).

An executable conceptual schema can be considered a program only when there is a general-purpose information processor (virtual machine) able to behave according to the structural and behavioral rules defined in the conceptual schema (International Standards Organization (ISO) 1982). Consequently, we may find some similarities between testing a program and testing a conceptual schema, but there are significant differences. Most of the work in software testing assumes that the system under test (SUT) consists of programs (objects, components) that provide only a set of operations, and testing a SUT means calling those operations with appropriate context and input parameters, and checking that they return the expected outputs. For example, the recent UML Testing Profile (UTP) is based on this assumption (Object Management Group (OMG) 2005, Baker et al. 2008) and the same happens in popular testing frameworks like JUnit. However, in contrast with lines of code, conceptual schemas explicitly define concepts, relationships between these concepts, integrity constraints, derived information, domain events, queries, etc.

Therefore, it is necessary to clarify what it means to test conceptual schemas by means of which kinds of tests are applicable. Section 4.2 defines the five kinds of tests that can be applied to conceptual schemas regardless of the conceptual modeling language.

Testing Executable Conceptual Schemas

Testing conceptual schemas is as important as testing programs in projects that follow the OMG's Model-Driven Development (MDD) approach (Pastor et al. 2007, Object Management Group (OMG) 2003) when the transformation from platform-independent models (PIM) to platform-specific models (PSM) is fully automatic and correct-by-construction. Our PIMs are complete conceptual schemas that include all structural and behavioral aspects, and the obtained PSMs include all aspects

related to database design, integrity constraints enforcement procedures, derivation rules procedures and the operations that correspond to the behavioral aspects. In the MDD approach, the testing activity is done at the conceptual schema level and, given that the transformation is correct-by-construction, there is no need to test again the outputs from that transformation.

Testing conceptual schemas is also beneficial in those projects in which the quality (Lindland et al. 1994, Gemino et al. 2005, Genero et al. 2008, Schewe et al. 2005, Burton-Jones et al. 2005) of the conceptual schema must be maximal, even if the transformation to the PSMs will be manual and, therefore, there will be the need to test again those PSMs. In particular, achieving maximal quality degrees is an important objective when the conceptual schema is the basis for the development of several information systems to be done by independent project teams working in parallel, when the conceptual schema is going to be used as a reference model for several applications, or when the conceptual schema is a metaschema to be instantiated in several applications. Testing conceptual schemas may also be useful in conceptual modeling education as a means for students to check that their conceptual schemas do in fact represent the knowledge they intend to define.

Even if we deal with complete conceptual schemas, we think that it makes sense to test incomplete conceptual schemas while they are developed, as a means of increasing their quality (Section 2.2). Even small fragments consisting of a few entity and relationship types, integrity constraints and derivation rules can be tested in order to uncover faults during the early stages of the development of a conceptual schema. This fact lays the groundwork for the development of a test-driven conceptual modeling methodology, based on the popular Test-Driven Development (Beck 2003), which is the second main contribution of this Thesis (see Chapter 8).

4.2 Test Kinds

A test case of a conceptual schema is a formalization of a concrete user story. This story is expected to be successfully executed if the required knowledge is correctly defined in the schema.

In conceptual modeling, (a fragment of) the lifetime of an information system is a sequence of Information Base (IB) states. The IB represents a snapshot of the state of the domain as an instance of the conceptual schema. In our approach, we conceive test cases for testing conceptual schemas as a sequence of states of the IB, together with formalized expectations about these states. IB states may be directly created, or they may be achieved by means of the occurrence of events.

In order to state what it means to test conceptual schemas, we define a complete set of *test kinds* that may be applied to conceptual schemas. *Test kinds* define which general types of expectations are needed to specify test cases for testing conceptual schemas.

Conceptual schemas define the functional requirements of an information system. Therefore, the definition of *test kinds* is based on the general functions that characterize any information system. According to (Boman et al. 1997), an information system performs three general kinds of functions: memory (maintains a representation of the state of the domain), informative (provides information about the state of the domain) and active (performs actions that change the state of the domain). Based on these functions, we determine which kinds of tests are needed to formalize expectations about the knowledge defined in a conceptual schema.

The result is a list of five *test kinds* that can be applied to conceptual schemas. We define these test kinds regardless the conceptual modeling language. These kinds of tests are the assertions of (1) the consistency of the Information Base (IB); (2) the inconsistency of the IB; (3) the occurrence of a domain event; (4) the non-occurrence of a domain event, and (5) the contents of an IB state. Sections 4.2.1, 4.2.2, 4.2.3, 4.2.4 and 4.2.5 describe the enumerated *test kinds*. In Section 4.4, we present the Conceptual Schema Testing Language (CSTL), which supports the specification of test cases for UML/OCL Conceptual Schemas, based on these general test kinds.

4.2.1 Asserting the Consistency of an IB State

The objective of the memory function of an information system is to maintain an internal representation of the state of the domain. This representation is needed by the other functions of the system and changes over time (since the information in the domain it represents also changes).

Usually, not all IB states are expected to be valid. Determining the IB states that are considered to be valid in the system is a key activity in requirements engineering. The set of integrity constraints of the conceptual schema restricts the set of valid IB states. An IB state which satisfies the defined integrity constraints is a consistent IB state.

This is the rational for the first test kind, which corresponds to the assertion of the consistency of an IB state reached by a test case. If the assertion is true, then no constraints prevent the IB state as expected (the state is consistent). Otherwise, some constraints of the conceptual schema are too restrictive.

4.2.2 Asserting the Inconsistency of an IB State

Determining the IB states that are considered to be invalid in the system is also a key activity in requirements engineering. The set of integrity constraints of the conceptual schema determines the set of invalid IB states. An IB state that does not satisfy some of the defined integrity constraints is an inconsistent IB state.

This is the rational for the second test kind, which corresponds to the assertion of the inconsistency of an IB state reached by a test case. If the assertion is true, then the conceptual schema include constraints to prevent the IB state as expected (the IB state is inconsistent). Otherwise, the set of constraints need to be more restrictive in order to prevent the asserted state as expected.

4.2.3 Asserting the Contents of an IB State

The objective of the informative function of an information system is to provide users with information about the state of the domain.

Determining if the knowledge defined in a conceptual schema is aligned to the expectations about its correctness is a critical activity in conceptual modeling. The objective of this test kind is checking that, in a concrete IB state (explicitly created or achieved by means of the occurrence of a set of events) the value of basic and derived knowledge defined in the schema is as expected.

This is the rational for the third test kind, which corresponds to the general assertion of the contents of an IB state reached by a test case. If the assertion is true, then the conceptual schema has the correct knowledge to provide information about the IB state as expected. Otherwise, the knowledge defined in the conceptual schema needs to be changed (the specification of the derived knowledge or the specification of some events is incorrect).

This test kind is the one more closely related to the kinds of tests used in programming. It is important to note that in code testing, tests are usually limited to check if the obtained results (after the execution of a set of operations) are those which are expected (Object Management Group (OMG) 2005, Baker et al. 2008).

4.2.4 Asserting the Occurrence of a Domain Event

With the active function, an information system performs actions that modify the state of the domain. In this context, conceptual schemas that define behavioral knowledge include domain events that specify changes in the state of the domain.

Usually, not all changes in the state of the domain are considered to be valid. In other words, not all domain events are expected to be allowed to occur in all situations. Therefore, determining which occurrences of domain events are valid in the system is a valuable activity in requirements engineering and, in particular, in conceptual modeling. The set of integrity constraints that are associated to the domain event restricts the states in which an event is allowed to occur. The occurrence of a domain event in a concrete IB state comprises checking (1) that the occurrence of the event is allowed by its constraints, and (2) checking that the IB state achieved by the occurrence of the event is according to the specification of the domain event.

This is the rational for the fourth test kind, which corresponds to the assertion of the occurrence of a domain event in an IB state reached by a test case. If the assertion is true, then the domain event has occurred as expected and the resultant IB state complies with its specification. Otherwise, some constraint prevents the domain event to occur, or the specification of the domain event is not correct.

4.2.5 Asserting the Non-Occurrence of a Domain Event

Determining the occurrences of domain events that should not be allowed by the system is also crucial for the conceptual modeling activity. The set of integrity constraints that are associated to the domain event prevent the states in which an event is not allowed to occur.

This is the rational for the fifth test kind, which corresponds to the assertion of the non-occurrence of a domain event. If the assertion is true, then the domain event has been not allowed to occur as expected. Otherwise, the set of constraints related to the event need to be modified in order to prevent its occurrence.

4.3 The Testing Environment

In order to perform conceptual schema testing, our approach is based on the testing environment (Tort et al. 2010) shown in the schema of Fig. 19.

In our approach to test conceptual schemas, conceptual modelers define an explicit specification of the conceptual schema of the information system under development and a collection of automated tests aimed to test the schema.

A formal language to define the conceptual schema and a formal language to define the test programs are required to make this approach applicable in practice. In this Thesis, we test conceptual schemas defined in UML and OCL modeling languages

(the corresponding concepts and notation are explained in detail in Section 2.1.1). One of the key contributions of this Thesis is the definition of a specialized language to specify test cases of conceptual schemas. This language, named Conceptual Schema Testing Language (CSTL), is presented in detail in Section 4.4. The CSTL language allows formally specifying the test kinds for testing conceptual schemas defined in Section 4.2.

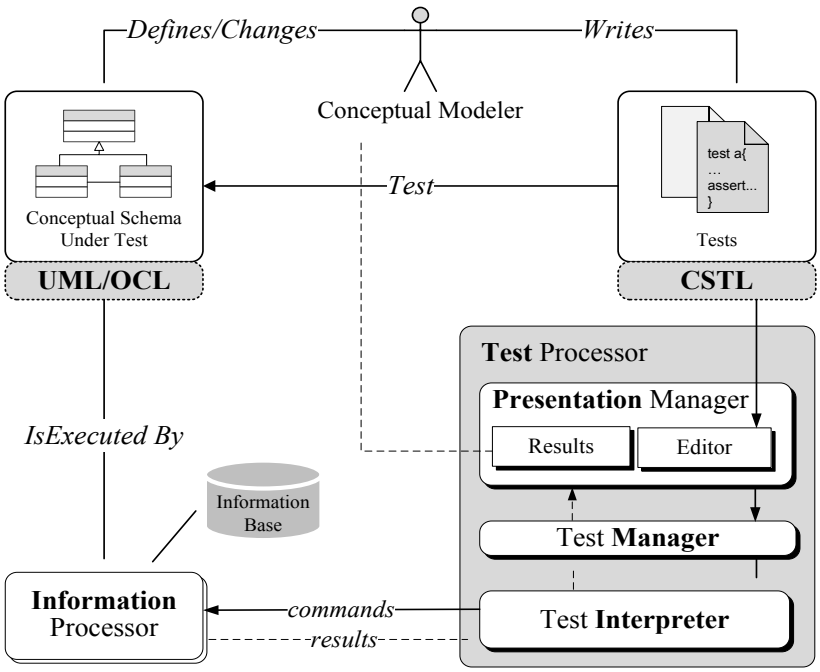


Fig. 19. Testing environment to test conceptual schemas

This approach also requires conceptual schemas to be executed by simulating the states of the IB base and its evolution. The execution of conceptual schemas is done by an *information processor*, while the execution of test programs is done by a *test processor*, which interacts with the information processor in order to modify and query the Information Base (IB) according to the test statements. Specific implementations of the *information processor* and the *test processor* are the main components of the *CSTL Processor*, a prototype tool that supports the creation, management, execution and computation of the verdicts of test cases defined in CSTL for testing UML/OCL conceptual schemas. More details on the features and the implementation of the *CSTL Processor* are described in Chapter 5.

In order to test conceptual schemas, the system's functions are captured by a set of concrete scenarios formalized as test cases. If the execution of these test cases produces the expected results, then, by definition, we can ascertain that the conceptual schema is complete according to the needs and expectations formalized in the test cases. If the conceptual schema were not complete, then some test case would not succeed because of required knowledge that is relevant but not defined in the schema.

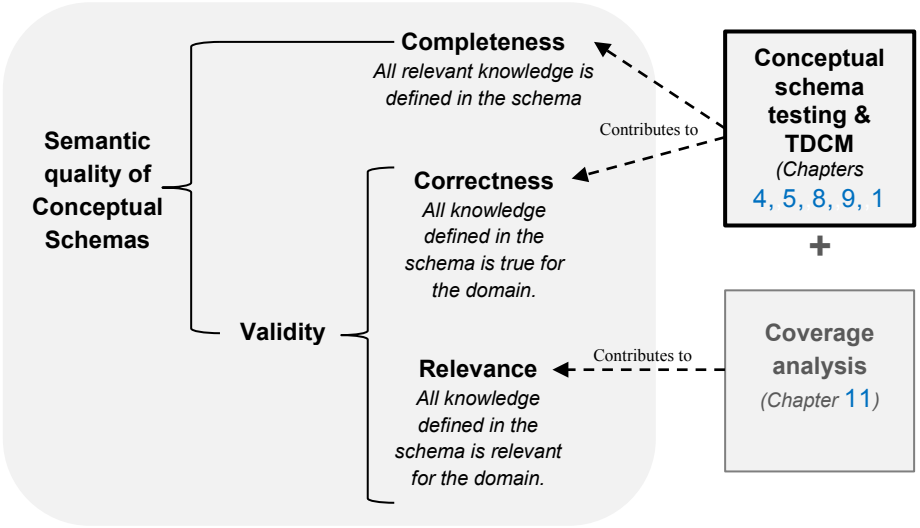


Fig. 20. Contributions to semantic quality by conceptual schema testing

By using conceptual schema testing, we can also ascertain that the part of the conceptual schema involved in the execution of the scenarios defined in test cases is correct, because otherwise some test case execution would not succeed due to failing expectations. However, the testing approach itself does not address the relevance of all the elements of the schema. It may happen that the conceptual schema (correct and complete according to the test set) includes more knowledge that is not relevant for the scenarios defined in the set of test cases, and this irrelevant knowledge could be incorrect.

In conclusion, the approach to test conceptual schemas proposed in this Thesis addresses the completeness and the correctness of the schema in accordance with the test set as sketched by Fig. 20.

However, the relevance of the knowledge defined in the schema is not addressed by the testing approach itself. As an extension of the presented approach, this Thesis presents a basic set of test adequacy criteria which allows automatic analysis of the relevance of all the schema elements. These criteria are a complementary contribution of this Thesis and they are presented in Chapter 11. An extension of the tool presented in Chapter 5 is presented in Section 5.4. The purpose of this extension is performing automatic coverage analysis.

4.4 The CSTL Language

The Conceptual Schema Testing Language (CSTL) is a specialized language developed in this Thesis in order to specify automated tests (Janzen et al. 2005) for testing conceptual schemas. CSTL makes possible to write programs to test conceptual schemas in the style of the modern *xUnit* software testing frameworks (Gamma et al. 1999), which are successfully used in many programming projects.

In these frameworks, test programs include assertions the verdict of which can be automatically computed by a test processor. This fact allows executing test sets as many times as needed, without the participation of people that observes if the results are the expected ones.

The assertions included in the CSTL language comply with the *test kinds* for testing conceptual schemas defined in Section 4.2. The expressiveness of CSTL relies on its capability of formally specifying the defined test kinds. Moreover, CSTL language syntax is designed to be readable and easily understandable to stakeholders, which are usually the source of the defined tests.

4.4.1 CSTL Test Programs

A CSTL program consists of a:

- A fixture (may be empty).
- A set (may be empty) of fixture components.
- A set of one or more test cases.

Fig. 21 shows a fragment of the metamodel of the structure of a CSTL program. We describe each CSTL Program component in the following. Fig. 23 shows a test program example based on the *osCommerce* case study described in Chapter 6. This test program is aimed at testing the conceptual schema of Fig. 24.

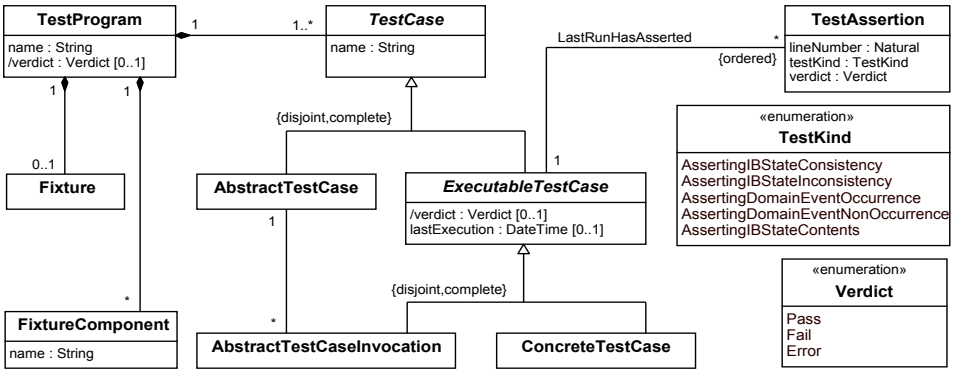


Fig. 21. Metamodel fragment of the structure of a CSTL program.

Test Cases

We adopt the UTP's terminology and consider that a test case is a "specification of one case to test the system including what to test with, which input, result, and under which conditions […] A test case always returns a verdict". The verdict may be *Pass*, *Fail*, *Inconclusive* or *Error* (Object Management Group (OMG) 2005). We consider that the verdict is *Error* when the conceptual schema or the test case is ill-formed (i.e. it is not a valid instance of the corresponding metaschema). The objective of the conceptual modeler is to write test cases whose final verdict is *Pass*.

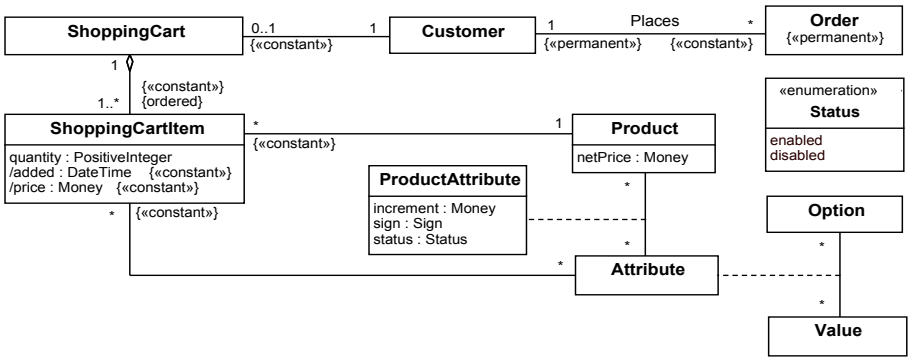
In CSTL, the basic construct for testing conceptual schemas is a *concrete test case*. Each concrete test case has a name and consists of a set of statements (Fig. 22). The last statement of a concrete test case is an assertion, but in general there may be several assertions in the same test case.

```
test testName {
...
assert ...
}
```

Fig. 22. Basic Structure of a Concrete Test Case

```
testprogram OrderConfirmation{

       //FIXTURE
       //Attributes initialization
       shirtSize := new Option;
       extraLarge := new Value;
       small := new Value;
       smallSize := new Attribute(option := shirtSize, value := small);
       extraLargeSize := new Attribute(option := shirtSize, value := extraLarge);
       //Products initialization
       fashionTShirt := new Product;
       fashionTShirt.netPrice := 10;
       smallFashionTShirt := new ProductAttribute
                             (product := fashionTShirt, attribute := smallSize);
       smallFashionTShirt.increment := 2;
       smallFashionTShirt.sign := Sign::minus;
       extraLargeFashionTShirt := new ProductAttribute
                             (product := fashionTShirt, attribute := extraLargeSize);
       extraLargeFashionTShirt.increment := 1;
       extraLargeFashionTShirt.sign := Sign::plus;

       //Customer shopping cart initialization
       c := new Customer;
       sc := new ShoppingCart;
       sc.customer := c;

       fixturecomponent addRegularSizedTShirts{
          item1 := new ShoppingCartItem;
          item1.product := fashionTShirt;
          item1.quantity := 3;
          item1.shoppingCart := sc;
       }

       fixturecomponent addSpecialSizedTShirts{
            item1 := new ShoppingCartItem;
            item1.product := fashionTShirt;
            item1.shoppingCart := sc;
            item1.quantity := 2;
            item1.attribute := Set{smallSize};
            item2 := new ShoppingCartItem;
            item2.product := fashionTShirt;
            item2.shoppingCart := sc;
            item2.quantity := 1;
            item2.attribute := Set{extraLargeSize};
          }

       test emptyShoppingCart{
          assert consistency;
       }

       abstract test confirmedOrderTotal (Fixture itemsAddition, Money expectedTotal){
            load itemsAddition;
            oc := new OrderConfirmation(shoppingCart := sc) occurs;
            assert equals oc.orderCreated.total expectedTotal;
       }

       test confirmedOrderTotal
           (itemsAddition := addRegularSizedTShirts,expectedTotal := 30);


       test confirmedOrderTotal
           (itemsAddition := addSpecialSizedTShirts,expectedTotal := 30);

  }
```

Fig. 23. CSTL program for testing order confirmations in the osCommerce system.




```
context ShoppingCartItem inv onlyOneAttributePerOption:
    self.attribute -> isUnique(option)

context ShoppingCartItem inv productHasTheAttributes:
    self.product.attribute -> includesAll(self.attribute)

context ShoppingCartItem::price:Money
    derive:
        if self.attribute -> isEmpty() then self.product.netPrice
        else
            self.attribute.productAttribute
                -> select(pa|pa.product = self.product) -> collect
                (if sign = Sign::plus then increment
                else -increment
                endif) -> sum() + self.product.netPrice
        endif

context ShoppingCartItem::added:Datetime
    derive: Now()
```

Fig. 24. Fragment of the osCommerce case study focusing on shopping cart items

Fixture

It is assumed that the execution of each test case of a CSTL program starts with an empty IB state. With this assumption, the test cases of a program are independent each other, and therefore the order of their execution is irrelevant. However, the test cases of a program often have a common initial IB state and variables, and it is practical to define this common part, called a fixture, separately.

The fixture is a set of statements that create an IB state and define the values of the common program variables. It is assumed that the execution of a test case starts with the execution of the fixture.

In the example of Fig. 23, the fixture initializes attributes of products and a customer shopping cart, which may be used in the test cases defined within the test program.

Fixture Components

A fixture component is a named set of statements that create a fragment of the state of the IB and define the values of a set of variables. If *fc* is a fixture component, then a test case may use the following statement:

```
load fc;
```

to add to the current state of the IB and to the current variables the fragment defined by *fc*. Fig. 23 shows a test program example based on the *osCommerce* case study reported in Section 6.1. In this example, there are two fixture components: one that instantiates a shopping cart item with a product without attributes, and another that instantiates two items with products having attributes. The same fixture component can be loaded by several test cases.

Concrete and Abstract Test Cases

In CSTL, there are three kinds of test cases: concrete, abstract and abstract invocation.

A concrete test case is a set of statements that builds a state of the IB, defines values of its variables, and executes one or more tests of one of the five test kinds described in the previous section. In Fig. 23, an example is the test *EmptyShoppingCart*, which checks the consistency of the IB state defined by the fixture. The verdict of this test is *Fail*, because the shopping cart *sc* has no items and, according to the cardinality constraints shown in Fig. 24, it should have at least one.

An abstract test case is a parameterized test case intended to be invoked one or more times in the same program. The parameters of an abstract test case may include fixture components and variables. An example in Fig. 23 is *confirmedOrderTotal*. The aim is to test that the total amount of the order created when the customer checks out his shopping cart (event *OrderConfirmation*) is as the domain expert expects (see (Tort 2009b) for further details). It loads the fixture component received, asserts that the domain event *OrderConfirmation* (with characteristic *sc*) occurs, and asserts that the amount of the order just created is *expectedTotal*.

An abstract test case invocation is the invocation of an abstract test case with the desired values of the parameters. In Fig. 23, there are two invocations of the abstract test case. The verdict of the first invocation is *Pass* and the second verdict of the one is *Fail*.

The execution of a concrete test case or of an abstract test case invocation always returns a verdict. The verdict is obtained from the verdicts of the test kinds defined in the previous section as follows:

1. If the verdict of an assertion of a test case is *Error*, then the execution of the test case ends, and the verdict of the whole test case is *Error*.
2. If the verdict of an assertion of a test case is *Fail*, then the execution of the test case ends, and the verdict of the whole test case is *Fail*.
3. The verdict of a test case is *Pass* if the verdicts of all its assertions are *Pass*.

CSTL Statements

All CSTL program components (fitxture, fixture components and test cases) are specified by using CSTL statements.

The formal definition of the CSTL language syntax is given in Appendix A. In the following subsections, we describe the syntax and the semantics of the four kinds of statements related to testing conceptual schemas:

- Statements that update the IB (Section 4.4.2).
- Statements that assert the state of the IB (Sections 4.4.3, 4.4.4 and 4.4.7).
- Statements that create domain events (Section 4.4.5) and assert its occurrence (Sections 4.4.5 and 4.4.6)

4.4.2 Updating the Information Base

When the execution of a test case begins, the IB is assumed to be empty and, during the normal execution of an IS, the IB can only change due to the occurrence of domain events, and a particular IB state can only be reached by the successful occurrence of one or more domain events. However, in testing conceptual schemas, we often need to set up an IB state independently of the domain events (Mellor et al. 2002). This happens when the domain events are not available, i.e. they have not yet been specified, or when we want to check an inconsistent state that cannot be reached by valid domain events.

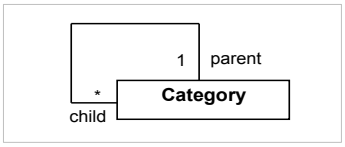


Fig. 25. A basic schema fragment

CSTL includes statements in order to explicitly set up an IB state in a test case. We describe them below using examples based on the schema fragment of Fig. 25.

We define that *entityID* is a new instance of the entity types $EntityType_1$,..., $EntityType_n$ with the following statement:

```
entityID := new EntityType1,...,EntityTypen;
```

where *entityID* must be a new identifier. For example:

```
cat1 := new Category;
```

Note that we assume *multiple classification* (Olivé 2007), and therefore we allow that an entity is an instance of two or more entity types not related by generalization/specialization relationships.

Entities can be deleted with the following statement:

```
delete entityExpr;
```

where *entityExpr* is an identifier or, in general, an OCL expression that evaluates to a previously created entity. The deletion of an entity implies the deletion of its attributes and the links in which it participates.

To define that the value of attribute *att* of entity *entityID* is *val* (where *val* is a valid OCL expression) we write:

```
entityID.att := val;
```

The types of *att* and *val* must be compatible; otherwise the verdict of the test case in which the statement appears is *Error*.

Similarly, to define that the entity *entityID* is related with role *role* in a binary link (an instance of an association) to one or more entities given by the OCL expression *participants* we write:

```
entityID.role := participants;
```

The types of *role* and *participants* must be compatible; otherwise the verdict of the test case in which the statement appears is *Error*. For example:

```
cat1.parent := cat2;
```

Often, it is convenient to state in a single statement the creation of a new entity *entityID* as an instance of entity type $EntityType_1$ and the assignment of an initial value for its attributes $att_1,...,att_n$ and of its binary links with roles $r_1,...,r_m$. The syntax is as follows:

```
entityID := new EntityType1
                (att1:= value1,..., attn:= valuen,
                 r1:= participants1,..., rm:= participantsm);
```

where $value_i$ and $participants_i$ are OCL expressions. For example:

```
cat3 := new Category(parent:=cat1);
```

Instances of an n-ary UML association *Assoc* with roles $r_1,...,r_n$ are created with the following statement:

```
new Assoc(r1:= entityExp1,..., rn:= entityExpn);
```

If *Assoc* is an association class, then the above statement returns the identifier of the instance of that class.

4.4.3 Asserting the Consistency of an IB State

A conceptual schema may include a large number of constraints, and the task of ensuring that all of them are correct is far from trivial. Using an appropriate modeling environment, testing the schema may be a practical means of uncovering faulty constraints. This is done by setting up one or more test cases, each with an IB state that domain experts believe is valid, followed by an assertion that the state satisfies all defined constraints. An IB state is called consistent if it satisfies all constraints defined in the conceptual schema.

In CSTL, the conceptual modeler asserts that the current IB state must be consistent by writing the following statement:

```
assert consistency;
```

The verdict of the assertion is *Pass* if the IB state satisfies all defined constraints. The verdict is *Fail* if, contrary to what the conceptual modeler expects, the IB is not consistent. When the verdict is *Fail* two cases are possible: (1) domain experts consider that the IB state is indeed invalid or, if it is valid, then (2) the non-satisfied

constraint(s) is incorrect. In the former, the problem lies in the test case, and the conceptual modeler may prefer to change the assertion to *assert inconsistency* (see below). In the latter, the corresponding constraint(s) must be corrected.

As an example, assume the schema shown in Fig. 25, and consider the following test case:

```
test CategoryCanBeInstantiated{
      cat := new Category;
      assert consistency;
}
```

The verdict will be *Fail* because according to the schema, each category should have a parent. If the domain experts confirm that the IB state just defined is valid (*cat* is a root category), then the test case has uncovered a schema error, which must be corrected by changing the cardinality (multiplicity) of *parent* to 0..1.

A conceptual modeler may use this kind of assertion not only to check that the constraints defined in the schema behave as expected, but also to check that each entity type is satisfiable (i.e. it may have a non-empty population). If the conceptual modeler is able to set up an IB state in which there is at least one instance of entity type *E* and that state is asserted as consistent, then by definition *E* is satisfiable. If the conceptual modeler is unable to set up such a state, this is not a formal proof that *E* is unsatisfiable, but in many practical cases it provides a clue that helps to uncover a faulty constraint.

For example, consider the original schema of Fig. 25 again, and assume that it is extended with the following invariant:

```
context Category inv thereMayNotBeCycles:
  not self.allChildren() -> includes(self)
```

where the operation *allChildren()* returns the whole set of subcategories of the category in the hierarchy (Gogolla et al. 2005).

Now *Category* becomes unsatisfiable, and the conceptual modeler is unable to set up a test case with at least one instance of *Category* and an *assert consistency* statement whose verdict is *Pass*. This result is not formal proof that *Category* is unsatisfiable, but in many practical cases it provides a clue that helps the conceptual modeler to uncover a faulty constraint. In this example, *Category* is satisfiable if we make the same change as above: set the cardinality of *parent* to 0..1.

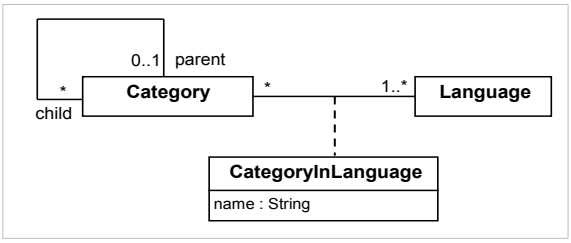


Fig. 26. Extension of Fig. 25 with names of categories

4.4.4 Asserting the Inconsistency of an IB State

For the set of constraints defined in the schema to be correct and complete, not only the constraints must be satisfied by valid IB states, but those constraints must also rule out invalid states. Testing the schema may be a practical means of detecting missing constraints. This is done by setting up one or more test cases; each with an IB state that domain experts believe is invalid, followed by an assertion that the state does not satisfy at least one schema constraint.

In CSTL, in order to assert that the current IB state is inconsistent, the conceptual modeler simply writes the following statement:

```
assert inconsistency;
```

The verdict of the assertion is *Pass* if the IB state does not satisfy one or more constraints; otherwise, it is *Fail*. If the verdict of the assertion is *Fail* then two cases are possible: (1) the IB state is indeed valid or, if it is not, then (2) some constraint is missing. In the former, the conceptual modeler may prefer to change the assertion to *assert consistency* (see above). In the latter, the conceptual modeler must define a new constraint or refine an existing one in order to make it more constraining.

For example, consider the schema of Fig. 26, which extends the schema of Fig. 25 with categories in a multilingual e-commerce system. Assume that the conceptual modeler writes the following test case to assert that if a category has no name in one language, then the IB state is inconsistent:

```
test CategoriesHaveANameInEachLanguage{
      cat := new Category;
      english := new Language;
      catalan := new Language;
      catInEnglish := new CategoryInLanguage
                        (category:=cat, language:=english);
      catInEnglish.name := "Food";
      assert inconsistency;
}
```

Contrary to what it is expected, the execution of the test case fails. This fact reveals to the conceptual modeler that the schema does not enforce that categories have a name in each language. If the domain expert confirms this requirement, then the test case has uncovered a schema error, which must be corrected by adding the following constraint to the schema:

```
context Category inv CategoriesHaveANameInEachLanguage:
  self.language = Language.allInstances()
```

Now, the verdict of the above test case is *Pass*, as desired.

4.4.5 Asserting the Occurrence of Domain Events

A domain event type is a complex schema element that may have several kinds of defects:

1. The constraints of the event type may not allow the occurrence of valid events.
2. The postconditions may not precisely define the intended effect of events.
3. The method of the effect operation may produce an IB state that does not satisfy both the postconditions and the schema constraints.

Testing the schema may be a practical means of detecting those defects. This is done by setting up for each domain event type one or more test cases with an IB state and an instance of that event type that domain experts believe may occur in that state, followed by an assertion of the (satisfactory) occurrence of that event.

In CSTL, the instances of a domain event type can be created in the same way as those of entity types. If $EventType_1$ is a domain event type, then:

```
eventId := new EventType1;
```

creates the instance *eventId* of $EventType_1$, whose characteristics (attribute values, binary links) can be defined as in the case of entities. Often, it is convenient to define in a single statement the creation of a new event *eventId* as an instance of domain event type $EventType_1$ and the assignment of the value for its attributes $att_1,...,att_n$ and of its binary links with roles $r_1,...,r_m$. The syntax is as follows:

```
eventId := new EventType1(att1:=value1,..., attn:=valuen,
                r1:=participants1,..., rm:=participantsm);
```

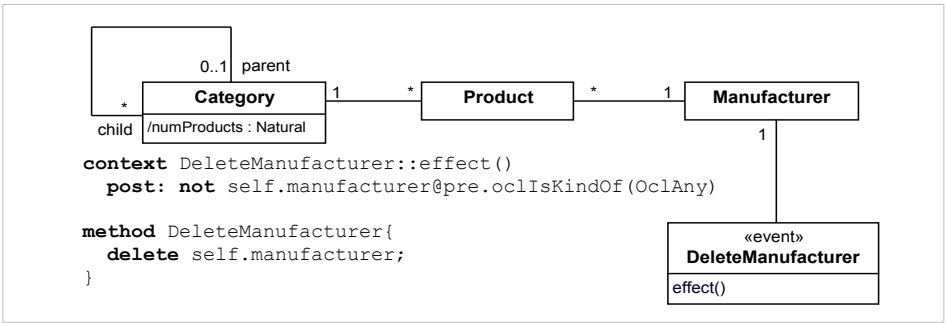


Fig. 27. Extension of Fig. 26 with products and manufacturers

Once the concrete event *eventId* has been created in a test case in order to assert that it may occur in the current state of the IB, the conceptual modeler writes the following sentence:

```
assert occurrence eventId;
```

The verdict of this assertion is determined as follows:

1. Check that the current IB state is consistent (as defined in Section 4.4.3). The verdict is *Error* if that check fails (events may not occur in inconsistent IB states).

2. Check that the constraints of the event are satisfied. The verdict is *Fail* if any of the event constraints is not satisfied.

3. Execute the method of the corresponding *effect()* operation.

4. Check that the new IB state is consistent (as defined in Section 4.4.3). The verdict is *Fail* if any of the constraints is not satisfied.

5. Check that the event postconditions are satisfied. The verdict is *Fail* if any of the postconditions is not satisfied; otherwise the verdict of the whole assertion is *Pass*.

If the verdict from step 1 is *Error*, then the conceptual modeler must change the IB state in order to make it valid. If the verdict from steps 2, 3 or 4 is *Fail*, then the event has not occurred as expected by the conceptual modeler. If the verdict from step 2 is *Fail* then the following two cases are possible: (1) domain experts consider that the IB state and event occurrence are indeed invalid or, if it is valid, then (2) the non-satisfied constraint(s) is incorrect. In the former, the conceptual modeler may prefer to change the assertion to *assert non-occurrence* (see below). In the latter, the corresponding event constraint(s) must be corrected. If the verdict from step 4 is

Fail, then the effect method, the event constraints or some schema constraint must be ill-specified. If the verdict from step 5 is *Fail* then either the method of the effect operation or some postcondition is incorrect: the method may not produce the intended IB state, or the postconditions may be ill-specified.

As an example, assume the extension of Fig. 26 shown in Fig. 27, in which products are classified in a category and manufactured by a manufacturer. The figure also shows the specification of the *DeleteManufacturer* event, including the postcondition and the method of the corresponding effect operation. The postcondition states that the manufacturer given by the event (`self.manufacturer@pre`) must not exist after the event occurrence (in OCL this can be stated by requiring that it is not an instance of *OclAny*, the supertype of all types). The only event constraint is that an instance of *DeleteManufacturer* is associated with a manufacturer. Consider, now, the following test case:

```
test AManufacturerIsDeleted{
      cat1 := new Category;
      english := new Language;
      cat1InEnglish := new CategoryInLanguage
                       (category:=cat1, language:=english);
      cat1InEnglish.name := "Food";
      m1 := new Manufacturer;
      p1 := new Product(manufacturer:=m1, category:=cat1);
      dmEv := new DeleteManufacturer(manufacturer:=m1);
      assert occurrence dmEv;
}
```

The execution of the test case fails (as detected in step 4) because the occurrence of the event *dmEv* reaches an invalid state in which *p1* is not manufactured by any manufacturer. There are at least two possible actions that can be performed to make the test case *Pass*:

1. If products may exist without a manufacturer, then the cardinality of *manufacturer* is set to 0..1.

2. If domain experts confirm that when a manufacturer is deleted, the system must also delete the products it manufactures, then the postcondition and the method of the event *DeleteManufacturer* must be changed to the following:

```
context DeleteManufacturer::effect()
  post theManufacturerDoesNotExistAnyMore:
    not self.manufacturer@pre.oclIsKindOf(OclAny)
  post theProductsAreDeleted:
    self.manufacturer@pre.product@pre->forAll(p|
                                  not p.oclIsKindOf(OclAny))
```

```
method DeleteManufacturer::effect(){
  delete self.manufacturer;
  for each p in self.manufacturer.product do
       delete p;
  endfor
}
```

A conceptual modeler may use this kind of assertion not only to check that the domain events defined in the schema behave as expected, but also to check that each domain event type is satisfiable. A domain event type is satisfiable if there is at least one consistent IB state and one instance of that event type such that the event constraints are satisfied (Olivé 2007). If the conceptual modeler is able to set up an IB state and an instance of the event type *Ev* for which *assert occurrence* gives the verdict *Pass*, then by definition *Ev* is satisfiable. If the conceptual modeler is unable to set up such an IB state and event, this is not a formal proof that *Ev* is unsatisfiable, but, in many practical cases, it provides a clue that helps to uncover a faulty event specification.

As an example, assume that we extend the schema in Fig. 27 with a new event type named *AddNewCategory* with characteristics *product* and *addedCategory*. The effect of *AddNewCategory* is adding a new category to a product. Consider, also, the following event constraint:

```
context AddNewCategory inv productHasNoCategories:
      self.product.category->isEmpty()
```

The conceptual modeler is unable to set up any IB state in which the event constraints are satisfied because the event *AddNewCategory* is unsatisfiable.

4.4.6 Asserting the Non-Occurrence of Domain Events

A correct domain event specification must not only accept valid event instances, but also reject invalid ones. An event instance is invalid if it may not occur in the current IB state. Testing the schema may be a practical means of detecting missing event constraints. This is done by setting up for each domain event type one or more test cases with an IB state and an instance of that event type that domain experts consider may not occur in that state, followed by an assertion of the non-occurrence of that event.

In CSTL, in order to assert that the event *eventId* may not occur, the conceptual modeler writes the following sentence:

```
assert non-occurrence eventId;
```

The verdict of this assertion is determined as follows:

1. Check that the current IB state is consistent (as defined in Section 4.4.3). The verdict is *Error* if that check fails.

2. Check the satisfaction of the event constraints. The verdict is *Fail* if the event constraints are satisfied and *Pass* if one or more event constraints are not satisfied.

If the verdict of the assertion is *Fail*, then two cases are possible: (1) the event is indeed valid or, if it is not, then (2) some event constraint is missing. In the former, the event may occur in the domain, and the conceptual modeler may prefer to change the assertion to *assert occurrence* (see above). In the latter, the conceptual modeler must define a new event constraint or refine an existing one in order to make it more constraining.

In the example of Fig. 27, if we assume now that a manufacturer cannot be deleted if there are products manufactured by it, then the following event constraint must be added:

```
context DeleteManufacturer inv manufacturerHasNoProducts:
  self.manufacturer.product->isEmpty()
```

With the above event constraint, the following test case will pass (as expected):

```
test theManufacturerCannotBeDeleted{
      cat1 := new Category;
      english := new Language;
      cat1InEnglish:= new CategoryInLanguage
                      (category:=cat1, language:=english);
      catInEnglish.name := "Food";
      m1 := new Manufacturer;
      p1 := new Product(manufacturer:=m1, category:=cat1);
      dmEv2 := new DeleteManufacturer(manufacturer:=m1);
      assert non-occurrence dmEv2;
}
```

4.4.7 Asserting the Contents of an IB state

It is often useful to include in a test case an assertion on the current state of the IB. The purpose may be to check that one or more derivation rules derive the expected results, or that a navigational expression yields the expected results or that the effect of one or more domain events implies an expected result in the IB.

In CSTL, to assert that the current state of the IB satisfies a *boolean* condition defined in OCL, the conceptual modeler writes the following statement:

```
assert true booleanExpression;
```

where *booleanExpression* is an OCL expression over the types of the IB and the variables of the test case. The verdict of the assertion is *Error* if the current state is inconsistent (as defined in Section 4.4.3). The verdict is *Pass* if *booleanExpression* is true and *Fail* otherwise. If the verdict is *Fail*, two cases are possible: (1) *booleanExpression* should not be *True* or (2) the derivation rules and/or domain events do not give the expected results. In the former, the conceptual modeler may prefer to change the assertion to *assert false* (see below). In the latter, the conceptual modeler must change the derivation rules and/or the domain events specification.

A similar CSTL statement is as follows:

```
assert false booleanExpression;
```

Additionally, CSTL includes the following assertions:

```
assert equals valueExpression1 valueExpression2;
assert not equals valueExpression1 valueExpression2;
```

As an example, consider again the schema of Fig. 27 and that the derivation rule of the derived attribute *numberOfProducts* is defined as follows:

```
context Category::numberOfProducts:Natural
   derive: self.product->size()
```

A conceptual modeler that wants to test that derivation rule may write the test case:

```
test NumberOfProductsInACategory{
      cat1:= new Category;
      english := new Language;
      cat1InEnglish := new CategoryInLanguage
                       (category:=cat1,language:=english);
      cat1InEnglish.name := "Food";
      cat2 := new Category;
      cat2InEnglish := new CategoryInLanguage
                      (category:=cat2, language:=english);
      cat2InEnglish.name := "Bakery";
      cat2.parent := cat1;
      m := new Manufacturer;
      p1 := new Product(manufacturer:=m, category:=cat1);
      p2 := new Product(manufacturer:=m, category:=cat2);
      assert equals cat2.numberOfProducts 1;
      assert equals cat1.numberOfProducts 2;
}
```

The verdict of the first assertion is *Pass*, but that of the second is *Fail*. The conceptual modeler expects that *numberOfProducts* includes the products of the children categories, and therefore the result should be 2 (*p1* and *p2*). The derivation rule does not derive the expected results because it does not take into account the products of children categories. The test case will pass if the derivation rule is corrected as follows:

```
context Category::numberOfProducts:Natural
   derive:
        let allParents() : Set(Category) =
          self.parent -> union(self.parent.allParents())
        in
        Category.allInstances() -> select(c | c.allParents()
        -> includes(self) or c=self).product->size()
```

5 The CSTL Processor

In Chapter 4, we proposed an approach for testing executable conceptual schemas. In this chapter, we present the *CSTL Processor* (Tort et al. 2011b), a research prototype that supports the specification, management and execution of automated tests (Janzen et al. 2005) of executable conceptual schemas. The *CSTL Processor* makes the proposed testing approach feasible in practice.

In Section 5.1 we describe the tool architecture. In Sections 5.2, 5.3 and 5.4, we present the main tool components of the *CSTL Processor* and remarks about its implementation. In Section 5.5, we describe the extension of the processor that deals with temporal constraints.

5.1 General Overview and Architecture

The *CSTL Processor* is a testing tool that supports automated testing of conceptual schemas specified according to our testing approach presented in Chapter 4. The CSTL Processor is available for downloading from the project website (Tort 2010). Video tutorials with examples of use for different purposes may also be found in the project website, together with additional information and resources such as source files, screenshots and complementary documentation.

The *CSTL Processor* has been developed in the context of *Design Research* (Hevner et al. 2004), which is the general framework of the present research work (Section 1.4). The development and refinement of the contributions presented in this Thesis were supported by the knowledge and experience acquired during continuous development of this tool and by its application in several case studies (Chapter 6 and Chapter 9).

This prototype is intended to be used in different application contexts in which conceptual schemas can be tested. The *CSTL Processor* supports:

- Test-last validation, in which correctness and completeness are checked by testing after the schema definition.
- Test-first development of conceptual schemas, in which the elicitation and definition of the schema is driven by a set of test cases.

The Test-Driven Conceptual Modeling (TDCM) method described in Chapter 8 is a test-first conceptual modeling method. TDCM requires the execution of CSTL test cases and, consequently, the *CSTL Processor* is also a necessary resource to put this method in practice.

The implemented release of the tool deals with schemas defined in UML/OCL (see Section 2.1.1). Additionally, the *CSTL Processor* is also able to deal with a representative set of constraints that involve two successive states of the IB, and on creation-time constraints. The definition of these additional features and its implementation are explained in Section 5.5.

Fig. 28 shows the main components of the *CSTL Processor* environment. The execution of test programs is done by the *Test Processor*, while the execution of conceptual schemas is done by the *Information Processor*. Moreover, automatic coverage analysis (according to the basic set of test adequacy criteria proposed in Chapter 11) is provided by the *Coverage Processor*.

The *CSTL Processor* integrates user input facilities for managing and executing both the CSUT and the test set.

The main features of the *CSTL Processor* are:

1. The definition of executable conceptual schemas under test.
2. The definition and management of CSTL test programs.
3. The execution of the test set and the automated computation of its verdicts, including reports of error and failing information.
4. The automatic analysis of testing coverage according to a basic set of testing adequacy criteria.

The user interface of the tool is composed by four tabs. Each tab corresponds to one of the above main features.

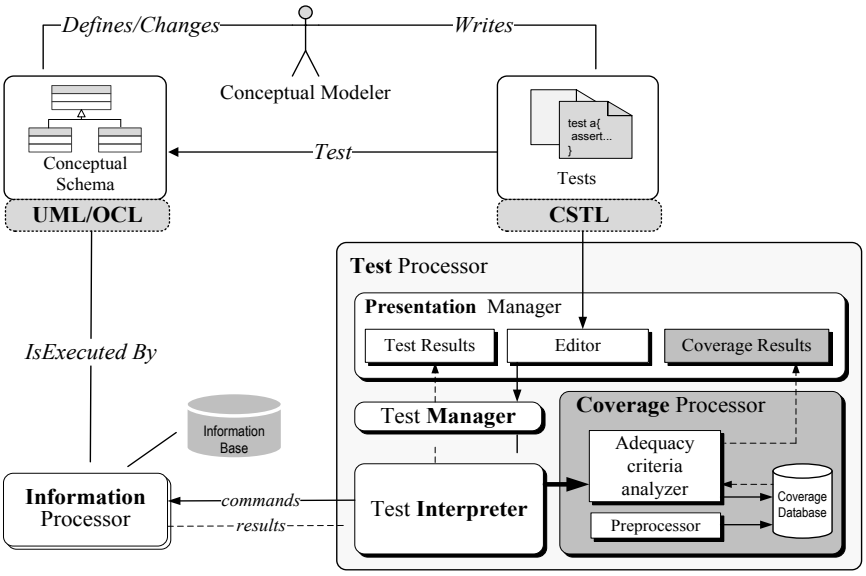


Fig. 28. CSTL Processor architecture

In the following sections, we describe the responsibilities and the implementation of the *Information Processor* (Section 5.2), the *Test Processor* (Section 5.3) and the *Coverage Processor* (Section 5.4).

5.2 Information Processor

The *Information Processor* provides functionalities for creating and editing the *Conceptual Schema Under Test* (CSUT), and it is responsible for its execution when requested by the *Test Processor* (Section 5.3). Fig. 29 shows the components of the *Information Processor*. We describe them in the following according to the three main capabilities of this component: (1) Managing the CSUT, (2) executing the CSUT, and (3) querying the CSUT.

5.2.1 CSUT Management

As explained in Section 2.1, the specification of the conceptual schema in an executable form includes both the definition of the structural knowledge and the behavioral knowledge as a set of domain events. Moreover, each domain event is associated to its procedural specification (*method*) in order to allow the simulation of its execution.

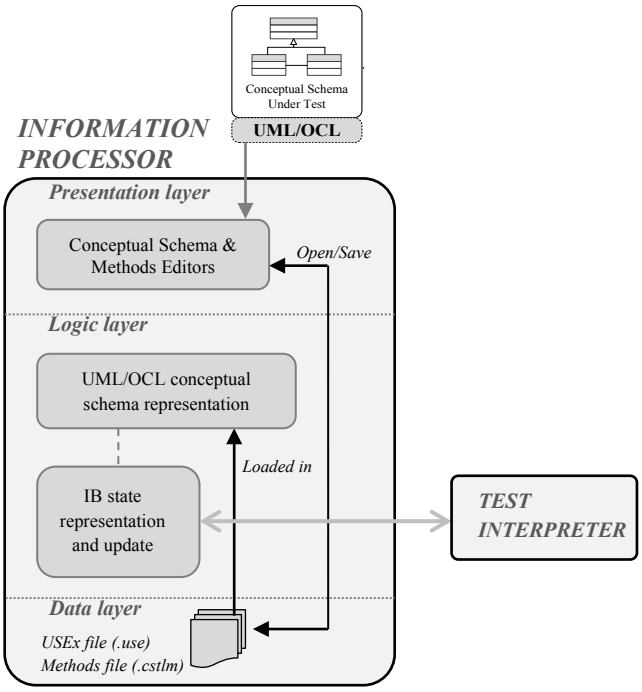


Fig. 29. Information Processor design

The *Information Processor* provides two editors to specify both the conceptual schema and the methods of the domain events.

An extended USE (Gogolla et al. 2005) syntax is used to specify the conceptual schemas under test in an executable form. The extensions included in the USE syntax are described in the next section. The methods are written in a subset of the CSTL language (Section 4.4) which does not include the assert statements.

Fig. 30 shows a screenshot of the CSUT editor that provides syntax validation and automatic highlighting of the language keywords. The CSUT editor also provides functionalities to open existing conceptual schemas, saving them as persistent files, and opening them in the structural diagram view provided by the USE tool (Gogolla et al. 2005).

5.2.2 CSUT Execution

The execution of test cases is required in order to execute test cases, which are sequences of IB states that represent user stories. In order to simulate the

execution of conceptual schemas, the *Information Processor* needs representing the conceptual schema in memory and providing operations in order to set up and update the information base state.

In order to represent the conceptual schema in memory, the *CSTL Processor* includes an extended implementation of the subset of the UML and OCL metamodels (Object Management Group (OMG) 2009, Object Management Group (OMG) 2010b) provided by the core of USE (Gogolla et al. 2005).

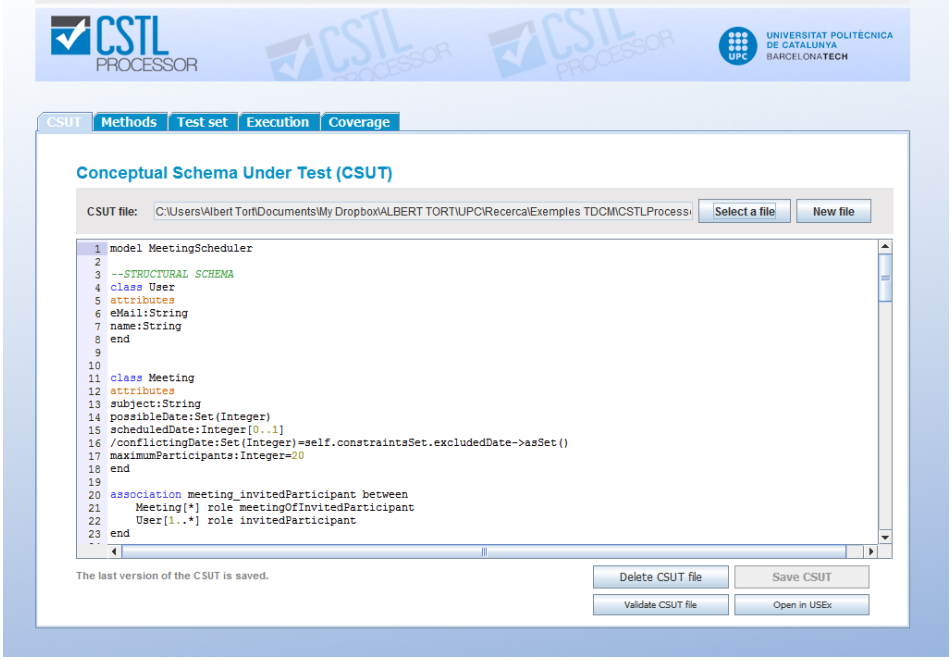


Fig. 30. CSUT Management screenshot

The implementation of the schema representation in memory consists of a set of Java classes (Fig. 31). Each Java class implements a *metaclass* of the metamodels that specify UML and OCL modeling languages. Since a conceptual schema specification is an instantiation of these metamodels, a conceptual schema may be represented in *Java* as a set of *objects,* which are instance of *Java* classes.

The IB state of a conceptual schema is also implemented in *Java.* The IB state is a set of instances of classes with operations to setup IB states by creating, deleting and updating entities, attributes and associations.

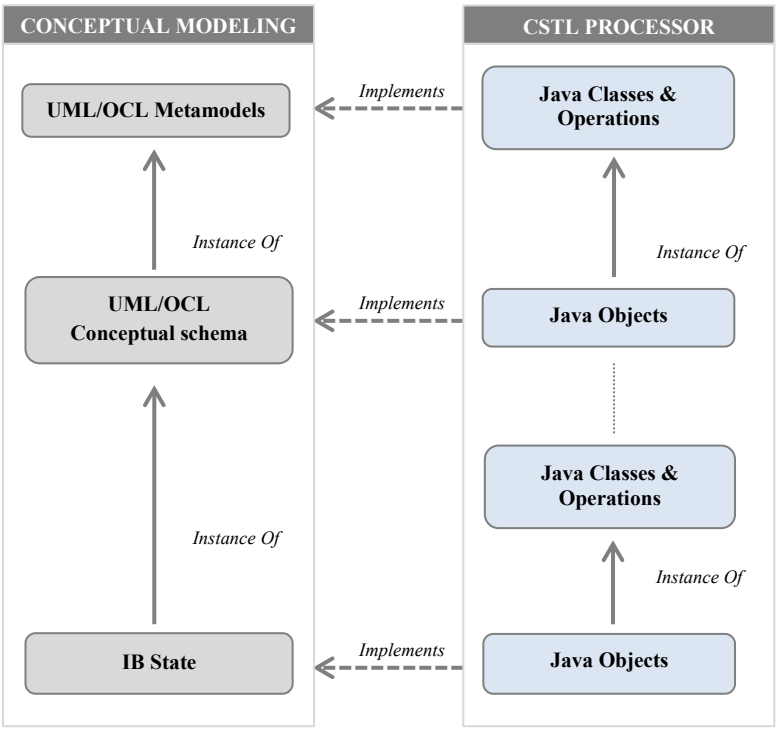


Fig. 31. Conceptual schema representation in memory

The current implementation of our testing prototype assume *static classification* (when an entity is created in the IB at some point in time, it is classified as an instance of one or more entity types at that time and these classifications do not change during the entities's lifespan). Reclassification operations should be considered in order to admit entities to be instances of different entity types at different times (*dynamic classification*).

The USE core has been extended in order to be able to deal with several new language features such as derived attributes, default values, multiplicities in attributes, domain events, temporal constraints, initial integrity constraints and generalization sets.

The extension also includes changes in the user interface in order to show the new schema elements. This extension is named *USEx* and it is comprehensibly described in the documentation of the *CSTL Processor* (Tort 2010). The referenced

description includes examples of use. In the following, we summarize the main extensions which are relevant to this work.

Derived attributes

If a property (attribute or association end) is derived, then its value or values can be computed from other information. A derivation rule specifies how to compute the derived information. *USEx* allows specifying derivation rules as OCL expressions.

USEx graphical view shows derivation rule expressions in the frame that describes the schema elements. It also automatically computes and represents the value of derived types in the object diagram view, by taking into account the current Information Base state. The OCL evaluation dialog also takes into account the derived information when evaluating side-effect free expressions.

The syntax is as follows:

Attribute
```
[/] attributeName:attributeType [multiplicity]
[=derivationOCLExpression]
```

Association end
```
associationEndClassName multiplicity role roleName
[= derivationOCLExpression]
```

Default values

Default values can be defined for properties (attributes and association ends). In *USEx*, the default value is specified as an OCL expression. This expression is computed when an instance of the property is evaluated, only in the case that a value of the property has not been explicitly set. The default value becomes the initial value (or values) of the property. The resultant type of the default OCL expression must conform to the property type or must result in a collection of objects of the property type (if its upper multiplicity is greater than 1).

USEx graphical view shows the default expressions in the schema elements description frame. It also shows the expression in the class diagram. Moreover, it automatically computes and represents the computed value in the object diagram, taking into account the Information Base state. The OCL evaluation dialog also takes into account the default values when evaluating side-effect free expressions.

The syntax is as follows:

Attribute
```
attributeName:attributeType [multiplicity]
[=defaultOCLExpression]
```

Association end
```
associationEndClassName multiplicity role roleName
[=defaultOCLExpression]
```

Multiplicity of attributes

The multiplicity of attributes allows users to specify a lower and an upper value of their cardinality. *USEx* allows the specification of attributes' multiplicity. By default, if no multiplicity is defined, the multiplicity of attributes in *USEx* is [1..1].

The syntax is as follows:

```
( class | event | domainevent | query | datatype ) name
  attributes
     attributeName: attributeType [lower..upper]
end
```

Note that allowed multiplicities are:

- [n..*] or only [*].
- [n..m] where *n* and *m* are Naturals, so that n < m.
- [n..n] or only [n] where both lower and upper values are equal.

Domain events

The definition of the domain event types is crucial for the specification of the behavioral schema, as explained in Chapter 2.

USEx supports the specification of domain events. The syntax is as follows:

```
domainevent  name
   attributes
   operations
   constraints
end
```

Temporal constraints

Temporal constraints take into account the evolution of the IB state. Many kinds of temporal constraints have been studied in the literature. *USEx* allows the specification of *constant*, *permanent* and *creation-time* invariants. More information on these constraint types can be found in (Olivé 2007) and in Section 5.5, where the implementation of temporal constraints in the *CSUT Processor* is discussed.

USEx allows specifying these constraints, which are shown in the schema elements description view and, also, in the class diagram view provided by the USE tool.

The syntax is as follows:

Constant/Permanent classes

```
[abstract] [constant | permanent] class  className
   attributes
   operations
   constraints
end
```

Constant/Permanent attributes

```
[/] attributeName:attributeType [multiplicity] [constant |
permanent]
```

Constant/Permanent association ends

```
associationEndClassName multiplicity role roleName
[constant | permanent]
```

Creation-time invariants

```
[ini] inv invariantName: invariantExpression domainevent  name
   attributes
   operations
   constraints
end
```

Multiple Classification

Multiple classification allows objects to be instance of one or more classes. USEx incorporates multiple classification to USE. Therefore, objects can be instance of more than one class.

USEx extends the *object creation dialog* of USE to allow the creation of objects classified into one or more classes. Moreover, object diagrams specify the classes of each object.

Generalization Sets

Multiple classification allows objects to be instance of one or more classes. *USEx* assumes multiple classification and so that, objects can be an instance of more than one class. In this context, *USEx* allows the specification of generalization sets and its constraints (see Chapter 2 for details about taxonomic constraints).

The syntax is as follows:

```
generalizationset (disjoint | overlapping)
                  (complete | incomplete)
between
   genclass parentName
   specificclass descendantName1 [ , descendantName ]*
end
```

Data Types

A data type consists of a set of values and a set of lexical representations, or literals. The set of values is the population of the data type, and is called the value space of the type. The set of lexical representations is called the lexical space of the type. Each value in the value space is denoted by one or more literals of the lexical space. Values are represented in an information base by means of one of their literals. The value space of a data type does not change over time. For this reason, data types are constant entity types.

The current implementation of the test processor prototype only admits the predefined data types *String, Boolean, Decimal* and *Integer*, although other custom data types may be defined. The syntax is as follows:

```
[abstract] datatype name
   attributes
   operations
   constraints
end
```

For example, we can specify an *Email* datatype as follows:

```
datatype Email
attributes

       email:String
end
```

5.2.3 Queries about the Information Base State

In order to compute the verdict of test assertions, the *Test Processor* (Section 5.3) may need to request the evaluation of OCL expressions about the IB State. OCL expressions that represent queries do not change the state of the domain (they are side-effect free).

The evaluation of OCL expressions on the IB state consists of three main steps:

- Syntactic analysis of the OCL expression.
- Parsing of the expression and its instantiation as *Java classes* that implement a semantic tree.
- Computation of the OCL expression by using the operations associated to each node type included in the semantic tree.

Operations aimed to query the IB State are provided by *USE* and extended by our *USEx* implementation according to the schema elements we deal with.

5.3 Test Processor

The *Test Processor* implements the management and the execution of the test cases. Fig. 32 shows the main components of the *Test Processor*, which consists of the *presentation manager*, the *test manager* and the *test interpreter*. In the following, we describe them.

5.3.1 Presentation Manager

The *Presentation Manager* implements two user interface parts: the one related to the management of the test set (Fig. 33), and the one related to the presentation of the execution results (Fig. 34). The user interface of the *CSTL Processor* is implemented in *Java Swing* (Loy et al. 2002), assisted by a specialized tool to design graphical interfaces in Java, called *JFormDesigner*.

The user interface for managing test programs allows creating, editing and deleting test cases, which are saved as persistent files in a specified directory of the files system. The editor for test cases is implemented by using the *JSyntaxPane* library. *JSyntaxPane* provides resources to handle basic syntax highlighting and editing of various languages within *Java Swing* application. Since *JSyntaxPane* does not include syntax highlighting for CSTL and USE, we have extended it to allow them.

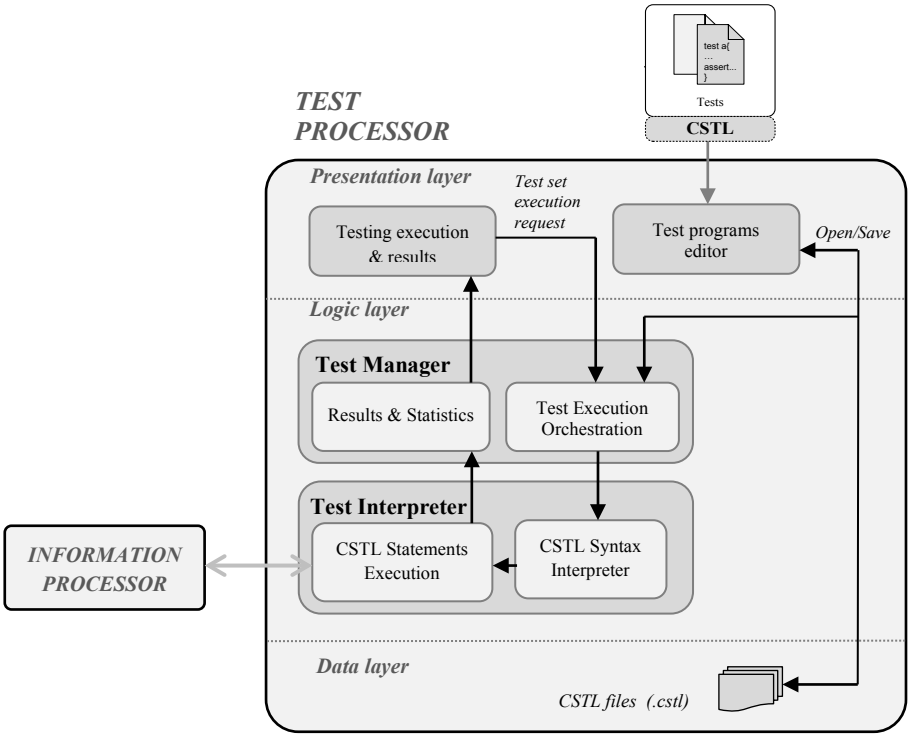


Fig. 32. Test Processor Design

On the other hand, the presentation of results includes a button to request the execution of test cases. After the execution, this module shows the errors (if any), the global verdict of the whole test set, and a visual tree that contains the summary of the verdict of all test cases in a hierarchical form. All this information is collected, organized and transmitted to this component by the *Test Manager*.

Fig. 34 shows the result of the execution of a CSTL program example. One test case has passed whereas the other one has failed, and therefore the global verdict is *Fail*.

Note that the test processor indicates the test case that fails, the number of the lines where the failure has been revealed, and gives an explanation of the failure in natural language. This information assists the modeler in order to point out the errors and failures.

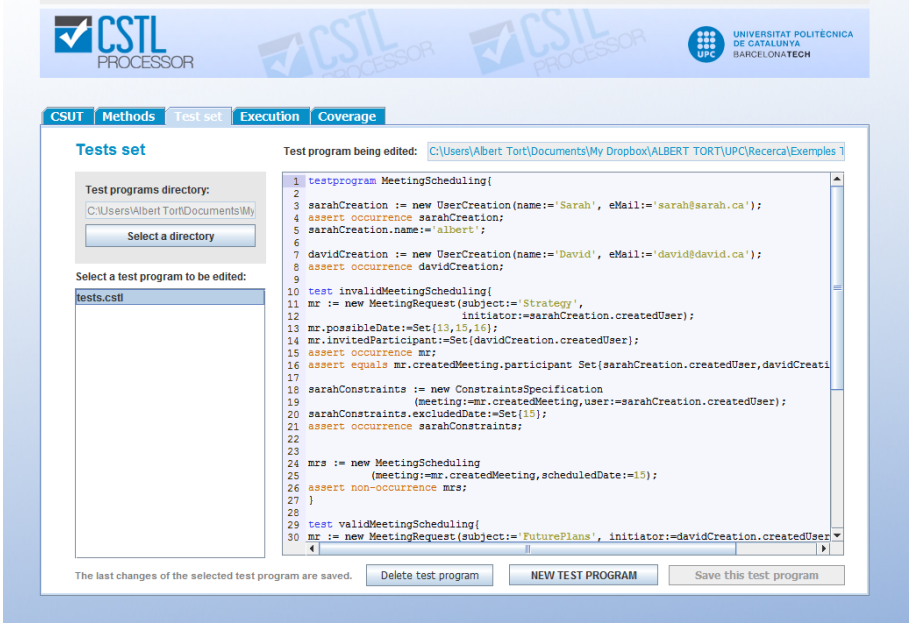


Fig. 33. Screenshot of a test set management example in the CSTL Processor

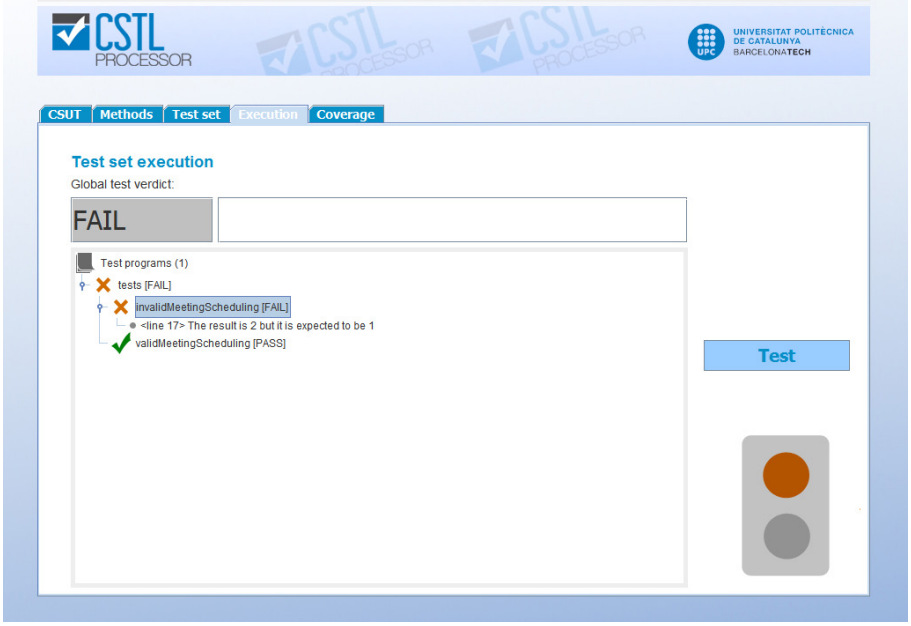


Fig. 34. Screenshot of a test execution report in the CSTL Processor

5.3.2 Test Manager

The *Test Manager* has two main roles in the process of executing test cases: (1) the *orchestration* of the test cases execution, and (2) the collection and organization of the results to be shown by the *Presentation Manager* (Section 5.3.1). This logical module is implemented in *Java*.

The *orchestration* of test cases consists of collecting all the test cases that are included in the test set, and requesting its execution by the test interpreter. The individual results provided by the *Test Interpreter* (Section 5.3.3) are collected, organized and transmitted to the *Presentation Manager* (Section 5.3.1). Moreover, the global verdict is computed as follows:

- The global verdict is *Pass* if the verdict of all test programs is *Pass*.
- The global verdict is *Error* if the verdict of one or more test programs is *Error.*
- The global verdict is *Fail* if there no errors in any test case, and the verdict of one or more test programs is *Fail.*

5.3.3 Test Interpreter

The *Test Interpreter* has two main roles: (1) read and parser the test programs written in CSTL (Section 4.4) and (2) perform the execution of the test cases specified in test programs as requested by the *Test Manager* (Section 5.3.2).

CSTL test programs are read by a module of the *Test Interpreter*, which has been implemented by using ANTLR facilities (Parr 2007). ANTLR is a parser generator for *Java* from grammatical descriptions. The result is a parser tree composed by nodes that are associated to a node type. Each node type has an associated operation that evaluates it.

For each test case, the interpreter sets up the common fixture (if any), executes the statements of each test case and computes the verdicts. The interpreter invokes the services of the *Information Processor* (Section 5.2) to create, delete and change entities, attributes and associations of the IB, and also to evaluate OCL expressions over the IB. The assert statements are executed as follows:

- *Assert consistency*. The interpreter requests that the information processor check all static and temporal constraints. The result is the set of constraints that are not satisfied. The verdict is *Fail* if the set is non-empty.

- *Assert inconsistency*. The interpreter requests that the information processor check all static and temporal constraints. As before, the result is the set of constraints that are not satisfied. The verdict is *Fail* if the set is empty.

- *Assert event occurrence*. In the first step, the interpreter requests that the information processor check all static and temporal constraints. The result is the set of constraints that are not satisfied. The verdict is *Error* if the set is non-empty. In the second step, the interpreter requests the information processor to check the event constraints. The result is the set of event constraints that are not satisfied. The verdict is *Fail* if the set is non-empty. In the third step, the test interpreter executes the method of the *effect()* operation corresponding to the event (recall that the methods are written in the CSTL language, but without using assert statements). The test interpreter executes the statements of this method in the same way as those of the test programs: by invoking the services of the information processor. In the fourth step, the interpreter again requests the information processor to check all static and temporal constraints. The result is the set of constraints that are not satisfied. The verdict is *Fail* if the set is non-empty. Finally, the interpreter requests that the information processor check the event postconditions. The result is the set of postconditions that are not satisfied. The verdict is *Fail* if the set is non-empty.

- *Assert event non-occurrence*: In the first step, the interpreter requests that the information processor check all static and temporal constraints. The result is the set of constraints that are not satisfied. The assertion is *Error* if the set is non-empty. In the second step, the interpreter requests that the information processor check the event constraints. The result is the set of event constraints that are not satisfied. The verdict is *Pass* if the set is non-empty.

- *Assert IB contents*. In the first step, the interpreter requests that the information processor check all static and temporal constraints. The result is the set of constraints that are not satisfied. The assertion is *Error* if the set is non-empty. In the second step, the interpreter requests that the information processor evaluate the OCL expression(s), and the interpreter computes the verdict from the results obtained.

5.4 Coverage Processor

The design and the implementation of our *Test Processor* were extended in order to include automatic coverage analysis, according to the basic set of test adequacy criteria defined in Chapter 11. The formal details and notation of these test adequacy criteria are explained in detail in that chapter. In this section, we describe the *Coverage Processor*, which has been designed to be extended with other criteria that could be proposed in the future.

The *Preprocessor* initializes the *coverage database*. The role of this database is to maintain the set of covered and uncovered elements for each test adequacy criterion. Initially, each element is registered in the database and marked as uncovered.

When the modeler requests the execution of the test set, the *Test Manager* (Section 5.3.2) delegates its execution to the *Test Interpreter* (Section 5.3.3). The *Test Interpreter* communicates information about the tests execution to the *Adequacy Criteria Analyzer* which is the responsible of updating the coverage *database*. The update process is as follows:

Every time the *Test Interpreter* asserts the consistency of the IB, it communicates to the *Adequacy Criteria Analyzer* the set *VTC(TA)* of valid type configurations and the set *BaseTypes(TA)* that have valid instances in the current state of the IB. The analyzer marks as covered the valid type configurations included in *VTC(TA)*. It also marks as covered the entity types included in *BaseTypes(TA)*.

Every time the *Test Interpreter* requests the evaluation of a derivation rule to the *Information Processor*, the *Test Interpreter* communicates it to the *Adequacy Criteria Analyzer*. Each evaluated derived type is marked as covered.

Note that a derivation rule may be evaluated when an integrity constraint is checked, when evaluating other derivation rules, when asserting the contents of the IB or when an event occurs.

The test interpreter informs the *Adequacy Criteria Analyzer* when a valid domain event occurs. The analyzer marks as covered the asserted domain event type.

After the execution of all test programs, the *Adequacy Criteria Analyzer* queries the *Coverage Database* in order to obtain the sets of covered and uncovered elements for each criterion. The analyzer also computes some statistical information about the coverage results. All this information is used by the *Coverage Results* module of the *Presentation Manager* in order to show the results of the coverage analysis.

These results are only relevant when the global verdict of all the test programs is *Pass*.

Fig. 35 shows a screenshot of the coverage results provided during the execution of a test set execution on the conceptual schema example of a Civil Registry system. Note that the set of uncovered elements and an intuitive progress measure for each criterion is shown.

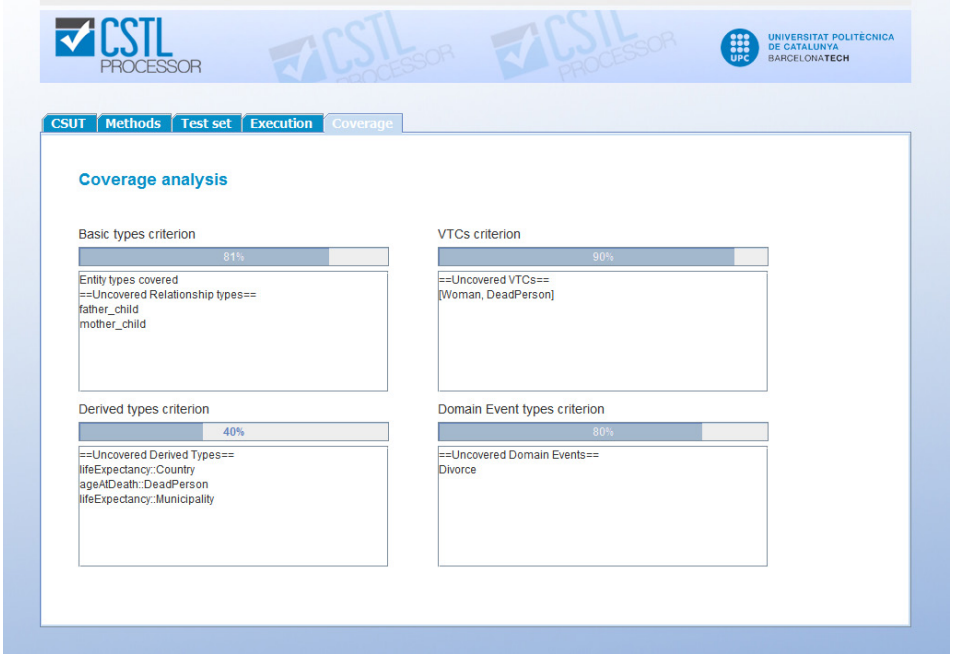


Fig. 35. Screenshot of coverage analysis in the CSTL Processor

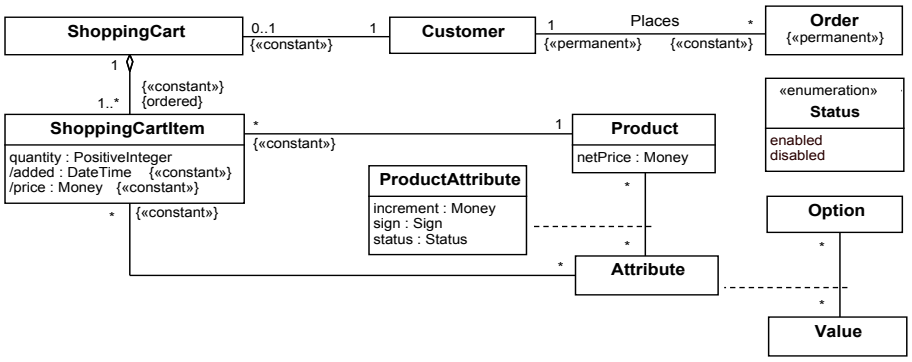
5.5 Testing Conceptual Schemas with Temporal Constraints and Derivation Rules

Many kinds of temporal constraints have been studied in the literature, and testing them poses particular problems. In this section, we explain the extensions of the *CSTL Prototype* in order to test schemas that use a representative set of constraints that involve two successive states of the IB (sections 5.5.1 and 5.5.2), and on creation-time constraints (section 5.5.3). We also explain (in section 5.5.4) how to test schemas with a particular kind of derived relationship types, which are similar to creation-time constraints.

The examples in this section are based on the schema fragment shown in Fig. 36, which is part of the *osCommerce* case study reported in Chapter 6.

5.5.1 Temporal Constraints on the Population of Entity Types

Two representative constraints on the population of entity types that can be evaluated taking into account two successive states of the IB are the constant and permanent entity types (Olivé 2007). An entity type is *constant* if its population is the same at all times, and *permanent* if its instances never cease to be instances of it. Most conceptual schemas include several entity types that are permanent. In the example of Fig. 36, *Order* is a permanent entity type because its instances never cease to be instances of it.




```
context ShoppingCartItem inv onlyOneAttributePerOption:
    self.attribute -> isUnique(option)

context ShoppingCartItem inv productHasTheAttributes:
    self.product.attribute -> includesAll(self.attribute)

context ShoppingCartItem::price:Money
    derive:
        if self.attribute -> isEmpty() then self.product.netPrice
        else
            self.attribute.productAttribute
                    -> select(pa|pa.product = self.product) -> collect
                    (if sign = Sign::plus then increment
                    else -increment
                    endif) -> sum() + self.product.netPrice
        endif

context ShoppingCartItem::added:Datetime
    derive: Now()
```

Fig. 36. Fragment of the osCommerce case study focusing on shopping cart items

When a conceptual schema includes one or more temporal constraints of the above kinds, then the semantics of the *assert consistency* and *assert inconsistency* statements defined in 4.4.3 and 4.4.4, respectively, must be extended as follows. In a test case, there is a consistent IB state every time that an *assert consistency* is executed and the obtained verdict is *Pass*. Note that such an assertion is executed in each of the assertions described in Section 4.4, except for *assert inconsistency* (Section 4.4.4). Once there is a consistent IB state in a test case, the next time that an *assert consistency* or *assert inconsistency* is executed in that test case, the constant and permanent constraints must be evaluated taking into account the current IB state and the previous consistent IB state.

For example, assume the following test case:

```
test DeletingAnOrder{
      c1 := new Customer;
      order1 := new Order;
      order1.customer := c1;
      assert consistency;
      delete order1;
      assert consistency;
}
```

The verdict of the first assertion will be *Pass*, but that of the second will be *Fail* and, therefore, the verdict of the test case will be *Fail*. If the domain experts confirm that the existing IB state in the second *assert consistency* is not valid, then the conceptual modeler may prefer to change that assertion to *assert inconsistency* in order to make the verdict *Pass*.

The above constraints can be evaluated in several ways. In our implementation, we simulate the execution of an operation between the two IB states. The operation has one postcondition for each temporal constraint. We may use the standard OCL in those postconditions in order to ensure that the constraints are satisfied. If *E* is a constant entity type, then the postcondition is as follows:

```
E.allInstances() = E.allInstances@pre()
```

where @pre is the OCL keyword to refer to the previous state. If *E* is a permanent entity type, then the postcondition is as follows:

```
E.allInstances() -> includesAll(E.allInstances@pre())
```

5.5.2 Temporal Constraints on the Population of Relationship Types

Two representative constraints on the population of relationship types (attributes, associations) that can be evaluated taking into account two successive states of the IB are the constant and permanent relationship types (Olivé 2007). A relationship type $R(p_1:E_1,...,p_n:E_n)$ is *constant* with respect to a participant p_i if the instances of R in which an instance e_i of E_i participates are the same during the temporal interval in which e_i exists. A relationship type $R(p_1:E_1,...,p_n:E_n)$ is *permanent* with respect to a participant p_i if the instances of R in which an instance e_i of E_i participates never cease to exist during the temporal interval in which e_i exists. Most conceptual schemas include several relationship types that are constant or permanent with respect to one of their participants.

In the example of Fig. 36, *Order* is constant with respect to its participation in *Places* because the instances of *Places* in which an order participates are the same during the temporal interval in which the order exists. The product and the attributes of a shopping cart item are constant because they remain the same during the temporal interval in which that item exists. *Customer* is permanent with respect to its participation in *Places,* because the instances of *Places* in which a customer participates never cease to exist during the temporal interval in which the customer exists.

As in the previous case, once there is a consistent IB state in a test case, the next time that an *assert consistency* or *assert inconsistency* is executed in that test case, the constant and permanent constraints must be evaluated taking into account the current IB state and the previous consistent IB state. In our implementation, the postconditions that we add to our simulated operation are as follows. If $R(p_1:E_1,...,p_2:E_2)$ is constant with respect to participant p_1, then:

```
E1.allInstances()-> intersection (E1.allInstances@pre)->
   forAll(e1|
e1.p2 = e1.p2@pre)
```

If $R(p_1:E_1, ...,p_2:E_2)$ is permanent with respect to a participant p_1 and the multiplicity of p_2 is 1 or 0..1, then:

```
E1.allInstances()-> intersection (E1.allInstances@pre)->
   forAll(e1|
e1.p2 ->includes(e1.p2@pre))
```

And if the maximum multiplicity is greater than 1, then:

```
E1.allInstances()-> intersection (E1.allInstances@pre)->
   forAll(e1|
e1.p2 ->includesAll(e1.p2@pre))
```

For example, assume the following test case:

```
test ChangingTheCustomerOfAShoppingCart{
      c1 := new Customer;
      sc := new ShoppingCart;

      sc.customer := c1;
      fashionTShirt:= new Product;
      fashionTShirt.netPrice := 10;
      sci1 := new ShoppingCartItem;
      sci1.shoppingCart := sc;
      sci1.product := fashionTShirt;
      assert consistency;
      c2 := new Customer;
      sc.customer := c2;
      assert consistency;
}
```

The verdict of the first assertion will be *Pass*, but that of the second will be *Fail* and, therefore, the verdict of the test case will be *Fail*. The customer of a shopping cart has been defined as constant, and therefore it cannot be changed. The postcondition that is not satisfied is as follows:

```
ShoppingCart.allInstances()-> intersection
   (ShoppingCart.allInstances@pre())
    -> forAll(sc| sc.customer = sc.customer@pre)
```

If the domain experts confirm the temporal constraint, then the conceptual modeler may prefer to change the last assertion to *assert inconsistency* in order to make the test *Pass*.

5.5.3 Creation-time Constraints

A creation-time constraint is a particular kind of temporal constraint that appears several times in most conceptual schemas. A creation-time constraint of an entity type *E* is a constraint that its instances must satisfy only at the time when they become an instance of that entity type. As proposed in (Olivé 2006) we define creation time constraints by means of operations stereotyped «iniIC» that must give a *True* result when the corresponding entity is created.

Consider, as an example, the schema of Fig. 36 and assume that the conceptual modeler adds the following constraint in order to enforce that the product attributes

of the attributes of a shopping cart item must be enabled when that item is created. The formal specification in OCL is as follows:

```
context ShoppingCartItem::productAttributesEnabled():Boolean
  body: self.attribute.productAttribute->forAll(status =
   Status::enabled)
```

where the operation *productAttributesEnabled* has the stereotype «iniIC».

In the following test case, the conceptual modeler declares that the schema allows changing the state of a product attribute once it has been used in the creation of a shopping cart item:

```
test changeProductAttributeStatus{
     color := new Option;
     black := new Value;
     blackColor := new Attribute(option:=color, value:=black);

     carInsurance := new Product(netPrice:=300);
     blackCarInsurance := new ProductAttribute
                          (product:=carInsurance,
                           attribute:=blackColor);
     blackCarInsurance.increment := 80;
     blackCarInsurance.status := Status::enabled;
     blackCarInsurance.sign := Sign::plus;

     c := new Customer;
     sc := new ShoppingCart(customer:=c);
     item1 := new ShoppingCartItem(shoppingCart:=sc);
     item1.product := carInsurance;
     item1.quantity := 1;
     item1.attribute := Set{blackColor};
     assert consistency;

     blackCarInsurance.status := Status::disabled;
     assert consistency;
}
```

The verdict of the first assertion is *Pass.* The verdict of the second assertion is also *Pass* because the constraint *productAttributesEnabled* is only evaluated when shopping cart items are created.

Creation-time constraints must be evaluated every time an *assert consistency* or *assert inconsistency* is explicitly or implicitly executed in a test case, taking into account the current IB state and the previous consistent IB state, if it exists.

The first time that the assertion is executed, the previous IB state is assumed to be empty. In the example, the constraint must be evaluated only for the shopping cart items that have been created since the execution of the previous assertion.

In our implementation, the evaluation of a creation-time constraint *ct* of entity type *E* is performed by adding the following postcondition to our simulated operation:

```
(E.allInstances() - E.allInstances@pre())
    -> forAll (e| ct replacing self by e)
```

In the example, the postcondition becomes the following:

```
(ShoppingCartItem.allInstances() -
   ShoppingCartItem.allInstances@pre())
   -> forAll (sci|sci.attribute.productAttribute
                 ->forAll(status= Status::enabled))
```

5.5.4 Derived Constant Relationship Types

A particular class of derived relationship type that often appears in many conceptual schemas is the derived constant relationship type, whose instances can be derived when the instances of one of its participants are created, and they remain fixed during their lifetime (Olivé 2003).

Fig. 36 shows two simple examples: attributes *ShoppingCartItem*::*added*, *price*. Their value is determined when an instance of *ShoppingCartItem* is created (using the derivation rules shown at the bottom of the figure), and those values do not change later on.

Consider as an example the following test case:

```
test priceDoesNotChange{
     p := new Product(netPrice:=15);
     c := new Customer;
     sc := new ShoppingCart(customer:=c);
     item1 := new ShoppingCartItem(shoppingCart:=sc);
     item1.product := p;
     item1.quantity := 1;
     assert equals item1.price 15;
     p.netPrice := 20;
     assert equals item1.price 15;
}
```

The verdict of the first assertion is *Pass* and the verdict of the second one is also *Pass* given that the attribute *price* is derived and constant.

Derived constant relationship types must be materialized when the corresponding entity is created, before the evaluation of the constraints. In our implementation, we materialize those types every time an *assert consistency* or *assert inconsistency* is explicitly or implicitly executed in a test case. The materialization needs to be done only for the entities that have been created since the execution of the previous assertion.

6

Case Studies on Conceptual Schema Testing

The approach to test conceptual schemas presented in Chapter 4 and its associated tool (Chapter 5) were tested and refined by applying them in case studies, taking advantage of the lessons learned when using them (Fig. 37). This research strategy is supported by the Design Science Research approach (Section 1.4), the main principles of which are adopted in this Thesis.

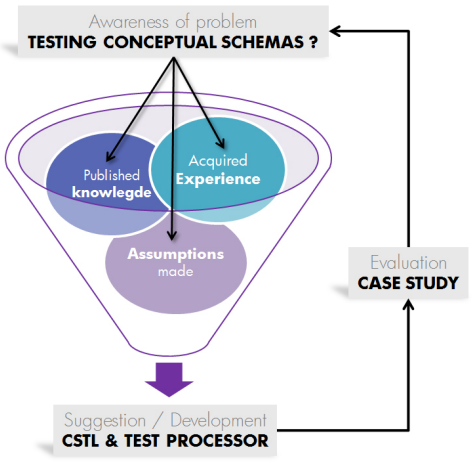


Fig. 37. Case studies in the context of Design Science Research

In this chapter, we present an overview of two case studies in which our conceptual schema testing environment (including the CSTL language and the CSTL Processor) was applied in practice. We also reference the reports in which the application results are detailed. Finally, we explain the lessons learned in each case.

In Chapter 9, more case studies on using our testing approach to perform Test-Driven Conceptual Modeling (Chapter 8) are reported.

6.1 The osCommerce Case Study

E-commerce is a professional domain of interest in the context of information systems. It allows people exchanging goods and services with no barriers of time or distance.

osCommerce (www.oscommerce.com) is an e-commerce solution available as free software under the GNU (General Public License). *osCommerce* project was started in March 2000 in Germany and since then, it has become the base of thousands of online stores around the world. *osCommerce* can be customized in order to operate in different countries (taking into account different languages, taxes, currencies, etc.) and can be used in several kinds of online stores. A conceptual schema of the *osCommerce* system was developed in (Tort 2007).

The objective of this case study was to test the conceptual schema of this system by writing a representative set of CSTL test cases and executing them by using the CSTL Processor. This conceptual schema specifies the structural knowledge and the main domain events of this real-sized system. It consists of 78 entity types, 202 attributes, 60 associations, 215 invariants and 216 event specifications.

The experience was performed with a preliminary version of both the CSTL definition and the tool. In this context, the purpose of the case study was twofold: (1) Analyzing the feasibility of writing and executing test cases for checking conceptual schema quality by using the CSTL language and the CSTL Processor, and (2) identifying improvement proposals and refinements to make progress in the development of the testing environment and the testing approach.

The testing process consisted in the specification and the execution of 162 test cases grouped into 35 test programs. Each test program was aimed at testing each knowledge group of the conceptual schema (Store Data, Configuration values, Payment methods, Shipping methods, Languages , Currencies, Location & Taxes, Products, Product attributes and options, Product categories, Specials, Manufacturers, Banners, Newsletters, Customers, Reviews, Shopping carts &

Orders). The resultant set of test cases consisted of 2752 lines of CSTL code. The test cases were inspired in real-world online shops which are based on *osCommerce*. Fig. 38 shows a fragment of a test program that was used in the experiment. Test cases were specified by using a preliminary version of the CSTL Language and Processor, which were refined according to the lessons learned.

```
testprogram PlaceAndOrder{

//The fixture (which is not reproduced here) contains the initialization of:
//Locations, currencies, languages, store configuration, default order status, stock, products,
//taxes, payment methods and shipping configuration
…

test placeAndOrder{

//Customer initialization
a:= new Address(country:=spain, zone:=catalonia, state:='Catalonia');
c := new Customer(address:=a,primary:=a);
//The customer logs in
ns:=new NewSession(currentLanguage:=english, currentCurrency:=euro) occurs;

/*
The customer adds to the shopping cart the following items: 2 standard laptops with no warranty, 1 Standard
laptop with Premium warranty, 1 Illustrated Start guide
*/
new AddProductToShoppingCart(session:=ns.createdSession,product:=standardLaptop,quantity:=2) occurs;

new AddProductToShoppingCart(session:=ns.createdSession,product:=standardLaptop,quantity:=1,
                             attribute:=premiumWarranty) occurs;

new AddProductToShoppingCart(session:=ns.createdSession,
                             product:=illustratedStartGuide,quantity:=1) occurs;

new LogIn(session:=ns.createdSession, customer:=c) occurs;

sc:=ns.createdSession.shoppingCart;
oc := new OrderConfirmation
            (shoppingCart:=ns.createdSession.shoppingCart, currency:=euro,
             shippingMethod:=sm, paymentMethod:=pm, billing:=a) occurs;
orderCreated:=oc.orderCreated;

assert equals orderCreated.orderLine.product->asSet()->size() 2;
assert equals orderCreated.orderLine->select(product=standardLaptop).quantity->sum() 3;
assert equals orderCreated.orderLine->select(product=illustratedStartGuide).quantity->sum() 1;
assert equals standardLaptop.quantityOnHand 297;
assert equals illustratedStartGuide.quantityOnHand 49;

/*
Order total details
=========================
          2 x standard laptop (no warranty)          x 949      =          1898,00
          1 x standard laptop (premium warranty)     x 1061     =          1061,00
          Subtotal ..........................................              2959,00
          VAT 16%............................................               473,44
          Total (16%)........................................              3432,44
          1 x illustrated start guide                x 15       =            15,00
          Subtotal ..........................................                15,00
          VAT 4%.............................................                 0,60
          Total (4%).........................................                15,6
          ---Shipping costs (Per Item)
          Handling fee ......................................                 5,00
          4 x Per Item Rate                          x 10       =            40,00

          Order Total _______________________________________              3493,04
*/
assert equals orderCreated.total() 3493.04;

//The store administrator can change the status of the order...
new UpdateOrderStatus(order:=orderCreated,newOrderStatus:=delivered) occurs;
assert equals orderCreated.orderStatus Sequence{pending,delivered};

//...or he can cancel the order (order information cannot be deleted)
new CancelOrder(order:=orderCreated) occurs;
assert equals orderCreated.orderStatus  Sequence{pending,delivered,cancelled};

}
}
```

Fig. 38. Fragment of a test program used in the osCommerce Case Study

The full details about the case study and the results (including the tested conceptual schema and the executed test programs) are detailed in the report (Tort 2009b).

6.1.1 Lessons Learned

In the following, we review some relevant lessons learned from the application of the testing approach to this case study.

Feasibility of the Conceptual Schema Testing Approach

The first version of the testing approach was based only on the state of the art on software testing and conceptual schema validation (Chapter 3). The successful application of the preliminary prototype version of the CSTL Processor in this case study gave us confidence about the feasibility of the testing approach. It also encouraged us to perform more experiences, such as the one referenced in the next section and those explained in Chapter 9.

Improvement Needs

The most significant challenge of this case study was analyzing if the test kinds defined in Section 4.2 were suitable to test a real-sized conceptual schema, like the one used in this case study.

Since each test kind has associated statements in the CSTL Language, we confirmed that these test kinds were suitable to test UML/OCL conceptual schemas with the constructs specified in Section 2.1.1. However, we detected some improvements proposals related to the CSTL language. Most of them were included in the CSTL language during the experience. The most relevant ones are:

- **The necessity of *fixture components* in order to reuse sets of CSTL statements (pieces of stories)**. The *fixture component* construct avoids repeating the initialization of shared states in different test cases that belong to the same test program. For example, in the context of this case study, it was very useful to encapsulate the instantiation of shopping carts, which were reused in different test cases. *Fixture components* and statements to call them were added to the CSTL language. The addition was limited to the scope of test programs. An additional feature would be adding *global fixture components* to be shared by all test programs of the test set.

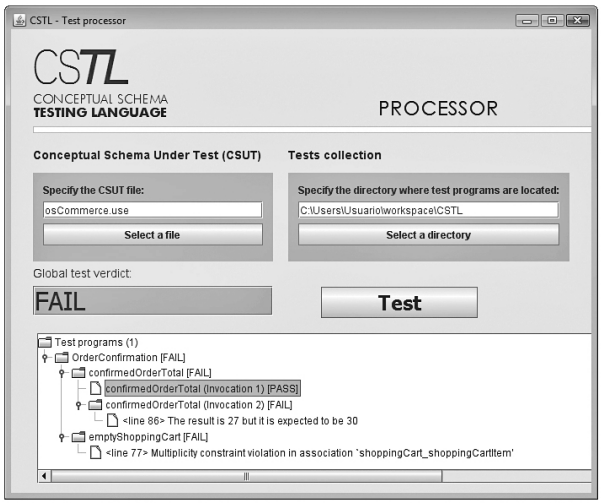


Fig. 39. Screenshot of the preliminary version of the CSTL Processor (osCommerce Case Study)

- **Creating states by direct instantiation.** In a test case, states may be reached by: (1) using the CSTL statements aimed at instantiating entity types and relationship types, or (2) asserting the occurrence of domain events. By applying CSTL to this case study, we conclude the following guidelines:

    - If the conceptual modeling objective is to define a correct and complete set of domain event types, then all states should be achieved by means of the occurrence of domain event types. If this rule is satisfied, then we are sure that all tested IB states may be reached by the define set of event types.

    - If we only plan to define a representative (and non-complete) set of relevant domain event types, then it is useful to define testing states by using the CSTL operations that allow building IB states.

We also detected some improvement needs related to the usability of the preliminary version of our tool (Fig. 39) in order to enhance the efficient application of the conceptual schema testing activity:

- **Edition of the test set and the conceptual schema from the CSTL Processor.** The preliminary version of the CSTL Processor only allowed modelers to execute test cases on conceptual schemas specified in USE. The definition of the conceptual schema and the definition of test cases were performed in external text editors. Since test case executions give

information to modify the conceptual schema and the test cases, we realized that providing integrated editors in the CSTL Processor tool would be a key usability improvement. Therefore, editors with syntax highlighting for creating and modifying the schema and the test set were developed.

- **Errors and failure information.** The error and failure information provided by the tool suggests changes to improve the quality of the schema. We used the experience acquired during the development of this case study in order to improve the error and failure messages.

Finally, many minor errors were also fixed in the CSTL processor as a result of testing the osCommerce conceptual schema.

The Necessity of Coverage

In this case study, we specified several test cases aimed at testing the existing conceptual schema specification of the explored system. One of the main difficulties experienced was keeping track of the specified tests and checking if they covered all elements of the schema. This fact suggested the necessity of analyzing the degree in which the set of developed test cases covered the elements of the conceptual schema under test. This is the rationale of the basic coverage proposal explained in Chapter 11. The present case study was complemented with the analysis of coverage for a subset of the osCommerce conceptual schema. The full details of the coverage application in this case study can be found in (Tort 2009a).

Mock Entities

By applying our conceptual schema testing approach to this case study, we realized that we were not able to test aspects that depend on unavailable external systems (e.g. the email service) or uncontrolled data (e.g. the current time or random data).

This is a limitation that has been also experienced in program testing, and it has been overcome by the use of *mock objects*. Mock objects "are simulated objects that mimic the behavior of real objects for testing purposes" (Hamill 2004). In conceptual schema testing, we can also define schema elements that simulate knowledge for testing purposes. In the following, we describe three examples about the use of mock objects in conceptual schema testing.

Testing time

Some test cases depend on time. For example, in the present case study, we can test that specials/offers may expiry in a given date. These test cases must preserve

the property that they can be executed as many times as needed regardless of time passing. Therefore, we realized that during the testing process we need to control the testing time instead of using the (uncontrollable) real time.

In order to set time as desired for testing purposes, we may represent the current testing date time as a property of a *mock entity*.

The used *mock entity* is an instance of a *mock entity type* that simulates the knowledge about the current time. Mock entities should be implemented in the developed system by external systems. In the case of time, the implementation derived from the conceptual schema will probably use the real system clock functions in order to implement the specified time knowledge.

External systems: The email case

In general, mock objects may be used to represent external systems that are not available during the testing process. In case we need to keep track of some knowledge about these external systems, we may define mock entity types to represent the required information for testing purposes.

As an example, assume that the system under development needs to save information about the emails sent by the system. Fig. 40 represents a mock conceptual schema fragment that can be used to define test cases considering this information. This mock schema fragment will be implemented in the final system by a real email service system.

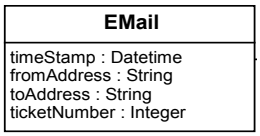


Fig. 40. Mock conceptual schema fragment example

Random instances

In general, we need to be careful when using random functions for testing purposes. Assertions that formalize expectations that involve random values may be invalid because (1) the results of computing two random values are not equal by definition, and (2) random values may vary in each test case execution.

In order to overcome this testing limitation, again we need to simulate random computation as mock functions that return a predefined constant which is controlled by the tester. By this way, we take control of random values for testing purposes.

6.2 The Magento Case Study

Magento (www.magentocommerce.com) is another popular e-commerce solution and, consequently, an interesting domain to develop its conceptual schema. The *Magento* system was made available in 2008 and it was developed by *Varien* company.

This system is widely used in the e-commerce business field (1.5 million downloads, more than 280.000 visits per day, etc.). *Magento* is presented as a "feature-rich, professional open-source e-commerce solution that offers merchants complete flexibility and control over the look, content, and functionality of their online store. Magento's intuitive administration interface contains powerful marketing, merchandising and content management tools to give merchants the powers to create sites that are tailored to their unique business needs".

The objective of this case study was to develop the conceptual schema of this system and to test it by writing and executing a representative set of CSTL test cases. The resulting conceptual schema consists of 218 entity types, 982 attributes, 255 associations, 703 invariants and 202 event specifications.

The experience used the revised version of the CSTL definition and the tool used in the case study reviewed in Section 6.1. In this context, the purpose of the case study was threefold: (1) Analyzing the feasibility of the conceptual schema testing approach by analyzing its use performed by a modeler not familiar with the CSTL language, (2) identifying improvement proposals and refinements to the CSTL Processor, and (3) analyzing the kinds of errors revealed during the testing process.

The testing process consisted in the specification of 30 test cases aimed at testing the main use cases of the system, which were previously prioritized. The resultant set of test cases consisted of 1140 lines of CSTL code. Test cases were inspired in real-world online shops which are based on the *Magento* system.

The full details about the case study and the results (including the tested conceptual schema and the executed test programs) are detailed in (Tort 2009a, Ramirez 2011). The study also includes a comparison between the *osCommerce* system and the *Magento* system performed by means of the comparison of their conceptual schemas.

6.2.1 Lessons Learned

In the following, we review some relevant lessons learned from the application of the testing approach to this case study.

Feasibility of the Conceptual Schema Testing Approach

Our conceptual schema testing language and its associated tool (refined according to the improvement proposals of the case study explained in Section 6.1) have been successfully applied in this case study. The application has been performed by a master project student with no previous experience about the testing environment. This fact confirms the feasibility of the approach in a real-sized case performed by an independent modeler.

Improvement Proposals to the CSTL Processor

Some improvement proposals have been proposed during the use of the CSTL Processor in this case study. We briefly summarize them in the following:

- **Specification of the executable conceptual schema in modules.** For large conceptual schemas, it would be interesting to encapsulate parts of the schema as modules, in order to be more manageable. In particular, it is suggested to provide functionalities to enable or disable schema modules for testing purposes (e.g. for testing only specific parts without considering other knowledge specified in the schema). We plan to include this functionality. However, it should be required that, at the end of the testing process, all test cases should pass with all modules enabled. Otherwise, some test cases could fail or contain errors because they did not consider all knowledge defined in the schema.

- **Select specific test programs to be executed.** The CSTL Processor executes all test cases that are included in the test set. It would be convenient to provide the possibility of executing only some test cases for better efficient executions.

- **Testing break points.** It would be interesting to set break points in test program specifications, and the possibility to explore the IB state at that point. This additional functionality would be interesting to provide better facilities in order to analyze failing and error causes.

- **Pre/postconditions failing trace.** When a pre or postcondition causes a failure or an error during the execution of a test case, the failing information could be complemented by including the evaluation trace of the failing pre/post expression.

Thanks to this case study application, other minor errors in the CSTL Processor have been fixed.

Analysis of Error Kinds

One of the main objectives of this case study was analyzing whether the testing process was able to correct requirements errors in the schema specification, in order to enhance its semantic quality.

During the testing phase, more than 400 errors and failures were tracked and categorized. The testing process was performed in two phases:

- **Conceptual schema specification in an executable form.** During the formal specification of the conceptual schema in USEx (Tort 2010), some errors were fixed. Most of the errors detected in this stage corresponded to UML/OCL syntactic errors, which were revealed by the USEx parser (Fig. 41). Other errors were detected by the semantic analyzer of this parser (referenced types that were not defined, expressions with incompatible resulting types, etc.). Finally, a few requirement errors were fixed by the modeler during the inspection process that was performed during the specification of the executable version of the schema.

- **Conceptual schema testing.** The main objective of conceptual schema testing is detecting and fixing requirements errors according to stakeholders' expectations formalized as test cases. When assisted by the execution of these test cases, more requirements errors are revealed as shown in Fig. 41. Moreover, fixing requirements errors implies modifying existing knowledge. These modifications also lead to fix new syntactic and semantic errors of the specification language definition.

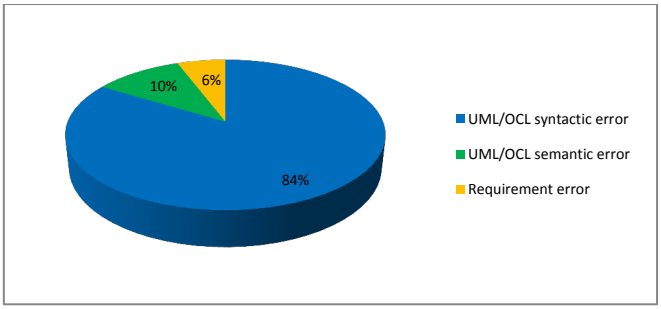


Fig. 41. Revealed errors during the executable specification of the conceptual schema

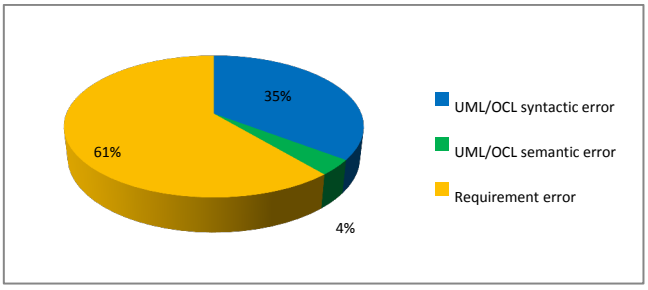


Fig. 42. Revealed errors during the conceptual schema testing process

The conceptual schema testing process also revealed conceptual improvement needs. These improvement needs are vital because, if not properly corrected, they may affect the semantic quality of the implemented system in the production environment.

Fig. 43 shows a screenshot of the real *Magento* system that illustrates an example of an error caused by a requirement error. This error was detected in this case study at the conceptual level, by applying conceptual schema testing. In this example, we show that, currently, the system does not consider countries and municipalities as entity types related between them and, consequently, inconsistent data is allowed by the system.

Fig. 43. Screenshot that illustrates a conceptual improvement need in Magento

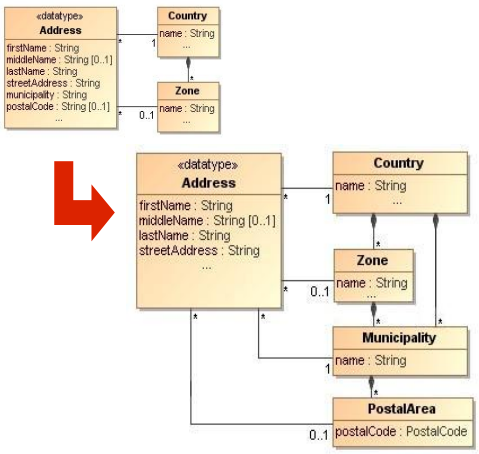


Fig. 44. Change in the schema of Magento in order to fix the improvement need of Fig. 43.

7 Related Work on Test-Driven Development

In this chapter, we review relevant state of the art regarding Test-Driven Development (TDD). The second main contribution of this Thesis, the Test-Driven Conceptual Modeling (TDCM) method (Chapter 8), lays on the basic principles of TDD.

In Section 7.1 we describe the main TDD activities. We also explain that regression testing is a basic property of a TDD environment (Section 7.2). Several languages and tools are aimed to support TDD. We review some of them in Section 7.3. In Section 7.4, existing studies about the effectiveness of TDD are presented. Finally, in sections 7.5 and 7.6 we discuss Acceptance TDD and the relationship between TDD approaches and modeling.

7.1 Test-Driven Development

Test-Driven Development (TDD) (Janzen et al. 2005, Beck 2003, Astels 2003, Koskela 2007) is a software development method in which: (1) an exhaustive suite of test programs is maintained; (2) no code goes into production unless it has associated tests; (3) tests are written first; and (4) the tests determine what code needs to be written (Astels 2003).

Throughout software engineering history (Sommerville 2010), several development process paradigms have been proposed. The traditional *waterfall process model* was conceived as a sequential development process consisting of five phases: Requirements analysis, design, implementation, testing and maintenance. After that, iterative paradigms such as the *spiral model* or the *incremental model* were proposed to allow more flexible development processes. In 1998, eXtreme Programming (XP) was introduced as an agile process based on very short development iterations.

TDD emerged in conjunction with the rise of agile paradigms for software development (Janzen et al. 2005), although some authors claim that test-first approaches were informally used as early as 1950s in the NASA's *Project Mercury* (Larman et al. 2003). Most popular development methodologies, like the Unified Process (UP), promote the use of agile practices.

XP is described in the *Agile manifesto* (Beck et al. 2001). It is based on the principle that the "highest priority is to satisfy the customer through early and continuous delivery of valuable software" and that changing requirements are "welcome, even late in development. Agile processes harness change for the customer's competitive advantage". TDD is considered to be a key practice of XP (Beck 2003). Its particularity is that testing plays a central role in the development process, instead of being a validation phase at the last stages of the development. TDD is a test-first approach, in contrast with the traditional test-last approaches.

In TDD, software is developed in short iterations. In each iteration, the developer: (1) Writes a test for the next bit of functionality that he or she wants to add; (2) Writes the functional code until the test passes; and (3) Refactors both new and old code to make it well structured.

Write a Test

For each bit of functionality to be implemented, the developer first writes a test that should pass. (Astels 2003) explains that developers should program tests by intention: "write tests without worrying about what classes or methods you will need to add". In other words, code that will test a required functionality is written before implementing the functionality itself. Note that, in this context, a test formalizes an implementation objective to be achieved.

Write Code to Pass the Test

After writing a non-passing test, its execution gives information about the failure. This information can be used to add code to pass the test.

TDD prescribes that changes in the code must be the minimum ones to make the test pass. According to (Astels 2003), "you write only enough to pass the test, no more. That means that you do the simplest thing that could possible work". Changing the code as a reaction to a failing test, allows validating the code changes just after the change is made. The reason is that we have the test and we can execute it immediately after a change.

Writing a test and changing the code to pass the test are the main activities to make progress in TDD: "you write a little bit of test, followed by just enough code to make that test pass, then a bit more test, and a bit more code, test, code, test, code, etc." (Astels 2003).

Refactoring Code

Refactoring is the process of making changes to existing working code without changing its external behavior in order to improve the internal structure. In other words, refactoring means to "clean code that works" (Beck 2003).

In TDD, the incremental development in small iterations raises the necessity of refactoring. TDD promotes refactoring when a code smell (Van Emden et al. 2002) is detected. The concept of code smell is widely used by the XP community to refer to characteristics of code that indicate less than acceptable code quality. Duplications, too large methods, or unclear code are typical examples of code smells.

Although much research work has been devoted to identifying and catalog software anomalies (Mens et al. 2004), some recent work proposes to extend the smell concept to models. Refactoring smells and patterns to fix them have also been proposed for UML/OCL schemas (a representative set of papers is (Sunyé et al. 2001, Zhang et al. 2005, Judson et al. 2003, Porres 2003, Correa et al. 2004)).

7.2 Regression Testing in TDD

Regression testing is about executing tests repeatedly throughout the project in order to verify that defects that once existed and were fixed do not reappear as the software evolves (Koskela 2007). In TDD, tests are not thrown away and they are used in regression testing in order to increase confidence about the already done work.

After a significant change in the code is performed, the developer may run the previous test set to detect if any previous passing test fails after the change. If some previous test case ceases to pass, then the failure information provides help to find

out the failure cause. Regression testing implies constant test feedback and it allows detecting errors in code as soon as changes cause them.

7.3 TDD Languages and Tools

Tools have played a decisive role in the emergence of TDD, which assumes the existence of an automated testing framework (Janzen et al. 2005). Automated tests (Meyer 2008) are fundamental to allow efficient regression testing (see Section 7.2). An automated testing framework consists of, at least, a formal language to specify tests and an interpreter to automatically execute them as many times as needed.

Developers have created languages and tools to support TDD. The most popular TDD framework is *JUnit* (Gamma et al. 1999, Glover 2007). *JUnit* is a Java-based language which includes statements to write test cases with assertions. *JUnit* tools allow writing, managing and executing such test cases, and they provide information about the results of the execution and feedback to identify failing tests. *JUnit* is open source and the software community has developed extensions to provide wide-reaching solutions to the challenge of testing (Astels 2003). By extension, other similar frameworks have been developed to support *JUnit* tests in other programming languages. All of them are generally referred as *xUnit* tools.

Other tools can be used in conjunction with *xUnit* tools to help practicing TDD. Most of them are aimed to analyze test coverage (Zhu et al. 1997). *Jester* (Moore 2001) is a popular tool for finding untested code by applying mutations. It makes systematic changes to the application's source code and runs the test suite. If all the tests still pass after making the change, then Jester reports that something in the code has not been tested. *NoUnit* (NoUnit team 2010) is another example that analyzes, for each method, whether it's called directly, indirectly or not at all from any test method. *Clover* (Atlassian Pty 2010) is a classic coverage tool that, in runtime, measures what code statements are executed and how many times. As far as we know, in the literature there have not been made proposals of adequacy criteria for testing conceptual schemas. In Chapter 11, we explore this challenge in order to enhance the quality checking process promoted by our conceptual schema testing approach and the TDCM method (Chapters 4 and 8).

JUnit can be integrated as a plug-in into *Eclipse* (Eclipse Foundation 2010), a widely-used Integrated Development Environment (IDE). Eclipse also provides refactoring functionalities and allows adding plug-ins to enhance the TDD practice.

7.4 TDD Effectiveness

The analysis of the effectiveness of Test-Driven Development (TDD) is essential to demonstrate the applicability and the usefulness of TDD in practice. Table 2 summarizes the main advantages and drawback of TDD, according to the literature reviewed in this Section. Analyzing the advantages and drawbacks of TDD is important to determine whether it is indicated or not to apply this method in a particular development project.

(Beck 2003), the primary author of the *Agile Manifesto*, explains that TDD is a way of managing fear during programming. In this context, he expresses the benefits of TDD as follows: "Instead of being tentative, begin learning concretely as quickly as possible. Instead of clamming up, communicate more clearly. Instead of avoiding feedback, search out helpful, concrete feedback".

(Koskela 2007) expresses that "TDD helps us speed up by reducing the time it takes to fix defects". He also explains that TDD "makes sure that there is practically no code in the system that is not required –and therefore executed- by the tests. […] TDD effectively guarantees that whatever you have written a test for works".

Several researches have conducted studies on the effectiveness of TDD practices. Although a few of them conclude that there seems to be no substantial improvements by applying TDD in contrast with other traditional approaches (Pancur et al. 2003, Muller et al. 2002), most of the studies contribute to analyze the strengths of TDD and to point out its challenges. However, each study uses different evaluation parameters and experimentation contexts, and most of them consider that more reliable evaluations are needed to reach more solid conclusions.

(Maximilien et al. 2003) report a case study conducted at IBM. The case study is the result of the development of a "non-trivial software system using TDD". The conclusions of the study state that "using this practice, we reduced our defect rate by about 50 percent compared to a similar system that was built using an ad-hoc unit testing approach".

(Janzen et al. 2008) conducted an industrial experiment to analyze the influences of TDD on design. Code size, complexity, coupling and cohesion are analyzed. The experiment does not achieve definitely conclusions about coupling and cohesion. However, the experiment shows that test-first programmers "tend to write smaller, simpler classes and methods".

(Edwards 2003) explain the results of a study about the use of TDD in an educational context. Results indicate that "students scored higher on their program assignments while producing code with 45% fewer defects per thousand lines of code".

TDD Advantages	TDD Drawbacks
Increasing code quality: Evolving a set of test cases and making them *Pass* increases code quality since code is added by fixing errors. (Janzen et al. 2005, Beck 2003, Astels 2003, Koskela 2007, Maximilien et al. 2003, Janzen et al. 2008, Edwards 2003, George et al. 2003)	**Stakeholders' availability:** Stakeholders (the source for test cases) may not be available during the implementation phase. (Stephens et al. 2003)
Focused objectives: The TDD cycle promotes stating and achieving focused objectives, instead of worrying about the growing whole program at any time. Solving small problems, validating them and moving forward is a more predictable way to develop. (Janzen et al. 2005, Beck 2003, Astels 2003, Koskela 2007, Janzen et al. 2008, Stephens et al. 2003, Stephens et al. 2010)	**Tests as specifications:** TDD is "code-centric" and lacks of focus on design and analysis. Tests are not substitutes of analysis and design specifications. (Stephens et al. 2003, Stephens et al. 2010, Boehm 2002)
Keeping code healthy with refactoring: Refactoring improves the design after adding new code. (Janzen et al. 2005, Beck 2003, Astels 2003, Koskela 2007, Janzen et al. 2008, Stephens et al. 2003, Stephens et al. 2010)	**Writing test cases is time consuming:** Writing test cases consumes time, although testing is essential before or after coding. (Maximilien et al. 2003, George et al. 2003)
Making sure that previous code still works: Automated regression tests help making sure that code that worked, still works after adding new code. (Janzen et al. 2005, Beck 2003, Astels 2003, Koskela 2007, Janzen et al. 2008, Stephens et al. 2003, Stephens et al. 2010)	**Rewriting tests:** When new code is added, previous tests may need to be rewritten, and it is time consuming. (Stephens et al. 2003)
Getting continuous feedback: Code is driven by the feedback provided in test errors and failure information. Therefore, code is evolved as a reaction to continuous feedback. (Janzen et al. 2005, Beck 2003, Astels 2003, Koskela 2007, Janzen et al. 2008, Stephens et al. 2003, Stephens et al. 2010)	**Tests origin:** TDD assumes that developers write tests directly from stakeholders, but does not consider previous analysis or design artifacts. (Stephens et al. 2003, Stephens et al. 2010)
Maintaining workable software at each iteration: After each iteration, you have code that works according to a set of tests. (Janzen et al. 2005, Beck 2003, Astels 2003, Koskela 2007, Janzen et al. 2008)	
Increasing programmer confidence: Code is testable as soon as it is written. Together with the previous properties, it increases programmers' confidence. (Janzen et al. 2005, Beck 2003, Astels 2003, Koskela 2007, Janzen et al. 2008, Geras et al. 2004, Lui et al. 2005, Madeyski et al. 2007)	

Table 2. Summary of TDD advantages and drawbacks

(Geras et al. 2004) empirically support that TDD can increase confidence during the development process. This study states that test-last processes may be less predictable than test-first approaches. Another conclusion is that TDD induces developers to create more tests and to execute them more frequently. Other studies such as (Lui et al. 2005, Madeyski et al. 2007) suggest similar benefits.

Several studies conclude that TDD has the potential of increasing the quality of code and the level of testing, although it may increase the time for development.

(George et al. 2003) report an experiment with 24 professional pair programmers. One group developed code using TDD while the other used a waterfall-like approach. The conclusions of the experiment state that "we found that the TDD developers produced higher quality code, which passed 18% more functional black box test cases. However, TDD developer pairs took 16% more time for development".

Other TDD evaluations, such as (Canfora et al. 2006, Bhat et al. 2006) reach similar conclusions.

7.5 Acceptance TDD

Originally, TDD only assumed the use of unit tests in order to verify and validate small bits of functionality. More recently, several researches realized that TDD approaches can enhance its contribution to requirements validation by using acceptance tests (Mugridge 2008, Melnik et al. 2006, Melnik et al. 2007, Sauvé et al. 2008).

An increasing number of organizations are interested in binding requirements and testing more closely together (Mugridge 2008, Martin et al. 2008, Uusitalo et al. 2008). Acceptance tests are a kind of tests specifically conducted to determine whether or not a system meets the customer needs.

Acceptance tests can also be written, automated and run in order to guide software development by applying the TDD principles. In the literature, this practice is named Acceptance Test-Driven Development (ATDD), Storytest-Driven Development or Customer Test-Driven development. In ATDD, user stories are written as automated tests.

The main difference between traditional TDD and ATDD is that ATDD tests are more customer-oriented and they are specifically designed to validate requirements. (Melnik et al. 2007) state that ATDD is a variant of TDD that "makes it possible to formalize the expectation of the customer into an executable and readable contract that programmers follow in order to produce and finalize a working system". (Koskela 2007) argue that in ATDD, tests should be used as a shared language that forces to transform ambiguous requirements into executable tests. He also summarizes that ATDD is about "specifying by example", which can be "a natural fit for our intuition and makes it easier to relate requirements to the concrete word and our software".

It is important to remark that the main difference between TDD and ATDD relies on the purpose and the scope of tests. Therefore, the same testing and language tools that support regular TDD can be directly used or extended in order to support ATDD (Mugridge 2008, Deng et al. 2007, Sauvé et al. 2006).

7.6 TDD and Modeling

Agile development methods were originally conceived in contrast with plan-driven development approaches that stress more formal specification of requirements as a base for design and implementation. (Beck et al. 2001) argues that agile methods are a reaction to "documentation driven, heavyweight software development process".

Nevertheless, many authors advocate that plan-driven development and agile development can and should be used in conjunction. The main aim of agile development is not avoiding documentation and models but "efficiently respond to changes". In other words, documents and models can also contribute to efficiently respond to changes during software development, especially in Acceptance TDD approaches.

In the eXtreme Programming community has raised the concept of *Agile Modeling* (AM). AM is a practice-based methodology for effective modeling and documentation of software-based systems. AM proposes a set of values, principles, and practices for applying modeling in software development projects in an effective and light-weight manner (Ambler 2002). Fundamental practices of AM include, for example, creating several models in parallel and iteratively, applying the right artifacts for the situation, being focused on the active participation of stakeholders, using models for communication, or modeling in small increments (Erickson et al. 2005). The TDCM method proposed in this Thesis (Section 8) is aligned with many of these guidelines. In this approaches, user stories and class diagram sketches are promoted to "think before you act". (Astels 2003) concludes that "experience shows that we can and should model on a project taking a TDD approach".

(Boehm 2002) expresses that "both agile and plan-driven methods have a home ground of project characteristics in which each clearly works best, and where the other will have difficulties" and he concludes that "hybrid approaches that combine both methods are feasible and necessary. […] Organizations must carefully evolve toward the best balance of agile and plan-driven methods". (Meyer 2008) states that "TDD, given prominence by agile methods, has brought tests to the center stage, but sometimes with the seeming implication that tests can be a substitute for specifications". He argues that "tests, even a million of them, are instances; they miss the abstraction that only a specification can provide", so that "tests are not substitutes for specifications".

(Sangwan et al. 2006) analyzes the application of TDD in geographically distributed development teams in large projects. They express the necessity of enforcing communication during the development process because "the TDD process works

well for projects in which a collocated team develops a small to medium system, but it can be challenging for large systems, especially those involving geographically distributed teams".

The TDCM method presented in this Thesis (Chapter 8) contributes to agile development because requirements defects can be more efficiently detected during the conceptual modeling activity in an incremental process, which is aligned to stakeholders' needs and expectations.

8

An Approach to Test-Driven Conceptual Modeling

In this chapter, we present the second main contribution of the Thesis: a novel method for the test-driven development of conceptual schemas that we call Test-Driven Conceptual Modeling (TDCM) (Tort et al. 2011a). TDCM is based on the principles of Test-Driven Development (TDD) (see Chapter 7).

Section 8.1 introduces the TDCM method and shows how to develop conceptual schemas using it. We explain the details of the TDCM cycle (Section 8.1.1), and two important guidelines (Section 8.3) on how to apply the method in some circumstances. Furthermore, we illustrate the method with its application to a fragment of the well-known Meeting Scheduler case study (Van Lamsweerde 2009) (Section 8.2). Finally, we present an analysis of the conjectured advantages and drawbacks of TDCM (Section 8.5).

8.1 Fundamentals of TDCM

In Chapter 4, we described an approach to test executable conceptual schemas written in formalized languages like UML/OCL. We also presented a tool to support the application in practice of the approach (Section 5). Therefore, since testing conceptual schemas is feasible, the following questions arise:

- Could we use conceptual test cases to drive the conceptual modeling activity?

- Could we improve the completeness and correctness of conceptual schemas by developing them using a TDD-based approach?

The Test-Driven Conceptual Modeling (TDCM) method that we present in this chapter is our answer to these questions.

TDCM is an iterative method aimed to drive the elicitation and definition of the conceptual schema of an information system. In TDCM, a system's conceptual schema is obtained by performing three kinds of tasks: (1) Write a test the system should pass; (2) Change the schema to pass the test; and (3) Refactor the schema to improve its qualities.

TDCM is a test-first approach in which conceptual schemas are incrementally defined and continuously validated. As far as we know, according to the state of the art reviewed in Section 7, this is the first work that explores the use of TDD in conceptual modeling.

According to the approach proposed in Chapter 4, a test case written in a conceptual schema testing language is an executable concrete story of a user-system interaction. A test case also specifies user expectations formalized as test assertions. The verdict of a test case is *Pass* if the conceptual schema includes the general knowledge to meet these user expectations. Otherwise, error and failing information is provided in order to point out why the conceptual schema does not meet the formalized expectations.

The main rationale of TDCM is the evolution of the schema by continuous fixing of testing errors and failures. The failing and error information obtained by writing and executing tests during the conceptual schema development provides feedback in order to correct the schema. This feedback promotes incremental changes in the schema and continuous improvement of its semantic quality according to a test set.

As explained in Section 1.4.2, the problem we try to solve is significant because each information system development project requires the development of its conceptual schema, and correctness and completeness are two fundamental quality properties of conceptual schemas (Olivé 2005). On the other hand, it is widely recognized that errors at the conceptual level should be detected and corrected as soon as possible.

TDCM is assumed to be used in the context of a requirements engineering (RE) method. This method determines the source artifacts available when TDCM begins, the iteration sequence, and the condition when TDCM ends. In this chapter, we define the TDCM method regardless the RE method in which it is applied and, in Chapter 10, we provide some details of the integration of TDCM into four well-known RE methods.

In general, TDCM contributes to the evolution of conceptual schemas. If the starting point is an empty schema, then full conceptual modeling process may be supported by TDCM. Otherwise, we may evolve existing (partial) schemas or some critical parts of it.

Test cases that drive the application of TDCM are designed from artifacts provided by the general method. TDCM obtains an executable conceptual schema and a resulting test set that validates the schema. The resulting conceptual schema may be used as a source for the next activities specified by the general method.

The environment for testing conceptual schemas (Chapter 4) and the *CSTL Processor* tool (Chapter 5) support the application of the TDCM method in practice. By using them, we have evaluated our method by means of four cases studies, which are discussed in Chapter 9.

8.1.1 TDCM Cycle

In TDCM, a conceptual schema is defined incrementally in short iterations. The iterations that must be performed and the objective of each test case depend on the RE method in which TDCM is used.

An iteration starts by adding a *new test case* to the passing test set of the previous iteration (*previous test set*)[1]. The *previous test set* is empty when the first iteration is initiated.

The objective of each iteration is to change the schema so that it includes the knowledge to correctly execute the new test case.

The *previous test* set in addition to the *new test case* is the *current test set* of the iteration. A TDCM iteration can only finish when the overall verdict of the *current test set* is *Pass*.

[1] We follow here the TDD principle of defining one test case in each iteration, but in practice nothing prevents defining more than one, if so desired.

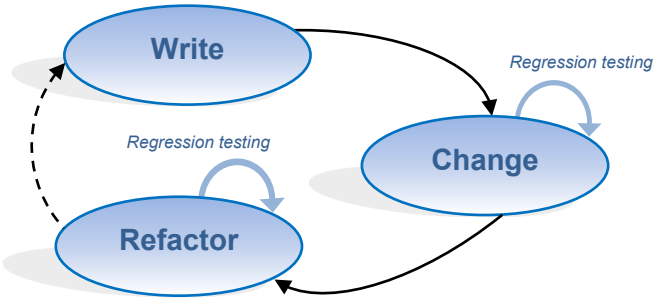


Fig. 45. The TDCM cycle

Fig. 45 shows the TDCM cycle and the order of its tasks. A TDCM iteration is a particular instantiation of the TDCM cycle. The TDCM cycle consists of three kinds of tasks:

1. Write a test case (which is expected to pass).
2. Change the schema (to pass the test case).
3. Refactor the schema (to improve the knowledge representation, if needed).

Write a Test Case

The first task of the TDCM cycle consists in setting up a new test case whose verdict will be *Pass* when the conceptual schema includes the knowledge to be added in the current iteration. In the general case, the schema does not include that knowledge initially, and therefore the verdict will be *Error* or *Fail*. At the end of the current iteration, the verdict will be *Pass*.

If the verdict of the *new test case* is *Error* or *Fail*, then the *new test case* is suitable to make progress in the TDCM cycle. If the verdict of the *new test case* is *Pass,* then the iteration objective is already achieved and, consequently, the iteration finishes. Such iterations are sometimes convenient to increase the confidence about already defined knowledge by executing complementary test cases.

Change the Schema

Changing the schema to make the verdict of the *new test case Pass* is the focused objective to be achieved by the conceptual modeler in this task. The testing

environment provides information about the failure or the error. The next change to be done in the schema is fixing this error or failure.

The changes made in the schema can be validated by the execution of the *new test case* after each change. Changing the schema and checking if the verdict of the *new test case* becomes *Pass* is the main activity to make progress in TDCM.

If the verdict of the *new test case* is *Error,* then the required knowledge is not defined in the schema. Information about the error helps to find out required knowledge to be added in the schema.

Table 3 summarizes the interpretation of the verdict of the current test set (CTC) and the regression test set (RTS) at each iteration. This interpretation drives the application of TDCM in order to evolve the Conceptual Schema Under Development (CSUD).

Verdict	Current test case (CTC)	Regression test set (RTS)	
ERROR	Relevant knowledge needs to be added to the CSUD	*Schema element removed*	The CSUD becomes incomplete and the deleted schema element needs to be restored
		Relevant constraint added	Modify inconsistent states to maintain consistent test cases
FAIL	The knowledge defined in the CSUD needs to be corrected according to the asserted expectations	*Both CTC and RTS may Pass without changing them (only changing the CSUD)*	The knowledge defined in the CSUD needs to be corrected
		Neither CTC nor RTS can Pass without changing them	Inconsistent requirements
PASS	The CSUD has the necessary knowledge and satisfies the asserted expectations formalized in CTC	The CSUD still has the necessary knowledge and satisfies the asserted expectations formalized in RTS	

Table 3. Failure/error/pass interpretation

If the verdict of the *new test case* is *Fail,* the schema needs to be changed because it does not produce the expected result (the schema is not correct according to the test assertions). The information about the failure provided by the testing

environment helps to find out the changes to be done in the schema in order to fix the failure.

While changing the schema, test cases of the *previous test set* can be automatically executed as regression tests. The purpose of regression testing is to detect if the verdict of any of the previous test cases ceases to be *Pass*. In this case, a collateral effect on previous defined knowledge of the schema is detected. The problem to be solved is indicated by the failure/error information. Note that test sets written in a conceptual modeling language designed in the xUnit fashion (like CSTL) are automated.

If the verdict of the *previous test set* becomes *Fail*, then two cases are possible:

- *Previous defined knowledge ceases to be correct due to the last changes:* A derivation rule ceases to derive information as expected, an event occurrence ceases to produce the expected IB state or a domain event ceases to be applicable as it was expected. The conceptual schema needs to be changed to fix the failure.

- *Inconsistent requirements are revealed:* If it is not possible to change the schema to pass both the *new test case* and the previous failing test cases, an inconsistency between requirements has been revealed. The inconsistency must be resolved. Involved test cases may be changed, if needed, to reflect updated expectations[2].

If the verdict of a previous test case becomes *Error*, then two cases are possible:

- *The conceptual schema has become incomplete:* A schema element that was necessary for a previous test case has been removed.

- *The IB state of a previous test case has become inconsistent:* IB states of previous test cases may become inconsistent when a new constraint is added to the schema. Inconsistent states need to be updated in order to ensure that test cases formalize consistent stories. In Section 8.3, two guidelines are proposed to minimize this case.

[2] The choice of what to do with the revealed inconsistency depends on the inconsistency handling strategy followed in the project NUSEIBEH, B. and EASTERBROOK, S. The process of inconsistency management: A framework for understanding, p. 364-368. In this work, we assume that inconsistencies must be solved when detected, but it would be interesting to explore the use of TDCM with other strategies, specially the one that differs the resolution and continues the development until such time deemed appropriate to re-visit the inconsistency.

The TDCM method is adaptable to the experience of the conceptual modeler. Smaller changes and more frequent checking of the verdict of the *new test case* provide more failing/error information to guide the definition of the schema.

More regression testing provides more confidence about the already included knowledge. It is important to remark that TDCM encourages continuous reflection about the knowledge of the schema and its alignment with user expectations.

When the whole verdict of the *current test set* becomes *Pass*, the iteration objective is achieved and the conceptual modeler can proceed to the next task.

Refactor the Schema

Refactoring aims to improve the quality of the conceptual schema without changing the knowledge specified in it. Much research work has been devoted to identify and catalog software anomalies (Mens et al. 2004). Refactoring smells to detect possible anomalies and refactoring patterns to fix them have also been proposed for UML/OCL schemas (a representative set of papers is (Sunyé et al. 2001, Zhang et al. 2005, Judson et al. 2003, Porres 2003, Correa et al. 2004)).

When a conceptual schema is defined in an iterative way, as proposed in TDCM, refactoring may be applied to improve the quality of the schema. TDCM encourages the conceptual modeler to refactor the schema and, in sequence, request the execution of the *current test set*.

If the verdict of the *current test set* becomes *Fail* or *Error*, then we realize that the knowledge of the schema has not been preserved by the refactoring process. The failure/error information provided by the testing environment helps to identify the invalid refactoring changes. If the verdict of the *current test set* is *Pass* and no more refactoring is felt to be needed, then we can start a new iteration.

8.2 Example

We now illustrate TDCM by means of an example based on the *Meeting Scheduler* system described in (Van Lamsweerde 2009). Assume that we have already developed the conceptual schema shown in Fig. 46. Three domain events types are included in this initial schema (two of them are graphically shown in Fig. 46). Their effect is informally described in the following:

- *MeetingRequest*: A new instance of the entity type *Meeting* is created with the indicated subject, the set of possible dates, the user that convenes the

meeting (*initiator*) and the users invited to participate in the meeting (*invitedParticipant)*.

- *ConstraintsSpecification*: A user specifies his/her excluded dates for a meeting. This information is registered in the IB as an instance of *ConstraintsSet.*
- *UserCreation*: A new *User* is created with his/her *name* and *eMail*.

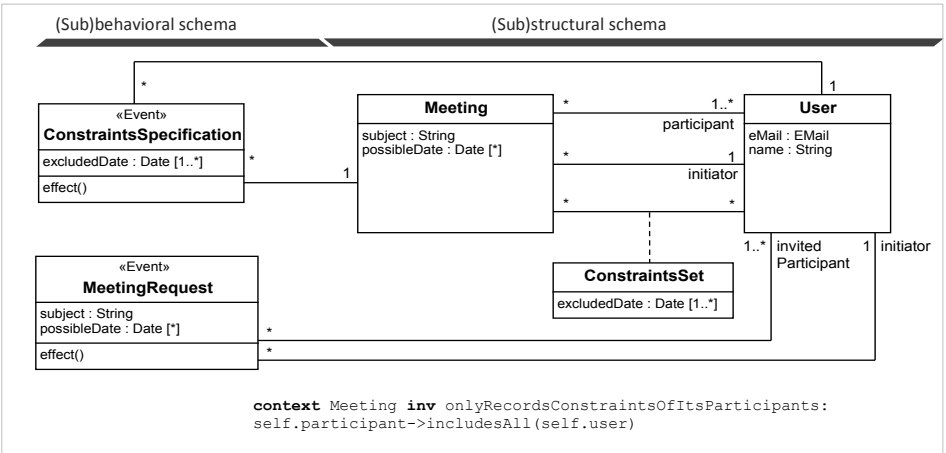


Fig. 46. Initial conceptual schema fragment of the Meeting Scheduler system

As an example, we show the formal specification of the postcondition (in OCL) and of the method (in CSTL) of the event *MeetingRequest*:

```
context MeetingRequest::effect()
 post:  (Meeting.allInstances() - Meeting.allInstances()@pre)
        -> one(m | m.subject = self.subject and m.possibleDate=self.possibleDate and
                   m.participant = self.invitedParticipant and
                   m.initiator = self.initiator)


method MeetingRequest::effect(){
  self.createdMeeting := new Meeting(subject := self.subject,
                                     possibleDate := self.possibleDate,
                                     participant := self.invitedParticipant,
                                     initiator := self.initiator);
}
```

In the following, the application of TDCM is illustrated by means of two example iterations. The purpose of these iterations is to make progress in the elicitation and the definition of the conceptual schema fragment of Fig. 46.

8.2.1 First Iteration Example: Invalid Meeting Scheduling

Consider the following informal requirement:

"A meeting cannot be scheduled on a conflicting date"

Using TDCM, the first task consists in writing a new test case whose verdict is *Pass* if the conceptual schema satisfies this requirement. In this case, we write the test case *invalidMeetingScheduling*, included in the test program *MeetingScheduling* shown below. Note that the fixture creates two users that will appear in the test case.

```
testprogram MeetingScheduling{

   sarahCreation := new UserCreation(name:='Sarah',
                                     eMail:='sarah@sarah.edu');
   assert occurrence sarahCreation;
   sarah := sarahCreation.createdUser;

   davidCreation := new UserCreation(name:='David',
                                     eMail:='david@david.edu');
   assert occurrence davidCreation;
   david := davidCreation.createdUser;

   test invalidMeetingScheduling{
    mr := new MeetingRequest(subject:='Strategy',
          initiator:=sarahCreation.createdUser,
          possibleDate:=Set{01-13-2011, 01-15-2011, 01-16-2011},
          invitedParticipant:=Set{sarah, david});
    assert occurrence mr;

    assert true mr.createdMeeeting.participant=
                Set{sarah, david};

    sarahConstraints := new ConstraintsSpecification(
                                meeting:=mr.createdMeeting,
                                user:=sarah,
                                excludedDate:=Set{01-15-2011});
    assert occurrence sarahConstraints;

    mrs := new MeetingScheduling
              (meeting:=mr.createdMeeting,
               scheduledDate:=01-15-2011);
    assert non-occurrence mrs;
   }
}
```

The verdict of the *new test case* is *Error* and the test processor informs that this is because the *MeetingScheduling* event does not exist in the conceptual schema. In order to fix the error, we add the event type *MeetingScheduling* with its

characteristics (Fig. 47, *a*). For the moment, we need to define neither the effect nor the method.

After this small change in the schema, the verdict of the new test case becomes *Fail* and the test processor informs us that nothing prevents the occurrence of the event *mrs* as expected by the assertion "*assert non-occurrence mrs".* We add the constraint *MeetingScheduling::doesNotScheduleTheMeetingOnAConflictingDate* in order to fix this failure (Fig. 47, *b*). The added constraint describes that the scheduled date should not be a conflicting date, but the knowledge about conflicting dates of meetings is not yet in the schema. Therefore, the verdict of the *new test case* is now *Error*. In order to fix it, we need to add the derived attribute *conflictingDate* and its derivation rule (Fig. 47, *c*). The derivation rule defines that the set of conflicting dates of a meeting is the union of the set of excluded dates of all the participants of the meeting.

The verdict of the test case *invalidMeetingScheduling* becomes *Pass* and no previous test cases cease to be *Pass*. The iteration objective stated by the *new test case* is now achieved. For the moment, no refactoring is felt to be needed and a new iteration can be initiated.

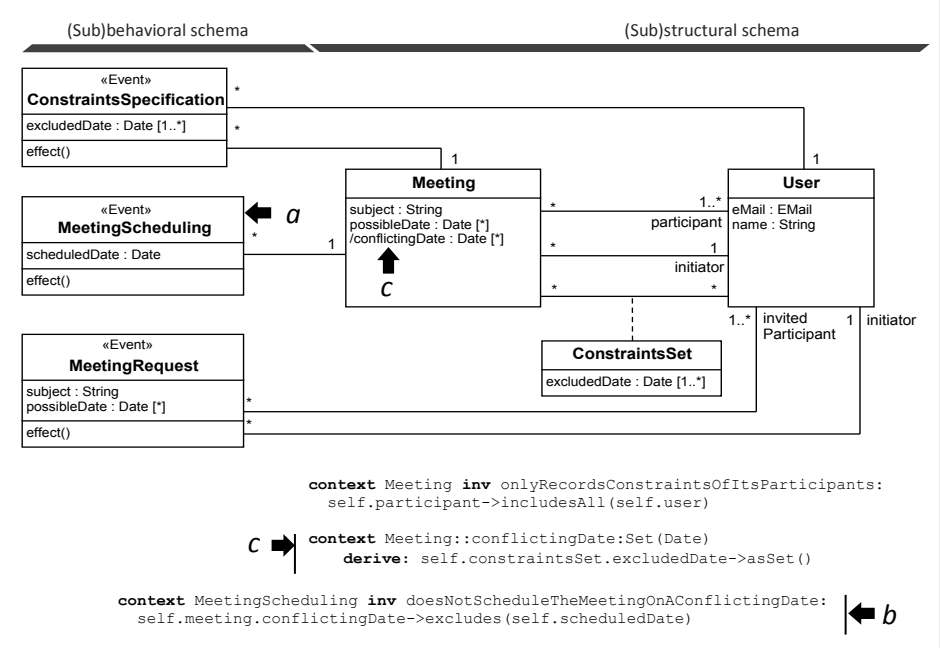


Fig. 47. Schema changes during the first iteration example

8.2.2 Second Iteration Example: Valid Meeting Scheduling

In this iteration, we again apply TDCM to evolve the schema of Fig. 47. We add to the previous test program *MeetingScheduling* the following *new test case,* aimed to ensure that a meeting can be scheduled:

```
test validMeetingScheduling{
   mr := new MeetingRequest(subject:='FuturePlans',
                  initiator:=david,
                  possibleDate:=Set{02-28-2011, 02-29-2011},
                  invitedParticipant:=Set{sarah});
   assert occurrence mr;

   davidConstraints := new ConstraintsSpecification(
                              meeting:=mr.createdMeeting,
                              user:=david,
                              excludedDate:=Set{02-28-2011});
   assert occurrence davidConstraints;

   mrs := new MeetingScheduling
             (meeting:=mr.createdMeeting,
              scheduledDate:=02-29-2011);
   assert occurrence mrs;
}
```

The verdict of the *new test case* is *Error*. The reason is that the occurrence of the event *davidConstraints* reaches an inconsistent state in which the created meeting violates the integrity constraint *Meeting::onlyRecordsConstraintsOfItsParticipants.*

The conceptual modeler realizes that the user *david* is not a participant as it is expected by the story formalized in the *new test case*. We have just detected an inconsistency between two requirements. Therefore, we need to decide whether the initiator of a meeting is a participant by default (as expected in the *new test case)* or not (as expected in the previous test case).

We assume the domain rule that the initiator of a meeting is one of its participants, and therefore we change the effect and the method of the domain event *MeetingRequest* to reflect this:

```
context MeetingRequest::effect()
 post:   (Meeting.allInstances() - Meeting.allInstances()@pre)
         -> one(m | m.subject=self.subject and
                 m.possibleDate=self.possibleDate and
                 m.participant=self.invitedParticipant->including(self.initiator) and
                 m.initiator=self.initiator)
```

```
method MeetingRequest::effect(){
  self.createdMeeting := new Meeting(subject := self.subject,
                                     possibleDate := self.possibleDate,
                                     participant := self.invitedParticipant
                                                    ->including(self.initiator),
                                     initiator := self.initiator);
}
```

The verdict of the *new test case* is now *Error*, because the method and the postcondition of the *MeetingScheduling* event are not specified and consequently, it is not possible to assert its occurrence. At this point, TDCM drives the conceptual modeler to define the effect postcondition and the method of the *MeetingScheduling* event:

```
context  MeetingScheduling::effect()
  post:   self.meeting.scheduledDate = self.scheduledDate

method MeetingScheduling::effect(){
  self.meeting.scheduledDate := self.scheduledDate;
}
```

The verdict of the new test case continues to be *Error* because the entity type *Meeting* does not have the attribute s*cheduledDate*. After adding this attribute, the verdict of the new test case is *Pass*. However, the verdict of the previous test set becomes *Error* unless the multiplicity of the attribute *scheduledDate* is set to 0..1 (there must be meetings without a scheduled date).

Note that TDCM drives the consistency about the behavioral and the structural part of the schema. Test cases assert the occurrence of domain events, and this occurrence cannot success unless the needed structural knowledge is correctly defined. In other words, in conceptual schemas comprising both the structural and the behavioral part, the behavioral knowledge (the functions of the system) is what users require, and in turn, the structural knowledge is necessary for the behavioral knowledge.

After the addition of the attribute *scheduledDate* and its multiplicity (Fig. 48, a), the verdict of the new test case becomes *Pass,* and no collateral effects in previous added knowledge are detected by the automated execution of the *previous test set* (regression testing).

The last task of the iteration is refactoring (if it is applicable). (Correa et al. 2004) presents the refactoring smell “rule exposure” for OCL expressions (a domain rule is specified in the postcondition or the constraint of an event).

The modified postcondition of the event *MeetingRequest* shown above includes the static domain rule “the initiator of a meeting is always one of its participants”. This domain rule should be represented in the structural part of the schema and removed from the postcondition. A possible refactoring consists in making the relationship

type *participant* derived (Fig. 48, b). Its derivation rule specifies that the participants of a meeting are the explicitly invited participants (*invitedParticipant*) and the *initiator* of the meeting.

The verdict of the *current test set* is *Pass* after refactoring and consequently, no changes in the conceptual schema knowledge are detected. The iteration finishes.

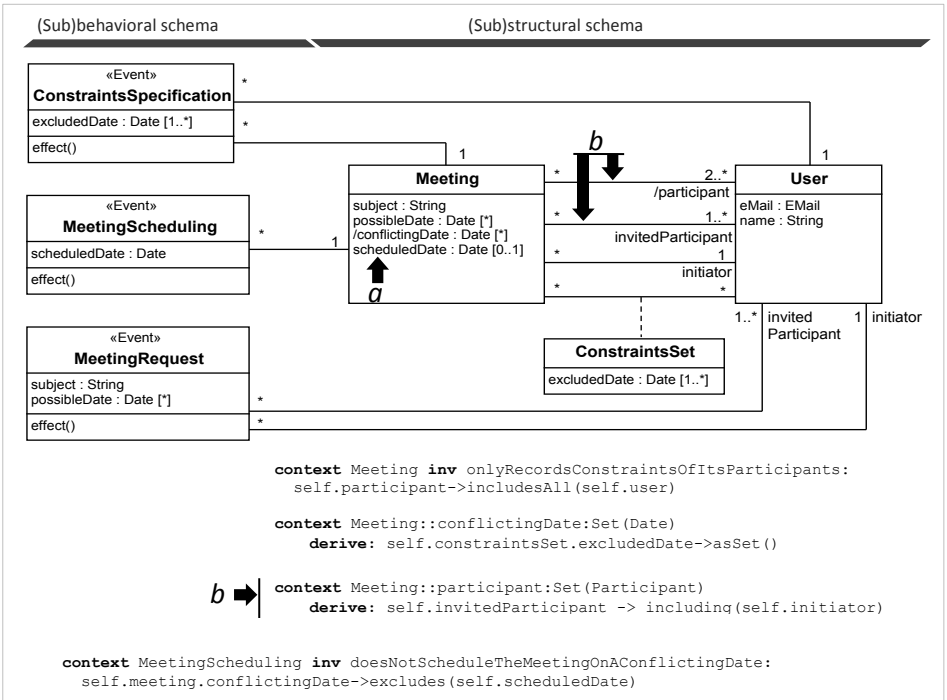


Fig. 48. Schema changes during the second iteration example

8.3 Guidelines

In previous sections of this chapter, we described the main activities that characterize the TDCM method. In this section, we present three guidelines aimed to support the efficient application of the method. Guidelines are well-grounded advices and they are a result of the experience acquired by applying the TDCM method in practice (Section 9).

The iterative nature of TDCM implies that new knowledge is added to the conceptual schema incrementally. The added knowledge may change the set of

valid IB states considered in previous iterations. This situation may invalidate previously processed test cases that need to be fixed, since TDCM always preserves all defined test cases as valid stories. The guidelines defined in Sections 8.3.1 and 8.3.2 are aimed at avoiding the need to fix previous test cases due to this reason.

Moreover, in Section 8.3.3, we present a guideline that uses coverage information (see Chapter 11) in order to enhance confidence in the conceptual modeling activity when applying TDCM. This guideline promotes ensuring that all changes in the schema are validated by executed test cases.

8.3.1 Guideline: Define Constraints as soon as They Are Noticed

When adding a constraint in the schema, the number of consistent IB states decreases. Therefore, IB states in some previous test cases may become inconsistent and those test cases need to be updated in order to preserve consistent stories. This issue can be minimized if constraints are added to the schema as soon as they are noticed.

Every time that in a new test case we define a new entity or relationship type, we should also define all potential constraints involving that type and satisfied by the new test case, even if we are not sure that future test cases will also satisfy them. If, later on, the need arises to remove one constraint, this can be done easily because in most cases the removal will not affect the test cases of the previous test set.

In the initial schema of the example (Fig. 46), we state the constraint that each meeting has at all times at least one participant (multiplicity 1..*). All test cases written since then must specify at least one participant for each meeting. If, later on, we learn that sometimes we do not know yet the participants of a meeting, then we can weaken that constraint (multiplicity *) without impacting previous test cases. In contrast, if the initial multiplicity had been * and later on we need to change it to 1..*, it might be necessary to change previous test cases.

8.3.2 Guideline: Use Default Values When Adding Properties to an Existing Type

New properties (attributes or associations) may need to be added to an entity or domain event type in a new iteration. If the added property is mandatory, previous test cases that use the type of the schema that own this property will become

erroneous (because the property was not specified when the previous test cases were written).

If the added property was not considered in previous test cases, it is probably because it was not relevant for their testing objectives (the value of the property does not affect the assertions of the test case). Changing previous tests due to this reason can be avoided by specifying a default value for the added property.

Consider, as an example, that we start a new TDCM iteration by adding the following test case to the test program *MeetingScheduling* used in Section 8.2.2:

```
test assistantInitiatesAMeeting{
   biancaCreation := new UserCreation(
                              name:='Bianca',
                               eMail:='bianca@bianca.edu');
   assert occurrence biancaCreation;
   bianca := biancaCreation.createdUser;

   mr := new MeetingRequest(
                subject:='Interview',
                initiator:=david,
                urgent:=true,
                possibleDate:=Set{03-15-2011, 04-16-2011},
                participant:=Set{david, sarah, bianca});
   assert occurrence mr;

   sarahConstraints := new ConstraintsSpecification(
                               meeting:=mr.createdMeeting,
                               user:=sarah,
                               excludedDate:=Set{03-15-2011});
   assert occurrence sarahConstraints;

   mrs := new MeetingScheduling(
                meeting:=mr.createdMeeting,
                scheduledDate:=03-15-2011);
   assert occurrence mrs;
}
```

This test case specifies a story in which a meeting is expected to be scheduled even if the meeting is considered to be urgent. The test case specifies that the user *david* requests a meeting and indicates that the users *david, sarah* and *bianca* are expected to be participants of the meeting. Although the user *sarah* specifies a date constraint that conflicts with the date on which *david* schedules the meeting, it is expected that the meeting will be successfully scheduled due to its urgency.

The verdict of the new test case is *Error* because the knowledge about the urgency of meetings is not in the schema. We need to add the attribute *MeetingRequest::urgent:Boolean[1]*. We also need to add the attribute

Meeting::urgent:Boolean[1] and change the postcondition of the event *MeetingRequest* to indicate that, when the meeting is created, the value of this attribute is set according to the value of *MeetingRequest::urgent*.

After these changes, the verdict becomes *Fail*. The reason is that the event *mrs::MeetingRequestScheduling* cannot occur (as expected), because the initial integrity constraint (which was previously defined in the conceptual schema) *MeetingScheduling::doesNotScheduleTheMeetingOnAConflictingDate* is not satisfied. We change this constraint in order to allow urgent meetings to be scheduled on conflicting dates:

```
context MeetingScheduling inv doesNotScheduleTheMeetingOnAConflictingDate:
  not(self.meeting.urgent) implies
    self.meeting.conflictingDate->excludes(self.scheduledDate)
```

The verdict of the current test case becomes *Pass*. Nevertheless, the verdicts of the previous test cases (see Section 8.2) are *Error*. The reason is that the attribute *MeetingRequest::urgent:Boolean[1]* is mandatory and it is not specified in previous test cases when meetings are requested. In order to achieve the verdict *Pass* for the test set, we could manually update all previous test cases to specify a value for the new mandatory attribute. However, according to the proposed guideline, this task can be avoided by setting its default value to *false*.

8.3.3 Guideline: Maintain High Basic Coverage Satisfaction at Each Iteration

In the programming field, Test-Driven Development includes the principle that “no code goes into production unless it has associated tests”. According to (Astels 2003), by writing only the code required to pass the current test, the developer puts a limit in each iteration in the written code. Moreover, it fosters that code is written in small increments according to the executed test cases. The key rationale of this principle is that the property that all code is tested as soon as it is written gives more confidence during the development. In TDD, this way of working is usually promoted by asking developers to “do the simplest thing that could possibly work”.

In Test-Driven Conceptual Modeling (TDCM), this TDD principle used in programming is not included as a *must-be* because (1) we allow initial conceptual schemas with elements that have not been tested prior to the TDCM application, and (2) we allow the addition of non-mandatory schema elements even if they are not driven by the current test case. This is possible when the conceptual modeler includes knowledge that he/she believes that it will be relevant according to his/her own knowledge about the domain, even if the test set execution has not revealed its relevance yet.

As we explain in Chapter 11, the relevance of all the elements of the schema is proved only if the test set covers all the elements according to a set of adequacy criteria. Since TDCM pursues confidence on the semantic quality (Section 2.2) of each (partial) schema being developed at each iteration and fosters the incremental development in “small” iterations, a high degree of basic coverage satisfaction (Section 11.1) is desired at the end of each TDCM iteration. In other words, TDCM allows the conceptual modeler to specify knowledge in the schema, although it is not required to pass a test. However, this knowledge (the correctness and relevance of which is not determined by any test case) need to be confirmed by stakeholders and tested promptly.

Basic coverage satisfaction indicates the degree in which the elements specified in the schema have been tested by the processed test cases. Therefore, it is convenient to check the basic coverage satisfaction at the end of each iteration. If there are uncovered elements, then the following actions are suggested: (1) confirm with stakeholders the conceptual modeler conjectures, and if confirmed, (2) formalize them as test objectives (in new test cases or extensions of existing test cases) and (3) prioritize the execution of the new test objectives to be processed by TDCM as soon as possible.

By following these actions, high degrees of basic coverage are maintained during TDCM application and, consequently, high degrees of confidence on conceptual schema quality are also maintained.

Consider as an example the test case *assistantInitiatesAMeeting* used in Section 8.3.2. Imagine that the conceptual modeler adds the non-mandatory attributes *MeetingRequest::expectedDuration [0..1]* and *Meeting::expectedDuration [0..1].* The specification of the event *MeetingRequest* is also updated to set up these attributes. The conceptual modeler adds this knowledge because, according to his/her experience, he/she believes that it is probably that stakeholders may need to specify the expected duration of a meeting, although the relevance of these properties has not been revealed yet by any processed test case. According to the guideline defined in this section, the basic coverage should be checked before the end of the iteration. When checking the coverage, we realize that the new attributes are not covered by the processed test cases and therefore, they are conjectures of the conceptual modeler whose relevance has not been confirmed by any test case. The conceptual modeler should (1) confirm with stakeholders that this feature is expected to be part of the system being modeled, (2) define new test cases or extending the existing ones to include stories that consider the expected duration of meetings, and (3) prioritize the new stories to be processed as soon as possible in order to maintain basic coverage.

8.4 Conceptual Schema Refactoring

In the context of software programming, refactoring "is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure" (Fowler et al. 1999).

In the last decade, several catalogs of refactoring operations for modifying source code have been proposed. However, more recently, several researchers have attempted to use refactoring operations at higher degrees of abstraction such as database models or conceptual schemas.

In general, refactoring implies "rewritten material to improve its readability or structure, with the explicit purpose of keeping its meaning or behavior" (Wikipedia's definition). This definition can be adapted to conceptual schema refactoring as follows: *Refactoring is the process of restructuring the conceptual schema specification to improve its external quality (understandability, structure, etc.) by preserving the specified knowledge.*

The refactoring process includes two main activities: 1) identifying refactoring opportunities (also called *refactoring smells*) and 2) applying *information-preserving* schema transformations (also called *refactoring operations*). Some of these activities may be automatically applied while others are based on metrics or require the manual intervention of the modeler.

Information-preserving transformations are schema transformations that preserve the knowledge defined in the schema. Identifying whether two conceptual schema specifications define the same knowledge implies analyzing the equivalence between the input schema and the output schema involved in the refactoring process. According to (Halpin 2001), "two conceptual schemas are equivalent if and only if whatever universe of discourse, state or transition can be modeled in one can also be modeled in the other".

TDCM fosters refactoring of the conceptual schema under development at the end of each iteration. Moreover, TDCM contributes to ensure the knowledge preservation during the refactoring process and helps identifying non information-preserving changes promptly (see Section 8.1.1). By this way, TDCM positively influences the pragmatic quality of the schema (see Section 2.2.2).

In this section, we briefly review relevant conceptual schema refactoring proposals (Section 8.4.1) which may be applied when using TDCM. Some of these refactoring proposals are based on programming refactoring catalogs, while others are specific to conceptual modeling. We also present a specific example of a refactoring process in the context of TDCM (Section 8.4.2).

8.4.1 Refactoring Catalogs

The range of tools that support the refactoring of software code has increased in the last decades. These tools are useful and work fine for small granularity refactoring operations which can be applied to code. However, some refactoring operations require more structured knowledge representations. This is the reason why some authors have proposed refactoring techniques on higher levels of abstraction. We briefly summarize some relevant proposals in the following.

(Sunyé et al. 2001) defined a refactoring catalog for UML class diagrams and statecharts. They include refactoring operations such as *Insert generalization element*, *Delete generalization element, Move method, Generalize element* and *SpecializeElement.* However, only those operations related to state machines are formally specified.

(Zhang et al. 2005) presented a framework for automatically executing refactoring operations on models. This framework allows to execute a subset of the operations defined by (Fowler 1999): *Add class, Extract superclass, Extract class, Remove class, Move class, Rename class, Collapse hierarchy, Add attribute, Remove attribute, Rename attribute, Pull up attribute* and *Push down attribute.*

(Judson et al. 2003) proposed a set of model transformation operations which may be systematized by applying patterns. Similarly, (Porres 2003) provides a set of transformation rules for conceptual schema refactoring.

(Van Gorp et al. 2003) presented a set of refactoring operations which can be formally specified by using *Action Semantics* (Sunyé et al. 2002). Refactorings are defined by operations composed of a precondition, a postcondition and a bad smell condition which is specified as an OCL operation. This approach can detect some refactoring opportunities for conceptu()al schemas.

(Correa et al. 2004) worked on the application of refactoring operations to integrity constraints written in OCL, by adapting some classical software refactoring operations (Fowler et al. 1999) to OCL. They also provide a set of OCL smells (rules to detect refactoring opportunities) for these operations. The OCL smells are: *Magic literal* (it detects when an integrity constraint uses a literal within the constraint source), *And chain* (it detects when a constraint consists of two or more subconstraints linked by the operator *and*), *Long journey* (it detects when an OCL expression traverses a large number of associations), *Rules exposure* (it detects when the business rules are specified in the postconditions or preconditions of system operations) and *Duplicated code* (it detects when OCL expressions are duplicated throughout the conceptual schema).

Each of these works is focused on some parts of the conceptual schema. The refactoring opportunities and the refactoring operations can be applied when performing TDCM. However, an integrated catalog of refactoring operations applicable to conceptual schemas is desired. A first attempt to present an exhaustive catalog, based on the current state of the art on software refactoring and conceptual schema refactoring is presented by (Conesa et al. 2011). This catalog is organized in a taxonomy of refactoring operations (Fig. 49). The taxonomy includes two main categories: (1) structural refactoring operations, aimed at refactoring the structural elements of the conceptual schema and (2) integrity constraint refactoring operations, aimed at refactoring behavioral elements and integrity constraints. As a guide to apply refactoring operations when using TDCM, we briefly review those operations proposed in this taxonomy in the following.

Moving Features between Concepts

Move Integrity Constraint: This operation moves an integrity constraint from one entity type to another. This is an adaptation of Fowler's *Move method* operation.

Change relationship context: This operation changes the type of one of the participants in a given relationship type. The old and new participants in the relationship type cannot be related with a generalization/specialization relationship.

Extract entity type: This operation splits an entity type in two and distributes its content between the two new entity types.

Inline entity type: This is the inverse of the operation *Extract entity type*. It merges two entity types into one new entity type.

Extract datatype: This operation changes an entity type into a datatype. In conceptual schemas that incorporate the concept of datatypes, such as in UML, this operation converts a class in a datatype and changes all the relationship types that dealt with the class to attributes.

Organizing Data

Delete PartOf constraint: This operation deletes a *PartOf* integrity constraint related to a given binary relationship type. In the particular case of conceptual schemas defined in UML, this operation replaces an aggregation with an association.

Add PartOf constraint: This is the inverse of the operation *Delete PartOf constraint*.

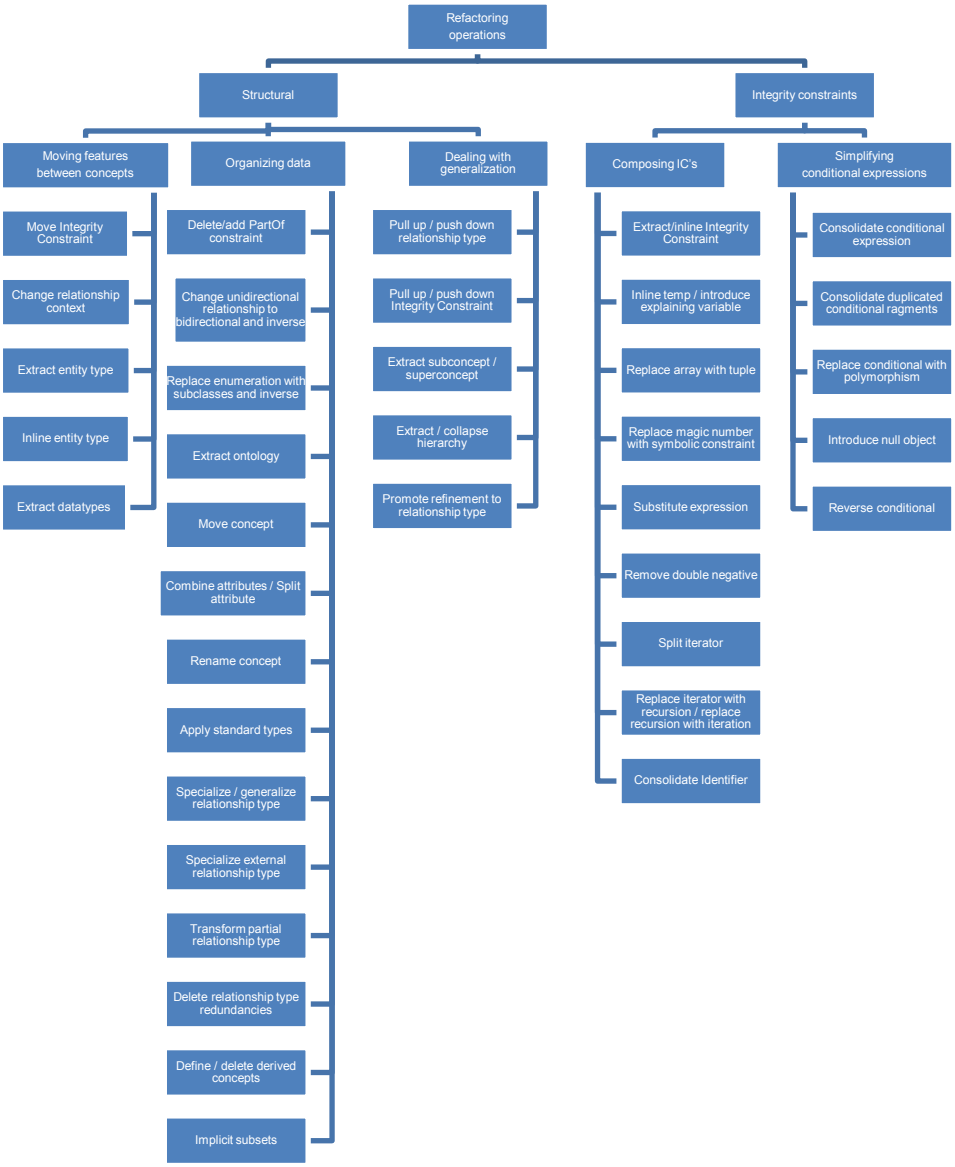


Fig. 49. Catalog of conceptual schema refactoring operations (Conesa et al. 2011)

Change unidirectional relationship type to bidirectional: This operation changes the navigability of a relationship type from unidirectional to bidirectional.

Change bidirectional association to unidirectional: it is the inverse of the operation *Change unidirectional association to bidirectional*.

Replace enumeration with subclasses: This operation replaces an enumeration attribute used to determine the type of the instances of a given entity type *E* with a set of subentity types of *E*. Each of the entity types created represents one of the possible values of the deleted attribute.

Replace subclass with an enumerated type: This is the inverse of the previous operation. This operation replaces a set of subtypes of a common entity type *E* with a relationship type in *E* that represents the same information. An integrity constraint is created to restrict the number of possible values of the new relationship type; this number is equal to the number of deleted subtypes.

Extract ontology: Some conceptual modeling (ontology) languages allow the knowledge to be grouped in small blocks based on its meaning, context or domain. For example, the UML language allows grouping conceptual schema fragments as packages.

Move concept: This operation moves a concept that belongs to one conceptual schema to another conceptual schema.

Combine attributes: This operation merges several attributes into one. This operation is very useful when a modeler creates attributes too specific for the objectives of the conceptual schema.

Split attribute: This is the inverse of the operation *Combine attributes*. It splits 1 attribute into *n*, with $n>1$. The types of the new attributes should be the same as the original or any of its subtypes.

Rename concept: This operation changes the name of a concept (an entity or relationship type).

Apply standard types: Sometimes, different attributes that represent the same (or similar) concepts have different types. For example, a phone number may be represented by a *String* and a cell phone number by an *Integer*. Obviously, the quality of the conceptual schema is improved if all phone numbers are represented in the same way.

Relationship type specialization: This operation change one n-ary relationship type for *m* (n-1)-ary more specific relationship types. This operation is particularly useful when one participant in a relationship type can only have a predefined set of values known prior to the conceptualization phase. In such a case, the n-ary relationship may be replaced with *m* relationship types, where *m* is the number of possible values of the participant.

Relationship type generalization: This is the inverse of the previous operation (*Relationship type specialization*). It replaces *m* n-ary relationship types with similar semantics with one relationship type with an arity of n+1. The new participant in the relationship type is used to identify the semantics (previous relationship type) used in each instance of the relationship type.

External relationship specialization: This operation makes more specific a given relationship type based on the values of another relationship type. To do this, the first relationship type absorbs the second one.

Transforming partial relationship type: This operation changes a partial relationship type (with a minimum cardinality of 0) into a total one (with a minimum cardinality of 1). To do so, a new subtype that contains the instances that used to participate in the partial relationship type is created. Thereafter, the relationship type is redefined by pointing the new created subtype and changing the minimal cardinality to one.

Transforming partial relationship types that are total in union: Sometimes, an integrity constraint can represent that the union of two partial relationship types is total, which means that for each instance of a partial participant there is at least one instance of these relationship types that relates it. In this case, this operation makes sense and replaces the two partial relationship types with one total relationship type. It also creates a new entity type defined as the union of the entity types that participated in the previous relationship types. This refactoring operation also deletes the integrity constraint that indicated the totality of the union of the two relationship types.

Deleting relationship types redundancy: This operation deletes redundant relationship types. To be redundant, two relationship types must share both the same participants and semantics. Therefore, this operation always requires designer intervention to determine whether the *n* relationship types, which are supposedly redundant, have the same semantics.

Defining derived concepts: This operation defines a new entity type or relationship type whose population is derived from the information base of the conceptual schema.

Deleting derived concepts: This operation deletes a derived concept (entity type or relationship type). This is possible when a derived concept is redundant in a conceptual schema. If it is not relevant, then its elimination does not imply any loss of semantics.

Implicit subsets: This operation deletes redundant generalization paths.

Dealing with Generalization

Pull up relationship type: This operation replaces one participant in a relationship type with its supertype.

Push down relationship type: This operation replaces one participant in a relationship type with the subtypes of that participant. It is the inverse of the operation *Generalize relationship*.

Pull up IC: This operation replaces the context of a relationship type with the supertype of that context.

Push down IC: This operation replaces the context of a relationship type with the subtype of that context. This operation is the inverse of the integrity constraint *Pull up IC*.

Extract subconcept: This operation creates a new concept as a subtype of an existing one and moves some of the existing concept's properties to the new one.

Extract superconcept: The operation creates a new concept as a supertype of an existing one and moves some of the existing concept's properties to the new one.

Collapse hierarchy: This operation collapses one concept (entity type or relationship type) with its subtypes. The new concept contains the semantics of the two collapsed elements.

Extract hierarchy: This operation splits a concept into several concepts related by generalization relationships and spreads the properties of the original concept to the new concepts. It is the inverse of the operation *Collapse hierarchy*.

Promote refinement to relationship type: The relationship types defined in a conceptual schema tend to be used, with some restrictions, by the subtypes of the entity types where they are defined. In such cases, a relationship type may be redefined using either a predefined construction of the ontology or a general integrity constraint. When a relationship type is only used by the elements where it is redefined or their subtypes, then it and its redefinition can be replaced with another relationship type that uses as participants the entity types where the relationship type was redefined and takes into account the integrity constraints added in the redefinition.

Composing Integrity Constraints

Extract IC: This operation splits one integrity constraint in two and spreads its code to the two created constraints.

Inline temp: This operation replaces a temporal variable used in an integrity constraint with the expression that defines its value. This operation is particularly useful when the temporal variable is used infrequently.

Introduce explaining variable: This operation assigns a variable the result of an expression used in an integrity constraint and replaces the expression with a reference to the new variable wherever it occurs. This operation improves maintainability when the same expression is used several times in an integrity constraint.

Replace array with tuple: This operation replaces an array used in the body of an integrity constraint with a tuple. The new tuple will have one field for each row of the array. We included this operation in our catalog because some of the languages for representing general integrity constraints may support tuples and arrays.

Replace magic number with symbolic constant: This operation replaces a number used in an integrity constraint with a constant of the same value.

Substitute expression: This operation replaces an expression of an integrity constraint with another expression with the same meaning.

Remove double negative: This operation replaces a double negation with an affirmation, for example replacing *if(Not NotFound)* for *if(Found)*.

Split iterator: This operation separates one iteration in *n* (*n*>1), where each of the *n* new iterators performs a different activity. We decided to include in the catalog some operations that deal with iterators because they are often used in the languages that represent general integrity constraints.

Replace iteration with recursion: This operation replaces an iteration structure with its equivalent recursive call.

Replace recursion with iteration: This operation replaces a recursive call structure with its equivalent iteration structure.

Consolidate identifier: This operation can be applied when two or more expressions use different identifiers to refer to the same concept. It forces all expressions to access the instances of a concept in the same way, which means using the same identifier.

Simplifying Conditional Expressions

Consolidate conditional expression: This operation combines *n* conditionals defined within an integrity constraint in a single conditional.

Consolidate duplicate conditional fragments: This operation extracts the parts of a conditional structure that are repeated in all of its branches.

Replace conditional with polymorphism: This operation distributes the conditional through the subtypes of the context entity type of the integrity constraint where it is defined. Thus, for each subtype, the conditions that its instances should satisfy are defined.

Introduce null object: This operation creates a new subtype of a given entity type *E*. This subtype represents the instances of *E* with undefined information, that is, the instances that have no value for a given relationship type.

Reverse conditional: This operation modifies a conditional to make it more comprehensible. It negates the entire conditional and, therefore, the condition and the *then* and *else* branches.

8.4.2 Refactoring Example

The TDCM example iteration described in Section 8.2.2, includes a refactoring example that uses the refactoring smell *Rules Exposure* (see Section 8.4.1). In that case, a business rule specified in the postcondition of an event specification is extracted as an integrity constraint.

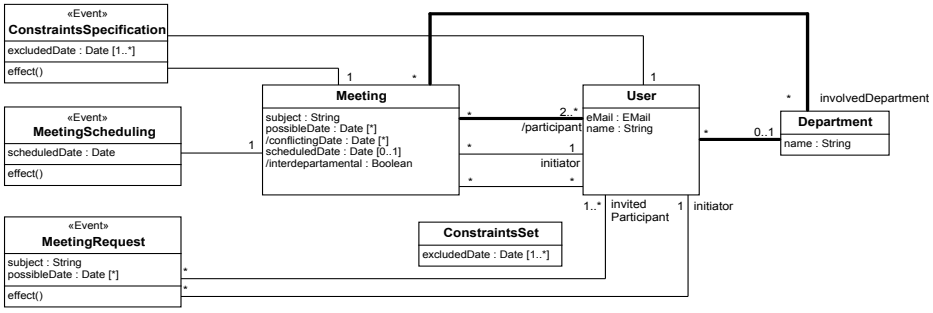
In this section, we explain an additional refactoring example by operations included in the refactoring catalog explained in the previous section.

Consider the following test case. The test objective is ensuring (1) that users of the meeting scheduler system can be assigned to departments, and (2) that interdepartmental meetings can be automatically detected, and (3) that the system knows the departments involved in each meeting.

```
test meetingAndDepartments{

   marketingDptCreation := new DepartmentCreation
                                   (name:='Marketing');
   marketingDpt := marketingDptCreation.createdDepartment;
```

```
    financeDptCreation := new DepartmentCreation
                                    (name:='Finance');
    financeDpt := financeDptCreation.createdDepartment;


    johnCreation := new UserCreation(
                                name:='John',
                                eMail:='john@john.edu'
                                department:=marketingDpt);
    assert occurrence johnCreation;
    john := johnCreation.createdUser;


    maryCreation := new UserCreation(
                                name:='Mary',
                                eMail:='mary@mary.edu'
                                department:=financeDpt);
    assert occurrence maryCreation;
    mary := maryCreation.createdUser;


    mr := new MeetingRequest(
                  subject:='Ideas',
                  initiator:=john,
                  urgent:=true,
                  possibleDate:=Set{07-15-2011, 07-16-2011},
                  participant:=Set{mary},
                  involvedDepartment:=
                                   Set{financeDpt,marketingDpt);
    assert occurrence mr;
    ideasMeeting := mr.createdMeeting;


    assert true ideasMeeting.interdepartamental;
    assert equals ideasMeeting.involvedDepartment
                     Set{financeDpt,marketingDpt);


    mrs := new MeetingScheduling(
                  meeting:=mr.createdMeeting,
                  scheduledDate:=03-15-2011);
    assert occurrence mrs;
}
```

This test case has driven the evolution of the conceptual schema obtained in Section 8.2.2 to the conceptual schema shown in Fig. 50. Assume that the following changes to the schema have been also performed (although they are not graphically represented in Fig. Fig. 50): (1) the *UserCreation* event type now admits specifying the department of a user, and (2) a new event type *DepartmentCreation* has been defined in order to create valid instances of the *Department* entity type.




```
context Meeting inv onlyRecordsConstraintsOfItsParticipants:
  self.participant->includesAll(self.user)

context Meeting::conflictingDate:Set(Date)
    derive: self.constraintsSet.excludedDate->asSet()

context Meeting::interdepartamental:Boolean
   derive: self.involvedDepartment.size()>1

context Meeting::participant:Set(Participant)
    derive: self.invitedParticipant -> including(self.initiator)

context Meeting inv hasOnlyValidDepartments:
  self.involvedDepartment = self.participant.department

context MeetingScheduling inv doesNotScheduleTheMeetingOnAConflictingDate:
  self.meeting.conflictingDate->excludes(self.scheduledDate)
```

Fig. 50. Meeting scheduler conceptual schema (evolution from Fig. 48)

The verdict of the test set (which includes the current test case and the previous test cases) is *Pass*. However, a refactoring opportunity is smelled because of the highlighted cycle in the conceptual schema specification.

According to the refactoring operation *Deleting relationship types redundancy* explained in Section 8.4.1, if the modeler confirms that one of the paths in the cycle is redundant, the corresponding relationship type may be removed from the schema. The reason is that the knowledge it represents, may be navigated through other relationship types. In the conceptual schema shown in Fig. 50, the integrity constraint *Meeting::hasOnlyValidDepartments* confirms that the knowledge about the departments involved in a meeting may be obtained through the navigation *self.participant.department*. Therefore, the relationship type may be removed by applying the mentioned refactoring operation.

After the application of a refactoring operation, TDCM encourages to execute the test set in order to check that refactoring does not produce errors or failures. If we execute the test set on the schema of Fig. 50, without the relationship type *meeting-involvedMeeting*, the verdict becomes *Error*. The reason is that the event *mr::MeetingRequest* executed in the test case explicitly populates this relationship

type. Therefore, by applying TDCM we detect that the refactoring process is not finished and we need to modify the event *MeetingRequest* in order to delete the explicit instantiation of this relationship type. Note also that we need to modify the *mr:MeetingRequest* event assertion as follows:

```
mr := new MeetingRequest(
            subject:='Ideas',
            initiator:=john,
            urgent:=true,
            possibleDate:=Set{07-15-2011, 07-16-2011},
            participant:=Set{mary},
            involvedDepartment:=
                        Set{financeDpt,marketingDpt);
assert occurrence mr;
```

After these changes, the verdict remains *Error*, because the expression "assert equals ideasMeeting.involvedDepartment Set{financeDpt,marketingDpt);" is now incorrect. Then, since it is relevant to know the involved departments of a meeting, we can apply the refactoring operation *Defining derived concepts* in order to include the relationship type *meeting-involvedDepartment* as a derived relationship type. Its derivation rule is:

```
context Meeting::involvedDepartment:Set(Department)
  derive: self.involvedDepartment = self.participant.department
```

After the addition of the derived relationship type, the verdict of the test set becomes *Pass*. It indicates that the test set that passed before refactoring continues to pass after the refactoring process.

Finally, we could also note the smell that both the derived rule *Meeting::involvedDepartment* and the integrity constraint *Meeting::hasOnlyValidDepartments* have exactly the same OCL expression. The reason is that since the relationship type *Meeting-involvedDepartment* is now derived, the integrity constraint is always satisfied. Then, the integrity constraint can be removed. If we execute the test set again, the global verdict is *Pass* again, and a new TDCM iteration could be started.

8.5 Conjectured TDCM Advantages and Drawbacks

Our approach for test-driven development of conceptual schemas is based on the main TDD principles applied to software programming, and it has the potential of exporting its benefits to conceptual modeling.

Test-Driven Development (TDD)	Test-Driven Conceptual Modeling (TDCM)
▲ **Code quality:** Evolving a set of test cases and making them *Pass* increases code quality (code is added by fixing errors). (Janzen et al. 2005, Beck 2003, Astels 2003, Koskela 2007, Maximilien et al. 2003, Janzen et al. 2008, George et al. 2003)	▲ **Conceptual schema quality:** Evolving a set of test cases and making them *Pass* increases the semantic quality (correctness and completeness) of the conceptual schema.
▲ **Focused objectives:** TDD cycle promotes to state and to achieve focused objectives instead of worrying about the growing program at any time. Solving small problems, validating them and moving forward is a more predictable way to develop. (Janzen et al. 2005, Beck 2003, Astels 2003, Koskela 2007, Janzen et al. 2008, Stephens et al. 2003, Stephens et al. 2010)	▲ **Focused objectives:** TDCM cycle promotes to evolve the conceptual schema incrementally according to focused objectives. Adding knowledge to the schema according to new expectations, validating them and moving forward is a more predictable way to perform conceptual modeling.
▲ **Keeping code healthy with refactoring:** Refactoring improves the design after adding new code. (Janzen et al. 2005, Beck 2003, Astels 2003, Koskela 2007, Janzen et al. 2008, Stephens et al. 2003, Stephens et al. 2010)	▲ **Keeping conceptual schema specification healthy:** Refactoring improves the way in which the conceptual schema is modeled.
▲ **Making sure that previous code still works:** Automated regression tests help making sure that code that worked still works after adding new code. (Janzen et al. 2005, Beck 2003, Astels 2003, Koskela 2007, Janzen et al. 2008, Stephens et al. 2003, Stephens et al. 2010)	▲ **Making sure that already defined knowledge is still valid:** After adding new knowledge to the schema, regression testing helps detecting undesired collateral effects on previously defined knowledge.
▲ **Getting continuous feedback:** Code is driven by the feedback provided in test errors and failure information. Therefore, code is evolved as a reaction to continuous feedback. (Janzen et al. 2005, Beck 2003, Astels 2003, Koskela 2007, Janzen et al. 2008, Stephens et al. 2003, Stephens et al. 2010)	▲ **Getting continuous feedback:** Conceptual modeling is driven by the feedback provided in test errors and failure information. Therefore, new knowledge is added as a reaction to continuous feedback.
▲ **Workable software at each iteration:** After each iteration, you have code that works according to a set of tests. (Janzen et al. 2005, Beck 2003, Astels 2003, Koskela 2007, Janzen et al. 2008)	▲ **Complete and correct schema at each iteration:** After each iteration, the result is a correct and complete conceptual schema according to the processed test cases.
▲ **Programmer confidence:** Code is testable as soon as it is written. Together with the previous properties, it increases programmers' confidence. (Janzen et al. 2005, Beck 2003, Astels 2003, Koskela 2007, Janzen et al. 2008)	▲ **Conceptual modeler confidence:** The conceptual schema is continuously validated by the execution of conceptual test cases. Together with the previous properties, it increases conceptual modelers' confidence.
▼**Stakeholders' availability:** Stakeholders may not be available during the implementation phase. (Stephens et al. 2003)	☐ **Stakeholders' availability:** The availability of stakeholders is inherent to conceptual modeling (as a requirements engineering activity).
▼**Tests as specifications:** TDD is "code-centric" and lacks of focus on design and analysis. **(Stephens et al. 2003, Stephens et al. 2010, Boehm 2002)**	☐ **Development of the conceptual schema:** The aim of TDCM is defining the conceptual schema of an IS, which is a requirements artifact.
▼ **Writing test cases is time consuming:** Writing test cases consumes time although testing is essential before or after coding. (Maximilien et al. 2003, George et al. 2003)	▼ **Writing test cases is time consuming:** Writing test cases for testing the conceptual schema consumes time (during or after developing the schema) although conceptual schema testing increases its quality.
▼ **Rewriting tests:** When new code is added, previous tests may need to be rewritten. (Stephens et al. 2003)	▼ **Rewriting tests:** Adding new knowledge to the CSUT may cause rewriting previous passing test cases. Some guidelines are proposed in Section 4.4. to minimize this drawback.
▼ **Tests origin:** TDD assumes that developers write tests directly from stakeholders, but does not consider previous analysis or design artifacts. (Stephens et al. 2003, Stephens et al. 2010)	☐ **TDCM is applied in the context of a general method:** Test cases may be derived from artifacts provided by the general method that embeds TDCM.
▲ *Advantage* ▼ *Drawback* ☐ *Non-applicable*	

Table 4. TDD advantages and drawbacks and its potential application to TDCM

Several researches have conducted experiments on the effectiveness of TDD (Section 7.4). The first column of Table 4 summarizes seven advantages and five drawbacks indicated by these researches and other published literature. The second column of the table shows our conjectures of the extent to which such advantages and drawbacks may apply to TDCM. Our conjectures are based on our own experience and the case studies reported in Chapter 9. The table shows that TDCM may "inherit" the seven advantages of TDD and two of its drawbacks, and also that three drawbacks of TDD are not applicable to TDCM.

9

Case Studies on Test-Driven Conceptual Modeling

Case study research is an observational experimentation method in which projects, methods or activities are monitored in a specific application case (Wohlin 2000). In this chapter, we report four different experimental case studies aimed to analyze the application of the Test-Driven Conceptual Modeling (TDCM) method presented in Chapter 8. In these case studies, the application of TDCM is supported by the conceptual schema testing approach proposed in Chapter 4, and assisted by the *CSTL Processor* tool described in Chapter 5.

The structure of the chapter is as follows. Section 9.1 introduces the four case studies. Sections 9.2, 9.3 and 9.4 report the main results of the TDCM application in each case. Finally, Section 9.5 discusses the lessons learned.

9.1 Case Studies Overview

Design Science Research, the general research framework of this Thesis (Section 1.4), fosters experimentation and refinement by use of new developed approaches. This framework states that *"knowledge and understanding of a problem domain and its solution are achieved in the building and application of the designed artifact"* (Hevner et al. 2004).

In this chapter, we report the application of TDCM in the following experimental case studies:

- The development of the conceptual schema of the bowling game system, a well-known exemplar used for illustrating *eXtreme Programming* practices.
- The reverse engineering definition of the schema of the *osTicket* support system, a real-world and widely-used open source system for managing customer support cases.
- The development of the conceptual schema of two systems performed by groups of master students in two projects developed in consecutive editions of a requirements engineering course: a flexible reservation system for events, and a localization-based system for care-control in old people's homes.

The overall goals of these case studies are: (1) Analyzing the viability of developing conceptual schemas by using TDCM and evaluating its effectiveness, (2) Characterizing the errors and failures which drive the development and the most common actions aimed to fix them, (3) identifying patterns that characterize TDCM iterations, and (4) using the lessons learned to improve and refine the method and the testing environment.

These case studies are representative of different kinds of information systems and they correspond to different development situations. In case studies, data is collected for a specific purpose throughout the study (Wohlin 2000). The *CSTL Processor* was adapted in order to automatically collect information about the TDCM application. We obtained information such as the errors and failures revealed, the time spent to complete each iteration, and the evolution of the conceptual schema under development.

In the following, we briefly present, for each case study, the universe of discourse, the objectives to be achieved, the testing/development strategy and the application context.

9.1.1 A Bowling Game System

Bowling is a sport in which players attempt to score points by rolling a bowling ball along a flat surface in order to knock down as many pins as possible.

The bowling game system is a popular case study used to demonstrate *eXtreme Programming* (XP) practices in action. Robert C. Martin popularized this case study in the *Agile Software Develoment* book (Martin et al. 2008).

The bowling game has two main events: the creation of a new game and the action of a throw. The main aspects of the system are about the computation of the game score, taking into account the bowling game rules and the different kinds of throws (regular, spare or strike).

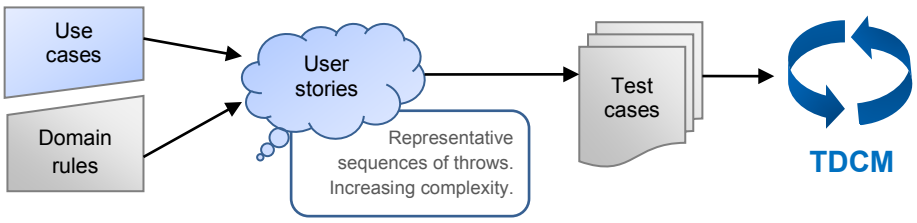


Fig. 51. Testing/development strategy (Bowling Game Case Study)

The objective of the TDCM application in this case was the development of the conceptual schema of a bowling game system from the description of the informal requirements found in the case study formulation (Martin et al. 2008).The case study was conducted by a conceptual modeler familiar with the testing environment.

The development strategy (Fig. 51) includes the definition of a representative set of game occurrences (which are formalized as test cases) that cover different types of throws. Test cases were processed according to a previously defined order based on the expected growing complexity of stories.

9.1.2 The osTicket Support System

Ticket support systems allow customers to create and keep track of support requests as tickets and allow staff members to organize, manage and respond them. A ticket contains all the information related to a customer support request.

In particular, *osTicket* is an open source support ticket system which *"is designed to improve customer support efficiency by providing staff with tools they need to deliver fast, effective and measurable support"* (osTicket 2011). The *osTicket* system allows users to create new tickets online or by email. It also allows the staff to create tickets in behalf of customers. Moreover, the system allows the staff members to add internal notes to tickets. Configurable help topics, assignment of staff responsibility for each ticket, due dates, departments, priorities, etc. are also offered. Customers are also allowed to access the system in order to keep track of the status of their tickets and reply them.

osTicket is an existing system. Therefore, the objective of the TDCM application in this case study was the reverse engineering development of its conceptual schema. Chikofsky and Cross (Chikofsky et al. 1990) state that "Reverse engineering is the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction. The purpose of reverse engineering is to understand a software system in order to facilitate enhancement, correction, documentation and redesign". The results and the observation of this study are the same defined for the previous case study (Section 9.1.1), but in the development of a real-sized and widely used system.

A conceptual modeler familiar with the *osTicket* system and the testing environment conducted the TDCM application in this case study.

Fig. 52 shows a schema of the testing strategy that was used for the development of the conceptual schema of the *osTicket* system. The development strategy included the partition of the system knowledge in two areas:

- *Basics and configuration of the system.*
- *Tickets creation and management.*

The main reason for partitioning the knowledge is that the use case objective was to develop the structural schema for *Basics and Configuration* (excluding event types), and both the structural and the behavioral schema of the *Tickets creation and management*. Moreover, *Tickets creation and management* knowledge was expected to depend on the *Basics and Configuration* of the system.

The definition of a representative set of expected stories (which were formalized as test cases) for each knowledge area was performed before applying TDCM, although new stories were defined and processed during the TDCM application. The source artifacts for the stories depended on the area of knowledge:

- *Basics and configuration of the system*: The source artifacts were (1) An informal list (written in natural language) of requirements related to the basics and configuration of the system, elicited from the public documentation of the *osTicket* system and by using it; and (2) An estimated dependency graph between the informal requirements.
- *Tickets creation and management*: The source artifacts were (1) The specification of the use cases that informally describe the behavior of the system; (2) The specification of the main domain rules of the system; and (3) An estimated dependency graph between the use cases and the domain rules.

Test cases about the basics and configuration were processed first and prioritized according to its expected complexity and dependencies. Stories about the tickets creation and management, which depend on previously tested configurations, were also processed according to the expected complexity and dependencies, in the second phase of the TDCM application.

The processing order of test cases during the TDCM application was determined by the defined stories associated to each test case. In each TDCM iteration, the test case that was processed was the one that specified a story that had the minimum number of dependencies with those stories that were not processed yet. This rule was applied in order to minimize the necessity of rewriting test cases and to improve the efficiency of TDCM.

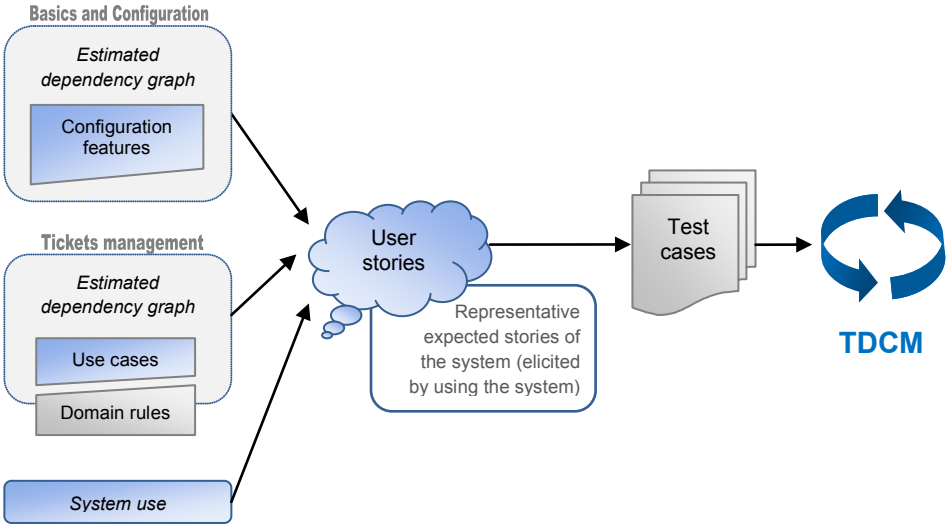


Fig. 52. Testing/development strategy (osTicket case study)

9.1.3 Reservations and Old People's Home Case Studies

In this section, we report two application cases which were conducted in the context of a Requirements Engineering master course at the Barcelona School of Informatics (UPC–BarcelonaTech). Similar experiences have been reported in the literature for TDD in programming (Edwards 2003).

Two groups of six master students used TDCM in two consecutive editions of the course.

The first group developed the conceptual schema of an online event ticketing system by using the innovative ideas of the *Yuuzuu Flexible Reservation* method (Miyashita et al. 2008). The *Yuzuu* method allows users to find, select and purchase tickets for events by introducing cooperation among customers with different preferences for the provided services. It admits regular reservations, flexible reservations (cheaper reservations in which session event and seat are admitted to change) and picky reservations (reservations for specific sessions or seats which are arranged by changing flexible reservations).

In the second edition, the challenge was the development of a system aimed at taking care of the users of old people's homes, taking into account the position of each resident, the care tasks to be provided, the allowed places for each of them, etc.

The objective of the TDCM application in these cases was the evaluation of development of the conceptual schema of a new system. The resultant schemas and the development process were compared with two parallel groups that developed the conceptual schema of the same system in each case, but without using TDCM. After the marks were published, an exhaustive questionnaire was used to collect the opinions of each group member about the conceptual modeling activity.

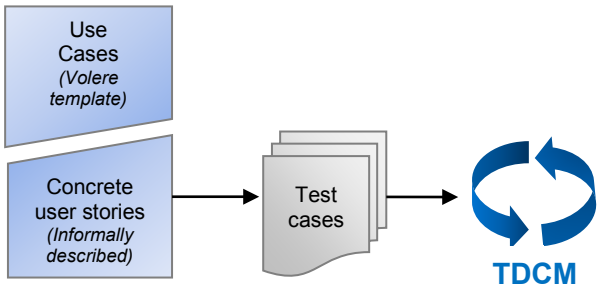


Fig. 53. Testing/development strategy (Yuuzzuu case study)

The source artifacts for the TDCM application were the previously defined use cases specification and concrete user stories for each use case. These artifacts were part of the requirements specification of the system based on the Volere method (Robertson et al. 1997), which was developed in the first part of the course. The user stories were quickly sorted according to its complexity, but no testing strategy was previously defined.

Fig. 53 sketches the TDCM development strategy used in the development of the conceptual schema of the present case study. The source artifacts for the TDCM application were the previously defined use cases specification and concrete user stories for each use case. These artifacts were part of the requirements specification of the system based on the *Volere* method (Robertson et al. 1997), which was developed in the first part of the course for both groups involved in the experience. The user stories were quickly sorted according to its complexity. However, no testing strategy was previously specified in these cases.

9.2 The Bowling Game Case Study

In this section, we describe the results of applying TDCM in the development of the conceptual schema of the bowling game case study introduced in Section 9.1.1. We analyze the resultant artifacts, the testing effort, the errors and failures that drive the development, and the properties of the TDCM iterations. We performed 12 iterations. 1,78 hours were invested in specifying the test cases and 4,59 hours were invested in the development of the TDCM iterations. The full details of the case study may be found in the research report (Tort 2011a).

9.2.1 The Resultant Conceptual Schema and the Test Set

The TDCM application was finished when two conditions held: 1) we formalized as test cases all representative stories, 2) the verdict of the whole test set became *Pass*.

The resultant conceptual schema consists of 3 classes, 10 attributes, 8 associations, 8 derivation rules of derived attributes and associations, 2 event types and 6 integrity constraints.

The resultant test set consists of 754 lines of test cases that are part of 10 test cases. The test set ensures that the resultant schema is correct and complete according to the test cases (the defined knowledge makes the stories feasible and fulfills the expectations formalized as test assertions).

9.2.2 Errors and Failures

We categorized the errors and failures obtained when applying TDCM in this case study. Neither syntactical errors nor incorrectly formalized expectations in test cases were considered in this analysis. Table 5 shows the categorization and the applicable changes to fix each error and failure.

Code	Description	Suggested changes to the schema to fix the error/failure	Number of times revealed	Perc. (%)
Rel_BT	An expected relevant base type (entity type or attribute or association) is not specified in the conceptual schema.	Specify the base type in the conceptual schema.	14	22,6
Rel_DT	An expected relevant derived type is not specified in the conceptual schema.	Specify the derived type (and its derivation rule) in the conceptual schema.	8	12,9
Rel_ET	An expected relevant event type (domain event or query) is not specified in the conceptual schema.	Specify the event type (and its effect or answer) in the conceptual schema.	2	3,2
EvOc_bef	The IB state before an expected event occurrence is inconsistent (the event specification is invalid).	Some (too restrictive) static constraints or preconditions are updated.	1	1,6
EvOc_after	The IB state after an expected occurrence of an event is inconsistent (the event specification is invalid).	The event postcondition, the event method or a static constraint are updated.	13	21,0
EvOc_post	The postcondition is not satisfied after an expected event occurrence.	Either the method or the postcondition are updated.	8	12,9
Sem_exp	An OCL expression in a test case or in the conceptual schema is not valid or inconsistent (e.g. invalid operations for specific types).	Either the expression in the test case is corrected or an element of the schema needs to be changed according to the semantic error revealed.	4	6,5
Ib_ass	A test assertion about the IB state fails.	The effect of an event type or a derivation rule needs to be corrected.	11	17,7
NonOc_ass	A test assertion about the non-occurrence of an event fails.	An event initial integrity constraint (postcondition) needs to be added/updated.	1	1,6

Table 5. Errors and failures categorization (Bowling game case study)

The chart of Fig. 54 shows the occurrence of the different types of errors and failures, which drove the development of the conceptual schema of the bowling game case study. We observe that:

- 39% of the errors and failures revealed basic types (Rel_BT), derived types (Rel_DT) or event types (Rel_ET) which were not defined in the schema.
- 36% of the errors/failures revealed incorrect definitions of domain event types, either because the state before the occurrence was inconsistent (EvOc_bef), or because the state after the occurrence was inconsistent (EvOc_after), or because the postcondition was not satisfied (EvOc_post).

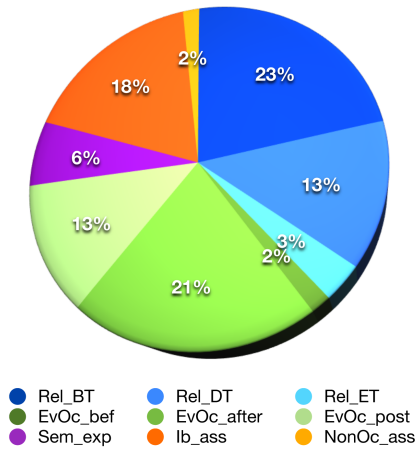


Fig. 54. Errors and failures revealed (Bowling game case study)

- 20% of the errors correspond to unexpected results (assertions that fail). Most of them are assertions about the Information Base (IB) state (18%). Others are failing assertions about the non-occurrence of events (2%).

- Some iterations have been driven by other semantic errors in OCL expressions (e.g. operations which are applied to invalid element types).

9.2.3 Iterations Analysis

In the following we analyze and compare the 12 iterations (we name them as *it1…it12*) that have been performed by applying TDCM to the bowling game case study.

Fig. 55 represents the testing specification productivity (lines of test added/minute). We observed that, in general, the productivity tends to increase as we make progress in the TDCM application. We also realized that there are peaks of productivity in the iterations that reuse previously used testing structures. In contrast, the testing specification consumed more time when we specified new and non-familiar stories.

Fig. 56 represents the total time spent in each iteration. In most iterations, the time spent in the TDCM development (fixing errors and failures) is greater than the time spent in the specification of test cases. It means that in most of the iterations, the testing specification time worth the while because the test case drives the evolution of the conceptual schema. In other words, we realize that the task of writing test

cases (which it should be considered also in the case of test-last strategies) is useful in order to progressively detect errors and failures that guide the conceptual modeling process.

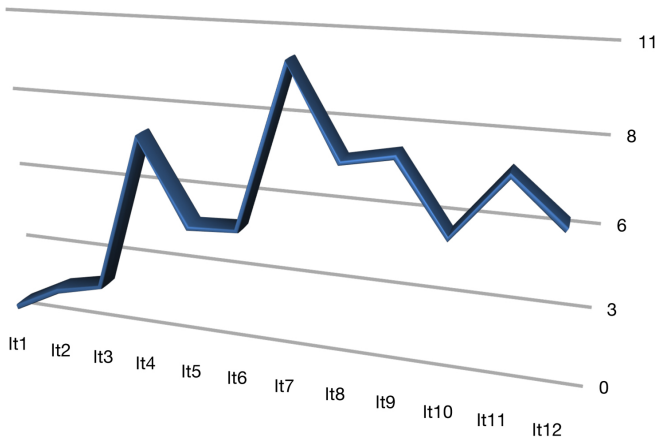


Fig. 55. Testing specification productivity in the bowling game case study (lines per minute)

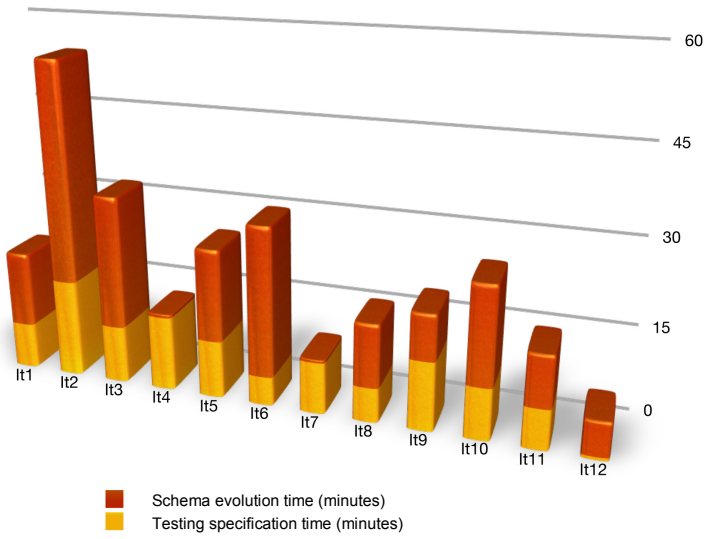


Fig. 56. Total time per iteration in the bowling game case study

Iterations *It4* and *It7* are exceptions. In these iterations, the time spent on evolving the schema is insignificant because the result is *Pass* in the first execution. They increase confidence on the validity of the schema but they do not drive its evolution.

Fig. 57 illustrates the distribution of the different kinds of errors and failures while TDCM is applied. We can observe that, in first iterations, the main errors found correspond to relevant types that are not in the schema. As we add domain events, inconsistent states that require refining static constraints and correctly specifying the effect of these events were revealed. After the peak of the first initial knowledge to be defined, the following iterations provided other kinds of errors and failures (incorrect derivation rules, inconsistent expressions, etc.) that lead to refine some events or to add new knowledge.

Another observation is that not all kinds of knowledge require the same effort to be evolved or corrected according to the processed test cases. In first iterations, the number of errors and failures is greater because we mainly add relevant knowledge. After that, when we specify the effect of the events and we make assertions about derivation rules or the IB states reached by the events, the required effort is greater because it is less evident how to change the schema in order to reach the verdict *Pass*.

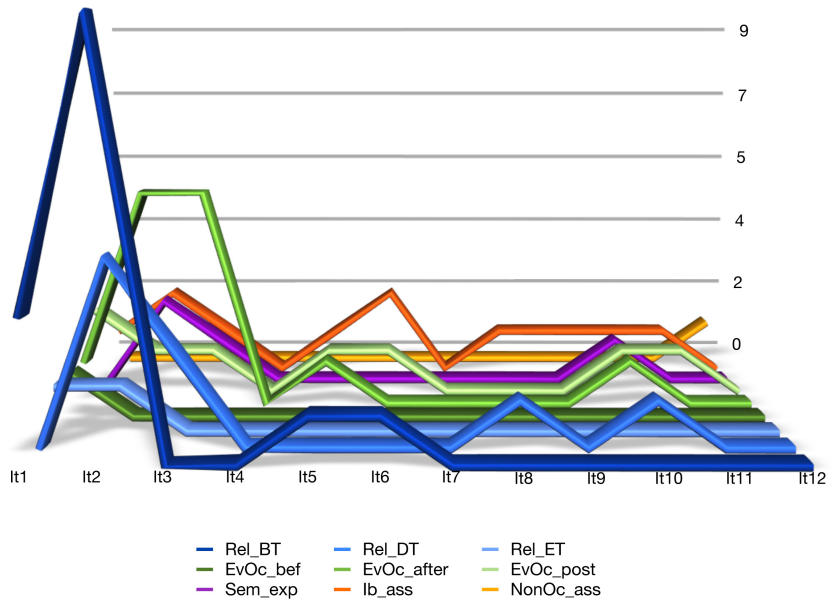


Fig. 57. Distribution of errors/failures throughout iterations (Bowling game case study)

9.3 The OSTicket Case Study

In this section, we analyze the results of the development of the conceptual schema of the *osTicket* system presented in Section 9.1.2 by applying TDCM. In this case study, we analyze the same aspects described in the bowling game case study in order to compare and verify the results by taking into account another experience in the development of the conceptual schema of an existing real-sized system.

We performed 100 iterations. There was not any intention for a particular number of iterations. 6,38 hours were invested on test cases specification and 13,8 hours were spent in the evolution of the schema under development. The full details of the case study may be found in (Tort 2011b).

9.3.1 The Resultant Conceptual Schema and the Test Set

The resultant conceptual schema consists of 28 classes, 92 attributes, 44 associations, 24 event types, 3 derivation rules and 51 integrity constraints. In contrast with the conceptual schema of the bowling game system, where derived knowledge characterizes the system, in this case there are several event types which increase the complexity of the behavioral part of the schema. The resultant test set consists of 2002 lines which are part of 101 test cases and 25 fixture components (reusable initial states shared by different test cases). Table 6 summarizes general and aggregated data about the case study.

osTicket Conceptual Schema

Classes	28
Attributes	92
Associations	44
Event types	24
Integrity constraints	51

osTicket Test Set

Number of test cases	101
Lines of test cases	2002
Fixture components	25

TDCM iterations

Number of iterations	100 iterations
Total development time of the iterations	20 hours 11 minutes
Total time to specify test cases	6 hours 23 minutes
Total time to evolve the conceptual schema under development	13 hours 48 minutes
Average of lines per test case	20,02 lines
Average of testing specification time per iteration	7,6 minutes
Average of conceptual schema development time per iteration	16,4 minutes
Average of changes to the conceptual schema per iteration	4 changes

Table 6. Aggregated data about the osTicket Case Study

The schema is correct according to the expectations formalized in the test set (the knowledge included in the conceptual schema fulfills the expectations formalized as test case assertions) and complete (the knowledge it contains makes feasible the test set execution).

However, more user stories could be designed and, consequently, more test cases could be specified in order to increase our confidence about the correctness and the completeness, by testing the schema in more representative cases. This is a drawback inherent to all the testing processes, because the number of possible test cases is infinite. In this case study, we learned that it is very important to specify the test cases based on a representative set of user stories according to a planned testing strategy.

9.3.2 Errors and Failures

In this case study, we applied the same categorization of errors and failures obtained in the previous case study. Table 7 shows the categorization of errors and failures and its frequencies revealed in the present case study. We realized that this categorization and the suggested associated actions are useful guidelines to help making progress in TDCM. The description of the suggested actions was refined and a new kind of failure was added (failing assertions about the consistency of a state). The new kind of error is only applicable when states are checked without the use of events.

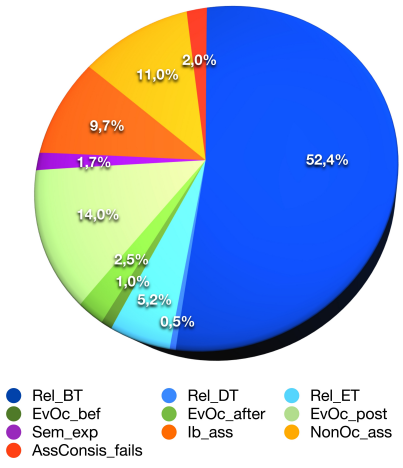


Fig. 58. Errors and failures revealed (osTicket Case Study)

Code	Description	Suggested changes to the schema to fix the error/failure	Number of times revealed	Perc. (%)
Rel_BT	An expected relevant base type (entity type or attribute or association) is not specified in the conceptual schema.	Specify the base type in the conceptual schema.	210	52,4
Rel_DT	An expected relevant derived type is not specified in the conceptual schema.	Specify the derived type (and its derivation rule) in the conceptual schema.	2	0,5
Rel_ET	An expected relevant event type (domain event or query) is not specified in the conceptual schema.	Specify the event type (and its effect or answer) in the conceptual schema.	21	5,2
EvOc_bef	The IB state before an expected event occurrence is inconsistent (the event specification is invalid).	Some (too restrictive) static constraints or preconditions are updated.	4	1,0
EvOc_after	The IB state after an expected occurrence of an event is inconsistent (the event specification is invalid).	The event postcondition, the event method or a static constraint is updated.	10	2,5
EvOc_post	The postcondition is not satisfied after an expected event occurrence.	Either the method or the postcondition is updated.	56	14,0
Sem_exp	An OCL expression in a test case or in the conceptual schema is not valid or inconsistent (e.g. invalid operations for specific types).	Either the expression in the test case is corrected or an element of the schema needs to be changed according to the semantic error revealed.	7	1,7
Ib_ass	A test assertion about the IB state fails.	The effect of an event type or a derivation rule needs to be corrected.	39	9,7
NonOc_ass	A test assertion about the non-occurrence of an event fails.	An event initial integrity constraint (postcondition) needs to be added/updated.	44	11,0
AssConsis_fails	A test assertion about the consistency of an IB state fails.	A static constraint prevents the IB state to be consistent and it is updated.	8	2,0

Table 7. Errors and failures categorization (osTicket case study)

Fig. 58 shows the occurrence of the different types of errors and failures which lead the changes to evolve the schema when TDCM was applied in the *osTicket* case study. We observe that TDCM drove the development of the conceptual schema by asking to fix three main kinds of errors and failures:

- 58,1% of the errors and failures revealed basic types (Rel_BT), derived types (Rel_DT) or event types (Rel_ET) which were not defined in the schema.
- 17,5% of the errors/failures revealed incorrect definitions of domain event types, either because the state before the occurrence was inconsistent

(EvOc_bef), or because the state after the occurrence was inconsistent (EvOc_after), or because the postcondition was not satisfied (EvOc_post).

- Other errors correspond to unexpected results (assertions that fail). Most of them are assertions about the non-occurrence of events (11%), and about the IB state (9,7%). Others are assertions that check the consistency of an IB state (2%).

- In this case study, a few iterations were driven by other semantic errors in OCL expressions, such as incompatible types or invalid operations for some types.

9.3.3 Iteration Analysis

Fig. 59 represents the testing specification productivity (added/updated lines of test cases per minute). We observe that, in general, the productivity tends to increase and decrease periodically.

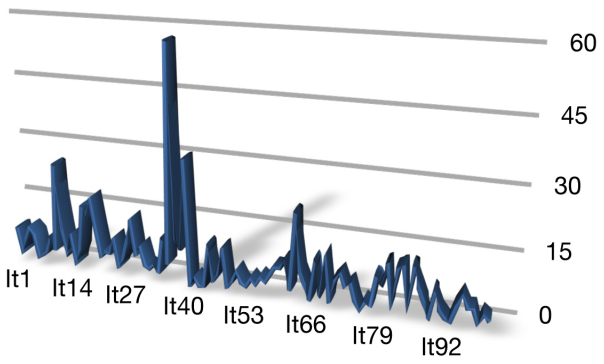


Fig. 59. Testing specification productivity in the osTicket case study (test cases lines per minute)

If we analyze the iterations, we may observe that test cases may be grouped into similar stories (e.g. stories which are tested with variations or using different initial states or conditions). The first time we specify a test case associated to different testing objectives, the testing specification productivity decreases. However, when we specify story variations, then the productivity increases.

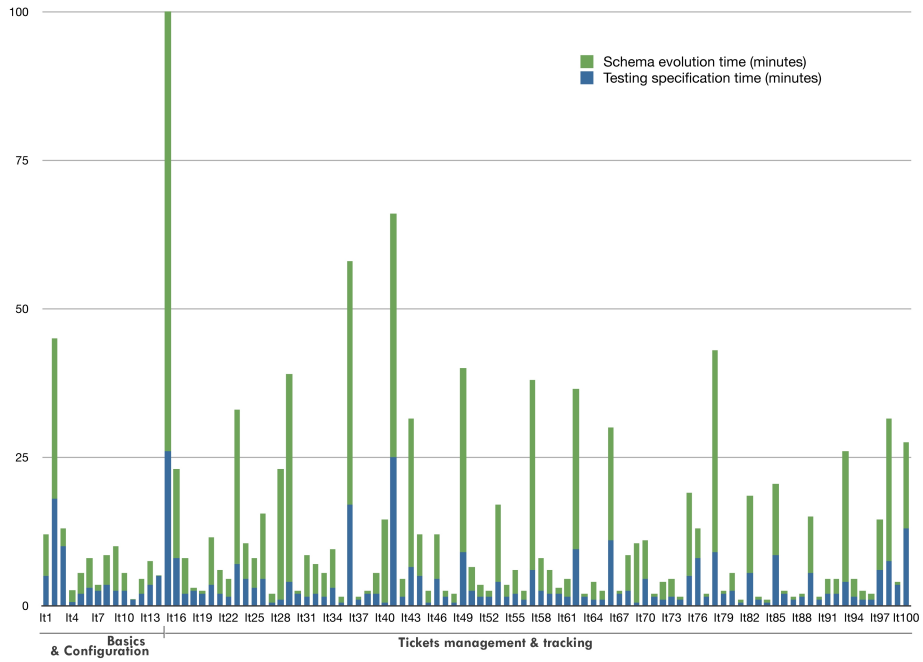


Fig. 60. Total time per iteration (osTicket case study)

We realize that there are peaks of productivity in iterations in which previously used testing structures are reused. In contrast, the testing specification consumes more time when we specify stories with new (and probably unknown) structures.

The chart presented in Fig. 60 analyzes the total development time that were used to complete each iteration. As it happens in the previous case study, in most iterations the time spent in performing changes to evolve the schema (fixing errors and failures) is greater than the time spent in the specification of test cases. Therefore, we observe again that the testing specification time worth the while because many test cases encourage the conceptual schema to be evolved. Similarly to the previous case study, the exceptions are those iterations which pass in the first execution. They only increase confidence about the correctness of the schema but no schema changes are induced. In these iterations, the time spent by designing and specifying the test case is higher in comparison with the TDCM iteration time spent.

Fig. 61 shows a diagram that summarizes the occurrence of errors and failures throughout the iterations performed in this case study. We can observe that, in first iterations, the main errors and failures found are about relevant types that are not in the schema. These iterations correspond to the processing of test cases about the

basics and configuration of the system. Once the main static schema elements are specified in the schema, the type of failures and errors that drive the schema evolution change significantly. As we add the first domain events, we detect inconsistent states which require refining static constraints and correctly specifying the effect of these events, but only some static knowledge is required to be added.

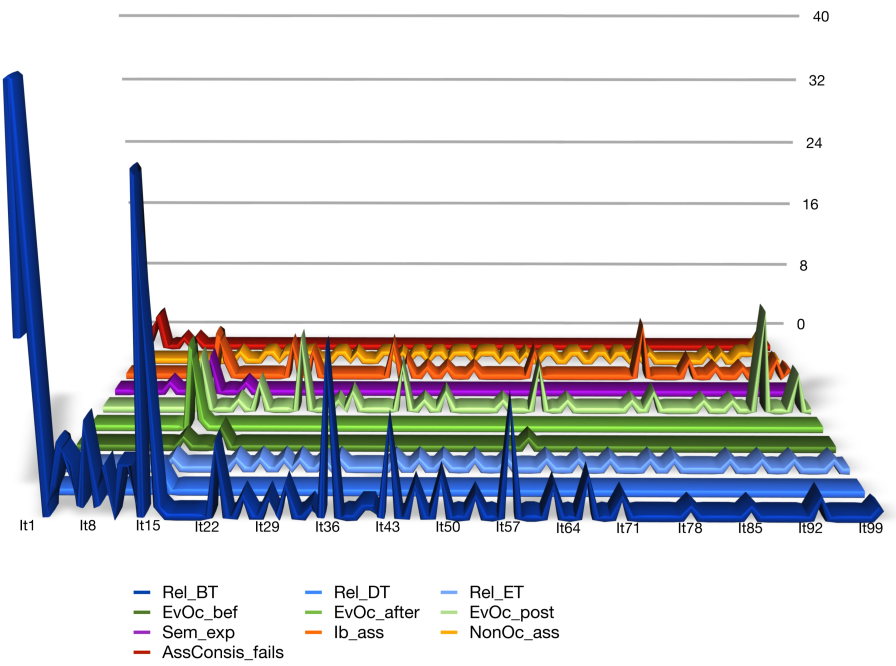


Fig. 61. Distribution of errors/failures throughout iterations (osTicket Case Study)

The analysis of the distribution of errors and failures (Fig. 61) together with the time spent on the evolution of the schema (Fig. 60) suggests that not all kinds of knowledge require the same effort to be fixed. In the first iterations the number of errors/failures is greater, mainly because only add relevant knowledge was added to the schema. After the addition of the basic static knowledge (which was necessary to support the execution of the following test cases), test cases that represent sequences of event occurrences (common user stories) were processed. When the specification of the events effect was defined and assertions about the IB states reached by the events failed, the required effort to fix them was greater, because it is less evident how to change the schema in order to reach the verdict *Pass*.

9.4 Event Reservations and Old People's Home Case Studies

In this section, we interpret and analyze the results of the application of TDCM for the two case studies presented in Section 9.4. These cases were developed by master students in the context of a requirements engineering course. Two different groups of master students developed each conceptual schema in each project from the same project formulation. For each project edition, one of the groups developed the schema by using TDCM. The objective of these cases studies was different from the previous ones. In this case, the main aim was collecting the opinions of developers who used the method for the first time.

9.4.1 Resultant Conceptual Schema Quality

The evaluation of the resulting schema delivered by each group showed that the semantic quality (correctness and completeness) of schemas developed by using TDCM is significantly higher in comparison with schemas developed by students who did not apply TDCM. Firstly, because the schema developed by TDCM was executed. Secondly, it was developed in conjunction with a test set that guides the development and enhances the validation of the schema.

We also observed that schemas developed without TDCM contain incorrect knowledge such as unsatisfiable constraints, incorrect knowledge according to the requirements specification or inconsistencies between the structural schema and the behavioral schema. These kinds of errors were not present in the schema which was developed with continuous testing by applying TDCM.

9.4.2 Observation of the Method in Practice

The development of the schemas was observed by the teachers during the project course sessions. The observations revealed that the groups of students that used TDCM developed the schema in a more systematic way. The objectives to be achieved in order to develop the schema were clearly stated at the beginning of each session and the test set was used to check the evolution of objectives and the remaining work to be done.

The participants who used TDCM were supported by continuous feedback. They also had executable tests to support the resolution of different points of view about the knowledge to be added and about inconsistent requirements.

9.4.3 Opinions about Use

The opinions of the participants were collected through a survey once the marks were published. The survey included several questions with predefined answers: (1) the degree of difficulty experienced for each kind of knowledge, (2) the importance of the test set for the confidence on the schema quality, (3) the difficulties related to reconsideration of previously defined knowledge, and (4) the general impression about the utility of the method.

The analysis of the opinions suggests that those students who apply TDCM are more confident about the semantic quality of the delivered schema. Note that those who did not use TDCM did not have the support of test cases to obtain feedback (only the questions answered by the teachers and the comments about a preliminary delivery). However, students who applied TDCM report that the time invested to develop the conceptual schema was greater (especially in initial iterations, where they need to be familiarized with the testing environment). Another TDCM feature that was positively rated is that TDCM provides a systematic way of performing conceptual modeling. Most of the students also remarked the importance of dealing with focused objectives as a way to avoid getting lost in a growing conceptual schema. Nevertheless, some of them considered that more effort is needed to develop more user-friendly tools to manage the application of TDCM.

The participants were also asked to write between three and five advantages and drawbacks of the method according to their experience. The results are summarized in Table 8.

Advantages		Drawbacks	
Confidence on quality / executability of the schema	45,5%	Technical improvements (parsing, text editor, etc.)	33,3%
Feedback during the conceptual modeling activity	27,3%	Testing effort consumes time	25,0%
Focus on small objectives / Work organization	18,2%	More detailed errors and failure information	25,0%
Consistency between structural and behavioral schemas	9,1%	More cooperative elements in the supporting tool	16,7%

Table 8. Advantages and drawbacks expressed by participants

9.5 Lessons Learned

The successful application of Test-Driven Conceptual Modeling (TDCM) in the four case studies reported in Sections 9.2, 9.3 and 9.4 shows that this novel method (which is one of the main contributions of this Thesis, presented in Chapter 8) may be applied in practice.

As a result from these application case studies, we categorized the kinds of errors and failures that drive TDCM iterations (see Table 5 and Table 7). We classified the most common actions to fix them in order to evolve the conceptual schema under development. Furthermore, we realized that this categorization is a useful guideline for future applications of TDCM, which complements the guidelines explained in Section 8.3.

Additionally, subjective information about the use of the method was also collected through questionnaires and direct observation during the application of TDCM by the participants of the *Yuuzuu* reservation system and the old people's home case studies. The analysis points out that TDCM contributes to increase the confidence on the quality (correctness and completeness) of the schema. The obtained feedback during schema evolution, the controlled resolution of inconsistent requirements and the process of taking focused decisions during conceptual modeling are also remarked as other significant contributions of TDCM.

Given that TDCM is developed in the context of design research (Hevner et al. 2004), the reported case studies were useful to improve by use the supporting tool. We also refined the information provided by the test processor in order to enhance the response to errors and failures information in future TDCM applications.

Finally, we experienced the importance of defining a testing strategy in the context of the general requirements engineering method in which TDCM is applied (see Section 10). The refinement of previously defined test cases may be minimized by defining a strategy that includes the order of testing based on the complexity and the dependencies between stories.

We consider that the discussed set of test cases validates the viability of TDCM and serves as an analysis to identify the main properties that characterize the TDCM iterations. However, more experimentation in industrial contexts should be done in order to enhance the evaluation of this new conceptual modeling method.

In summary, the analysis of the data collected in the case studies allows learning lessons about the resultant conceptual schema, the testing effort, the iterations productivity and the common iteration patterns have been learned and discussed. A summary of these lessons learned is discussed in the following.

9.5.1 The Viability of TDCM

The initial objective of the present work was checking the viability of TDCM. This objective included the applicability of the TDCM cycle, the use of the automated testing language and the adequacy of the supporting tool.

In the four case studies, TDCM was successfully applied and improvement opportunities were revealed. We realized, for example, that is useful and possible to apply TDCM with more than one test case per iteration. We also experienced that the general development method and the context in which TDCM is applied constraints the elicitation of the stories to be processed. The reported case studies show that TDCM is a viable method in practice.

It is important to remark that, in these experiments, all the evolution changes in the schema have been systematically performed when an error or a failure has driven it. However, TDCM is adaptable to the experience of the modeler. Smaller changes and more frequent checking of the testing verdicts provide more failing/error information to guide the definition of the schema. More regression testing provides more confidence about the already included knowledge.

9.5.2 Conceptual Schema Quality

After the application of TDCM in the three discussed case studies, we obtained two artifacts: (1) an executable conceptual schema, and (2) a test set that validates the correctness and the completeness of the schema according to the expectations formalized in the test cases.

It is important to note that, in contrast with traditional conceptual modeling approaches, we additionally obtain a test set which provides a high level of confidence on the quality of the schema. The opinions expressed by the participants of the case studies point out that the test set is valuable for self-confidence on the quality of the schema and also as a transparent and executable artifact to externally prove its quality.

9.5.3 Errors and Failures to Drive Conceptual Modeling

In TDCM, the schema evolution depends on the errors and failures information obtained by the automated execution of test cases.

In the bowling game and the *osTicket* case studies, we focused on the analysis of the kinds of errors and failures that drive schema changes. Table 5 and Table 7 show the categorization of the different kinds of errors and failures. We also

categorized the common actions to solve each error/failure. We first performed a first version of the categorization during the bowling game case study. After that, during the application of TDCM in the *osTicket* report, we refined the categorization and we observed that it was very helpful for the modeler, because it suggests fixing actions when a new failure or error needs to be processed.

9.5.4 Testing Effort

The reported case studies suggest that writing and managing test cases implies specification effort. Nevertheless, the test set is highly rated as useful or providing feedback during the conceptual schema evolution. It is also considered to be a valuable artifact because it provides justification of the validity of the resultant conceptual schema. In the context of automated transformation of conceptual schemas into executable code, the test set corresponds to the conventional program test set.

It is important to remark that, in the four case studies, the time spent to specify the test cases in each iteration varies depending on the complexity of the formalized stories, according to each testing objective. In the reported case studies, we also observed that in most iterations, the testing effort time is significantly lower than the time spent on evolving the schema. This is an important observation for the feasibility of TDCM: the testing effort worth a while taking into account the continuous feedback and the schema evolution time.

Another observation is that the testing specification productivity varies periodically: when a new story with different knowledge needs to be formalized, the specification effort increases; when variants of a story are formalized as test cases, then the specification effort decreases. The reason is that when testing structures are reused the testing specification productivity increases.

Finally, we also observed that as the testing language is more familiar to the modeler, the testing specification productivity tends to increase.

9.5.5 Iterations Productivity

The bowling game and the *osTicket* case studies analysis shows that the time spent in the conceptual schema evolution (by fixing errors/failures) is greater than the time used to specify the test cases. Therefore, most of the test cases are productive because they lead to make progress in the evolution of the schema. The exception are those iterations that pass in the first execution (they increase our confidence about the validity of the schema, but they do not drive changes).

9.5.6 Iteration Patterns

By analyzing the reported studies in conjunction, we found the following iteration patterns:

- *The kinds of errors and failures that drive the schema evolution are not uniformly distributed in the iterations*: In first iterations, the most common errors are about missing relevant types (which are necessary to build IB states). After that, the most common errors/failures are about the correct definition of domain event types and the correctness (according to the assertions) of the reached IB states.

- *The time spent on evolving the schema depends on the kind of errors and failures to be fixed*: We observed that not all errors/failures required the same effort to be fixed. This analysis suggests that missing relevant types are trivial to be fixed (they need to be added). However, the changes to fix failing assertions about the state of the domain or incorrect domain event specifications may require more complex actions such as changing derivation rules, integrity constraints or the precondition and postcondition of the effect of the event.

These observations may be useful to estimate the development time when using TDCM.

9.5.7 Feedback and Guidance

The participants involved in the RE course case studies highly rated that, in TDCM, the conceptual schema is developed accompanied by a validation test set. They pointed that one of the main advantages of TDCM is the constant feedback provided in each iteration. In contrast, the students who did not use TDCM only obtained feedback from the teachers in the provisional delivery.

It is important to remark that the teachers observed that the management of the development was better organized and clear. The students also commented this feeling in several occasions. The resolution of conflicts between different points of view about the knowledge to be added in the schema led to long discussions in the group that did not apply TDCM. The main reason was that discussions were usually about the whole conceptual schema without centering on local decision points. In the group that applied TDCM, discussions about the knowledge to be added to the schema were focused in the testing objective to be achieved in each moment.

Finally, the observed experience suggests that traceability between the different requirements engineering artifacts (including the conceptual schema) is higher

when TDCM is applied, because test cases are based on user stories which belong to use cases. In each moment, the passing test cases determine which stories the conceptual schema under development admits.

9.5.8 Testing/Development Strategy

The application of TDCM in the presented case studies reveals the necessity of a testing strategy to support TDCM, which should be defined prior to the method application. We envisioned that the incremental development of conceptual schemas could imply the necessity of rewriting test cases to maintain consistent stories while the set of valid states changes. Given that test cases drive the development, in TDCM the testing strategy is, in fact, the development strategy.

We have observed that in the bowling game and the *osTicket* case studies, the rewriting testing effort and the reconsideration of previously defined knowledge is irrelevant. In both cases, the definition of a testing strategy prior to the TDCM application was performed. The strategy took into account the dependencies and the complexity of the user stories. The reported case study suggests that when the strategy is aligned to the incremental nature of TDCM and minimizes the necessity of testing rewriting.

In contrast, in the case studies developed in the requirements engineering course no previously specified testing strategy was formalized. The developers who applied TDCM reported that they needed to reconsider previously defined knowledge and test cases several times. Nevertheless, they expressed that regression test cases were useful to facilitate this task. The group who did not apply TDCM expressed that going backs and reconsideration of knowledge was the main issue and the most difficult one to be solved.

10 Integrating TDCM into Existing Software Methods

In Chapter 8, we defined the TDCM method regardless the general method in which it is applied. In this chapter, we present an integration approach of TDCM into the Unified Process (Section 10.2), into Model-Driven Development (Section 10.3), into goal and scenario-oriented methods in Requirement Engineering (RE) (Section 10.4), and into hybrid methods that combine agile storytest-driven development with conceptual modeling (Section 10.5). These integration proposals are based on the context and notation explained in Section 10.1.

10.1 Integration Context

Conceptual schemas are usually developed in the context of existing requirements engineering or general software development methods. The Test-Driven Conceptual Modeling (TDCM) method presented in Chapter 8 is applicable to different kinds of projects and may be integrated into existing software development methods.

Nevertheless, it is important to note that TDCM may be used even if the conceptual schema to be developed is the main purpose of the project. For example, TDCM can be applied when the developed conceptual schema is going to be used as a reference model, when it is a metaschema to be instantiated in several applications or in conceptual modeling education to assist students and novel designers to develop conceptual schemas.

Fig. 62 presents the context in which TDCM is used within a general method that includes conceptual modeling as a Requirements Engineering (RE) activity. The general method and the integration approach condition the *testing strategy* for applying TDCM. Its main purpose is determining (1) the source artifacts available when starting the TDCM application, (2) how test cases are derived from the source artifacts, (3) the iteration sequence, and (4) the finalization condition of TDCM. Since TDCM is driven by tests, the testing strategy is, in turn, the strategy for the development of the conceptual schema. Therefore, in this document, we simply refer it as *testing* or *development strategy*.

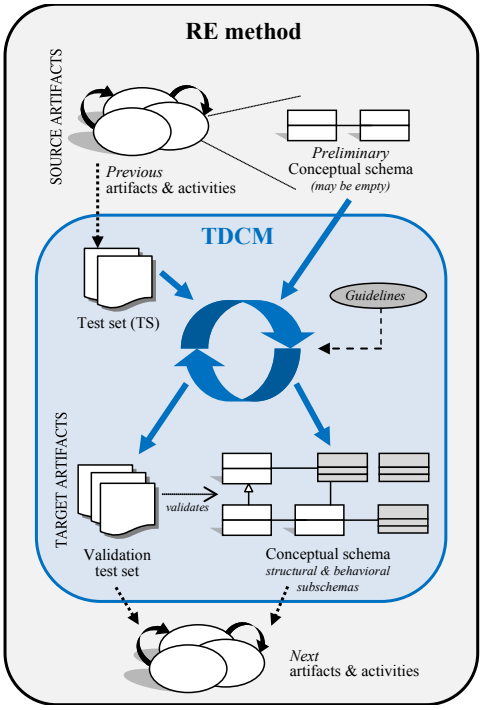


Fig. 62. TDCM context

Table 9 summarizes the integration approaches discussed in the following sections. For each case, we define the artifacts that may be used to specify the conceptual schema (*target artifacts*) and we describe an integration context to develop them by applying TDCM. We also suggest the artifacts that may be used as a base for the TDCM application (*source artifacts*). Source artifacts may contribute to suggest test cases for the TDCM application. These artifacts can also contribute to prioritize and plan the processing order of test cases. Finally, we describe the *finalization condition* of the TDCM iteration sequence.

Integration method		Source artifacts	TDCM iteration sequence	TDCM finalization conditions	Target artifacts
Unified Process (UP) ***Larman, C.*** (Larman 2005)		▪ Use case model ▪ System Sequence Diagrams (SSDs) ▪ Domain model (may be empty)	Determined from the order in which use cases are iteratively defined.	For each use case scenario, there is at least a representative concrete scenario formalized as a test case.	▪ Structural design model ▪ System events specification (contract-based) ▪ Validation test set
Model-Driven Development (MDD) ***MDA*** (Mellor et al. 2002, Pastor et al. 2007, Object Management Group (OMG) 2003)		▪ Use case model ▪ Scenario models (specified as sequence diagrams) ▪ Preliminary conceptual schema (may be empty)	Determined from the defined use cases and scenarios.	For each use case scenario, there is at least a representative concrete scenario formalized as a test case.	▪ Structural schema (PIM) ▪ Behavioral schema (PIM) ▪ Validation test set
Storytest-Driven agile methods ***Storytest-Driven Development*** (Mugridge 2008)		▪ User stories informally defined ▪ Structural schema sketch (may be empty)	Determined from the order in which user stories are iteratively defined.	For each user story, there is at least a test case that formalizes the story.	▪ Structural schema ▪ Behavioral schema ▪ Validation test set
Goal and scenario-oriented methods ***GORE*** (Van Lamsweerde 2009)	*Specification of operations*	▪ List of operations ▪ Object model	Determined from the list of operations which are expected to *operationalize* the goals.	For each operation signature, there is at least a test case that tests it in a consistent state.	▪ Operational model: operations specification to *operationalize* the goals. ▪ Object model ▪ Validation test set
	Object and operation model refinement and validation according to scenarios	▪ Operation model ▪ Behavioral Model (scenarios)	Determined from the defined scenarios.	For each defined scenario, there is at least a representative concrete scenario formalized as a test case.	▪ Operation model ▪ Behavioral model ▪ Object model ▪ Validation test set

Table 9. TDCM integration summary

10.2 Unified Process

The Unified Process (UP) (Larman 2005) is an iterative method for software development. UP organizes the development process into five phases (Inception, Elaboration, Construction and Transition). In each phase, artifacts are started or refined by applying a practice. UP proposes several practices and guidelines to develop each artifact. Each UP project is a particular instantiation of the process that selects a subset of the proposed artifacts and the convenient practices used to develop them.

UP encourages including new practices from other iterative methods. (Larman 2005) explains that "the set of possible artifacts described in the UP should be viewed like a set of medicines in a pharmacy. On a UP project, a team should select a small subset of artifacts that address its particular problems and needs".

In the UP, the conceptual schema is specified in several artifacts (target artifacts). The structural schema is defined in a UML class diagram in the design model. System events are specified as operations and their effects are defined in a contract with pre and postconditions, which can be formally expressed in OCL. By applying TDCM, we can obtain, in a systematic way, the conceptual schema specified in the structural design model and the formal specification of the events (and its procedural methods). We also obtain a test set that validates the schema.

The application of TDCM in a UP project should be performed at the elaboration phase (Fig. 63). UP proposes guidelines and techniques to develop the conceptual schema. UP suggests reusing reference schemas for many common domains, managing category lists or analyzing use case sentences to find concepts and relationships between them. We propose TDCM as a UP practice aimed at developing the structural and the behavioral parts of the conceptual schema of an information system under development.

The source artifacts for the TDCM application are:

- Domain model: It is a preliminary sketch with some static entity and relationship types used in the inception phase. This model, if not empty, can be used as the initial schema to be evolved by applying TDCM.

- Use case model: It specifies the expected functionalities of the system. It is composed by the *use case diagram* and textual specifications of the representative system-actors interactions for each use case.

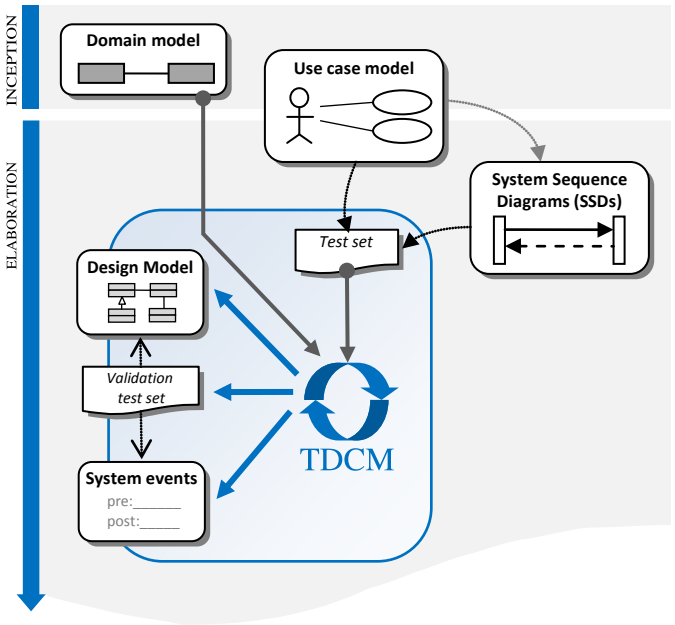


Fig. 63. TDCM integration into the Unified Process

- System Sequence Diagrams (SSD): UP considers that a use case is often too complex to be defined or processed in a short iteration. Therefore, SSDs are used to specify scenarios for each use case. A test case to validate each SSD diagram can be written to drive the definition of the conceptual schema by applying TDCM.

The TDCM iteration sequence would be determined by the use cases and their corresponding SSDs. For each representative use case scenario, there should be at least one test case to be processed in TDCM iterations. The correctness, the relevance and the completeness of the source artifacts (use cases and their corresponding SSDs) contribute to the quality of the derived test cases, and these are pursued by the integration method. The iteration sequence ends once all use case scenarios have been considered.

Additionally, we obtain the event methods (a procedural specification of their effects). Event methods are of particular interest when a UP project uses UML/OCL conceptual schemas as a blueprint (relatively detailed model to allow code generation) or as a programming language (a complete executable specification in a conceptual schema centric development context (Olivé 2005)).

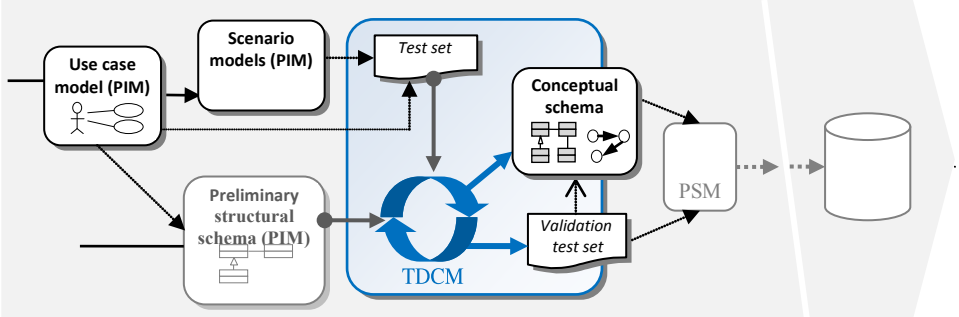


Fig. 64. TDCM integration into an MDD approach

10.3 MDD Approaches

MDD (Mellor et al. 2002, Pastor et al. 2007, Object Management Group (OMG) 2003) is a software development paradigm in which a system is developed as a set of transformations from Platform-Independent Models (PIM) to Platform-Specific Models (PSM).

The Object Management Group (OMG) adopted the Model-Driven Architecture (MDA) (Pastor et al. 2007, Object Management Group (OMG) 2003) to support MDD. MDA prescribes certain kinds of models to be used in MDD and specifies relationships between these models.

In MDD approaches, conceptual schemas are specified as PIMs. (Mellor et al. 2002) suggests defining the structural schema in a UML class diagram and the behavioral schema as state machines with procedures that specify events as a sequence of actions. TDCM can be used to incrementally define and evolve these models (target artifacts).

In MDD, the UML class diagram is obtained as a manual abstraction process from the domain, assisted by guidelines. MDD promotes the use of executable conceptual schemas in order to validate and verify them. In particular, (Mellor et al. 2002) proposes to check class diagrams with expected instantiations. In this context, TDCM is a systematic method that puts this idea into practice while defining the schema.

Before modeling the behavioral schema in UML artifacts, it is suggested to describe use cases and scenarios represented as sequence diagrams.

The source artifacts for applying TDCM are:

- Preliminary structural schema: It is a PIM with static knowledge manually derived from previous artifacts such as the use case model. This model, if not empty, can be used as the initial schema to be evolved by applying TDCM.
- Use case model: It specifies the expected functionalities of the system. It is composed by the *use case diagram* and textual specifications of the representative system-actors interactions for each use case.
- Scenario models: They are PIMs that specify representative use case scenarios as sequence diagrams.

The TDCM iteration sequence would be determined by the use cases and their corresponding scenario models. For each representative use case scenario, there should be at least one test case to be processed in TDCM iteration. The iteration sequence ends once all use case scenarios have been considered.

After the application of TDCM, we obtain a PIM that models the conceptual schema (comprising both the structural and behavioral aspects) of the system-to-be. The resultant conceptual schema is correct and complete according to the set of concrete scenarios specified as test cases. The resulting test set validates these quality properties.

Moreover, we obtain a set of methods that specify in a procedural way the effect of the events, which make the conceptual schema executable. Methods can be used to facilitate the transformation from PIMs to PSMs, which can be manually performed or automated by model compilers. In contexts in which the conceptual schema is sufficient to obtain executable components of the system (Insfrán et al. 2002, Olivé 2005, Nieto et al. 2010), then no transformation to PSMs needs to be performed, because the conceptual schema is executable software.

10.4 Storytest-Driven Agile Methods

Storytest-Driven Development (SDD) (Mugridge 2008) is an agile method based on eXtreme Programming (XP) principles (Beck et al. 2001). Agile development methods were originally conceived in contrast with plan-driven development. In Storytest-driven development, the dominant form of communicating the requirements is user stories. User stories are domain-oriented descriptions of concrete use examples of the system (Mugridge 2008). Stories are formalized as program tests that are intended to be executed on the implementation. (Koskela 2007) argues that in SDD, tests

should be used as a shared language that forces to transform ambiguous requirements into executable tests. He also summarizes that storytest-driven development is about “specifying by example”.

As explained in the related work reviewed in Chapter 7, many authors advocate that plan-driven development and agile development can and should be used in conjunction. For instance, (Boehm 2002) indicates that “hybrid approaches that combine both methods are feasible and necessary. (Meyer 2008) states that program tests “even a million of them, are instances; they miss the abstraction that only a specification can provide”, so that “tests are not substitute for specifications”. In summary, these authors claim that the main aim of agile development is “efficiently respond to changes”, and models can contribute to agility by providing a base in order to efficiently respond to changes during software development. Class diagram sketches are promoted to “think before you act”.

In projects in which requirements elicitation is performed by capturing stories and the conceptual schema is explicitly modeled in one or more artifacts (target artifacts), then TDCM can be used to define the schema. Stories can be formalized as conceptual schema tests, as it can be observed in Chapter 4. These stories can be executed at the conceptual modeling stage, providing traceability between the captured stories and the test cases that guide the conceptual modeling activity.

In this context, the source artifacts for applying TDCM are the set of user stories and the conceptual schema sketch (if exists).

The target artifacts are the diagrams that specify the conceptual schema in an executable form and at the desired level of detail.

User stories are elicited and informally specified iteratively. The TDCM iteration sequence would be determined by these stories. For each story, there should be at least one iteration (test case) that formalizes it. The iteration sequence ends once all stories have been formalized as test cases and all test cases have been processed.

By applying TDCM, contradictions between stories (i.e. requirements conflicts) can be detected (the verdict of more than one non-passing test cases cannot become *Pass* unless at least one test case is changed). Using the terminology of (Robinson et al. 2003), we can say that there is a negative interaction between the functional requirements captured by the stories, because the satisfaction of one of them is reduced as the result of satisfying the other one.

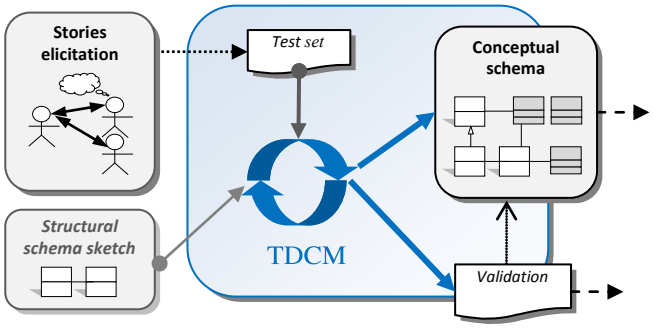


Fig. 65. TDCM integration into Story-test driven Development

Moreover, TDCM can reveal the necessity of describing new stories. If the set of defined stories is prioritized, then TDCM may process user stories according to this prioritization.

10.5 Goal and Scenario-Oriented Methods

Goal-Oriented Requirements Engineering (GORE) advocates the use of goals for requirements elicitation, evaluation, definition and validation. A goal is a prescriptive statement of intent that the system should satisfy through the cooperation of its agents (active system components playing a specific role in goal satisfaction) (Van Lamsweerde 2009).

Several approaches to identify goals and refine them have been proposed. Goals refinement can be modeled in diagrams (Fig. 66). Leaf nodes are goals whose responsibility can be assigned to a single agent. Leaf goals that can be assigned to a single agent are requirements.

In the following we propose the integration context for using TDCM in the goal-oriented requirements engineering approach which has been recently proposed by (Van Lamsweerde 2009). This approach is based on the development of several interrelated models:

- Goal model: It specifies the system's goals in terms of individual features, such as their specification, elicitation source or priority, potential conflicts, etc.

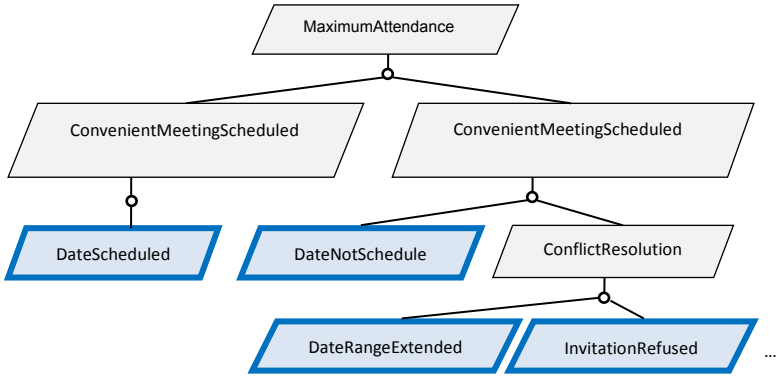


Fig. 66. A goal model fragment of a meeting scheduler system

- Obstacle model: It specifies conditions (in the form of a special kind of goals) that may obstruct the realization of the goals defined in the goal model. It is a goal-form of a risk analysis.
- Object model: It models the structural view of the system by means of the static knowledge that concerns the realization of goals.
- Agent model: It models the responsibility view of the system. It specifies the agents that are responsible for each goal.
- Operation model: It models the functional view of the system. It specifies the operations (by means of pre and postconditions) that *operationalize* each goal.
- Behavior model: It models the behavioral view of the system in terms of scenarios for capturing interaction (operations execution) among agents. In Requirements Engineering, goals and scenarios complement each other (Van Lamsweerde 2009, Pohl 2010, Rolland et al. 2005). Heuristics are suggested to ensure that the set of scenarios cover the goals, by considering both positive and negative scenarios.

In this approach, the conceptual schema is defined by the object model (structural conceptual schema) and the operation model (behavioral conceptual schema).

TDCM may be applied for two different purposes (Fig. 67) in the context of this GORE approach: 1) Specify the operations that *operationalize* the goal model, and 2) refine and validate the object and the operation model according to scenarios.

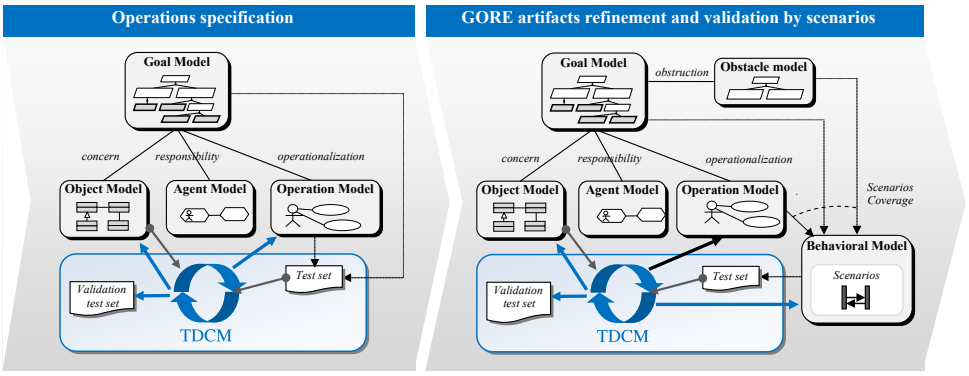


Fig. 67. TDCM integration into a GORE method

In the first case, the purpose is the definition of the effect (pre and postconditions) of the *operation model* and refining the *object model* (target artifacts) according to the *goal model*, the preliminary definition of the *object model* and the list of operations to realize each leaf goal (source artifacts). The TDCM iteration sequence would be determined by the operations list and the goals that they intend to *operationalize*. For each operation, there should be at least one TDCM test case that builds an initial state based on the associated goal description (CSTL allows defining IB states with specific CSTL statements, instead of asserting the occurrence of events (Tort et al. 2010)) and tests the occurrence of the operation. The iteration sequence ends once all operations have been considered. We also obtain a resulting test set that validates that each operation correctly *operationalizes* its associated goal in at least one representative case.

In the second case, the purpose is refining and validating the *object model* and the *operation model* (target artifacts) by considering the scenarios defined in the *behavioral model* and the previous versions of the *object model* and the *operation model* (source artifacts). The TDCM iteration sequence would be determined by the scenarios of the *behavioral model*. For each scenario, there should be at least one test case. The iteration sequence ends once all scenarios have been considered. The resulting test set validates that both the *operation model* and the *object model* allow the execution of the scenarios of the *behavioral model*.

It is important to note that, as proposed by GORE, TDCM preserves the user intention expressed in goals to the conceptual schema, because the knowledge added to the schema is driven from failures/errors produced by the execution of test cases (which are derived from goal-based models).

As explained in Table 3, TDCM can detect conflicts between requirements (which can be graphically represented in the goals model). In general, the use of TDCM may suggest changes (additions, updates) to the goals model.

Techniques for goal risk analysis and prioritization are also proposed in goal-oriented methods. These techniques may be useful to decide the processing order of test cases while applying TDCM.

11

Test Adequacy Criteria for Testing Conceptual Schemas

In this chapter, we present a set of four basic properties for determining the adequacy of a test set aimed to ensure the validity of the knowledge defined in a conceptual schema. These properties, named *test adequacy criteria*, may be used in conjunction with the approach for testing conceptual schemas (Chapter 4) and in Test-Driven Conceptual Modeling (TDCM) (Chapter 8).

The structure of the chapter is as follows. In Section 11.1, we present the four adequacy criteria for testing the necessary conditions for conceptual schema validity. Furthermore, in Section 11.2, we explain a procedure for testing conceptual schema satisfiability by testing, based on the theory that fundaments the basic set of adequacy criteria.

11.1 A Basic Set of Adequacy Criteria for Testing Conceptual Schemas

In previous chapters, we have seen how to test an executable conceptual schema by writing a set of tests and making them pass. However, not all possible test sets are equally adequate to increase the confidence about the semantic quality (validity and completeness) of the conceptual schema.

In Chapter 2, we have seen that a conceptual schema of an information system has semantic quality when it is valid and complete. Validity means that the schema is correct (the knowledge defined in the schema is true for the domain) and relevant (the knowledge defined in the schema is necessary for the system). Completeness means that the conceptual schema includes all relevant knowledge.

Our approach to test conceptual schemas (Chapter 4) is aimed at validating correctness and completeness according to stakeholders' needs and expectations. In our approach, system's functions are captured by a set of expected concrete scenarios written as test cases. When their execution produces the expected results, then we can ascertain that the conceptual schema is complete according to the test set. Otherwise, some test case would not succeed, because the schema does not define some relevant knowledge that is required to execute the test cases. We can also ascertain that the part of the conceptual schema involved in the execution of the test cases is correct according to the expectations defined in the executed test set. Otherwise, some expectation formalized in the test set would be wrong and the test case would fail.

However, our approach to test conceptual schemas does not guarantee by itself that all the knowledge defined in the conceptual schema is valid according to the executed test set. The reason is that the schema could include knowledge that is not relevant for the formalized scenarios, and this irrelevant knowledge could even be incorrect.

At this point, the following question arises: Which are the basic properties that tests of conceptual schemas should have? The key concept developed in the software testing field for this purpose is that of adequacy criterion (Zhu et al. 1997). A typical example, in program testing, is the criterion that requires that each statement of a program is executed at least once by a test set. Of course, many other criteria are possible. In the context of conceptual schema testing, we can say that an adequacy criterion *C* is a requirement on a test set *TS* of a conceptual schema *CS* such that if *TS* satisfies *C* then *TS* is considered adequate to test *CS* according to *C*.

In the following, we describe a basic set of four adequacy criteria for checking the validity of conceptual schemas by testing. The overall goal of this set is threefold: 1) determining which parts of the conceptual schema have been exercised by a test case, 2) determining which elements of the schema are potentially irrelevant (or even incorrect) and 3) ensuring the satisfiability of the entity types, relationship types, integrity constraints and domain event types, which is a necessary property for correctness.

Each of the proposed criteria is a necessary condition for conceptual schema validity. The criteria are independent each other but taken together they ensure the relevance of the defined knowledge. They also have the unifying (and interesting) characteristic that they ensure the satisfiability of the entity types, relationship types, integrity constraints and domain event types defined in a schema, which is a necessary property for its correctness (Thalheim 2000). This fact lays the ground to define a procedure for checking conceptual schema satisfiability by testing, which is described in Section 11.2.

Moreover, the criteria are independent of the conceptual schema language and of the testing language. We have implemented its automatic analysis in our test processor prototype (Chapter 5).

The proposed adequacy criteria have also been applied to a test set for a fragment of the *osCommerce* conceptual schema, a popular and widely-used e-commerce system. This test set completely satisfies the four criteria. The results of the application in this case study are reported in (Tort 2009a).

It is important to note that our basic set of adequacy criteria may be used in Test-Driven Conceptual Modeling (Chapter 8) in order to check, at the end of each iteration, that all knowledge added to the schema is relevant. The analysis of the proposed criteria is able to detect which schema elements have been added although they are not driven by any test set. In a strict application of TDCM, only the necessary changes to pass the current test case should be done in the schema at each iteration. In this situation, all the proposed properties should be satisfied at the end of all iterations. Even if this principle is not strictly applied, the analysis of elements that cause these properties to fail are useful to reveal untested parts of the schema and suggesting writing and processing new test cases to cover them.

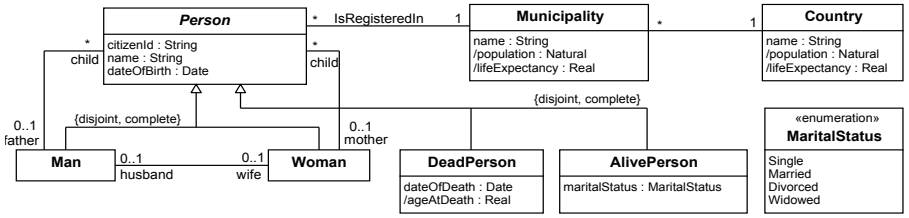
The formalization of the adequacy criteria considers the following notation aspects:

- We denote by *TS* a test set that consists of a set of one or more test cases TC_i.
- The execution of a test case implies the execution of one or more test assertions TA_k.
- *TA* denotes the set of all the test assertions whose verdict is *Pass.*

The conceptual schema fragment of a civil registry domain presented in Fig. 69 is used as a running example, together with the test program shown in Fig. 68. For details about the CSTL language, we refer the reader to Section 4.4.

```
testprogram PeopleRegistration{

   belgiumCreation := new CountryCreation(name:='Belgium');
   assert occurrence belgiumCreation;
   belgium := belgiumCreation.createdCountry;
   brusselsCreation := new MunicipalityCreation(name:='City of Brussels', country:=belgium);
   assert occurrence brusselsCreation;
   brussels := brusselsCreation.createdMunicipality;
   antwerpCreation := new MunicipalityCreation(name:='Antwerp', country:=belgium);
   assert occurrence antwerpCreation;
   antwerp := antwerpCreation.createdMunicipality;

   test familyWithoutChildren{
     audreyBirth := new Birth(citizenId:='AUU', name:='Audrey', sex:=Sex::Woman,
         dateOfBirth:='10-10-1934', municipality:=brussels);
     assert occurrence audreyBirth;
     audrey:= audreyBirth.createdPerson;
     assert true audrey.oclAsType(AlivePerson).maritalStatus = MaritalStatus::Single;

     alexBirth := new Birth(citizenId:='ALL', name:='Alex', sex:=Sex::Man,
                            dateOfBirth:='02-31-1936', municipality:= antwerp);
     assert occurrence alexBirth;
     alex:= alexBirth.createdPerson;
     assert true alex.oclAsType(AlivePerson).maritalStatus = MaritalStatus::Single;
     assert equals brussels.population 1;
     assert equals antwerp.population 1;
     assert equals belgium.population 2;

     m := new Marriage(husband:=alex, wife:=audrey);
     assert occurrence m;
     assert true alex.oclAsType(AlivePerson).maritalStatus = MaritalStatus::Married;
     assert true audrey.oclAsType(AlivePerson).maritalStatus = MaritalStatus::Married;

     alexDeath := new Death(person:=alex, dateOfDeath:='06-11-2003');
     assert occurrence alexDeath;
     assert equals belgium.population 1;
     assert true audrey.oclAsType(AlivePerson).maritalStatus = MaritalStatus::Widowed;
   }

   test familyWithADaughter{
     vincentBirth := new Birth(citizenId:='VVV', name:='Vincent', sex:=Sex::Man,
                               dateOfBirth='01-01-1918', municipality:= brussels);
     assert occurrence vincentBirth;
     vincent:= vincentBirth.createdPerson;

     emmaBirth := new Birth(citizenId:='EEE', name:='Emma', sex:=Sex::Woman,
                            dateOfBirth:='01-01-1922', municipality:= brussels);
     assert occurrence emmaBirth;
     emma:= emmaBirth.createdPerson;

     m := new Marriage(husband:=vincent, wife:=emma);
     assert occurrence m;

     julieBirth := new Birth(citizenId:='JJJ', name:='Julie', sex:=Sex::Woman,
                             dateOfBirth:='01-01-1953',
                             father:=vincent, mother:=emma, municipality:= brussels);
     assert occurrence julieBirth;

     div := new Divorce(husband:=vincent, wife:=emma);
     assert occurrence div;

     vincentDeath := new Death(person:=vincent, dateOfDeath:='01-01-1996');
     assert occurrence vincentDeath;
     emmaDeath := new Death(person:=emma, dateOfDeath:='01-01-2007');
     assert occurrence emmaDeath;
     assert equals brussels.lifeExpectancy 81.5;
     assert equals belgium.lifeExpectancy 81.5;
   }
 }
```

Fig. 68. Test program example



OCL Derivation rules

```
context Municipality::population:Natural
derive: self.person->select(oclIsTypeOf(AlivePerson))->size()

context Municipality::lifeExpectancy:Real
derive: let deadPeople:Set(Person)=self.person->select(oclIsTypeOf(DeadPerson))
        in  if deadPeople->size()>0 then
                deadPeople.oclAsType(DeadPerson).ageAtDeath->sum() / deadPeople->size()
            else 0.0
            endif

context Country::population:Natural
derive: self.municipality.population->sum()

context Country::lifeExpectancy:Real
derive: let deadPeople:Set(Person)=
            self.municipality.person->select(oclIsTypeOf(DeadPerson))->asSet()
        in  if deadPeople->size()>0 then
                deadPeople.oclAsType(DeadPerson).ageAtDeath->sum() / deadPeople->size()
            else 0.0
            endif

context DeadPerson::ageAtDeath:Real
 derive: dateOfDeath-dateOfBirth
```

OCL Integrity constraints

```
context Country inv identifiesInhabitantsByCitizenId:
 self.municipality.person->isUnique(citizenId)
```

Fig. 69. Structural schema fragment of a civil registry system

11.1.1 Base Type Coverage

The base types (entity types, attributes and associations) defined in a conceptual schema are valid if they are relevant and correct (Pohl 2010, Olivé 2007, Lindland et al. 1994). We denote by T_{base} the set of base types. The relevance of each base type $T_i \in T_{base}$ can be ensured by means of testing. The test set *TS* should include at least one test case TC_j such that it:

- builds a state of the IB having at least one instance of T_i, and
- makes an assertion TA_k that can only *Pass* if the above IB state is consistent (that is, it satisfies all constraints).

If the test set includes such test case TC_j, and the execution of TA_k gives the verdict *Pass*, then it is experimentally proved that T_i is relevant according to the expectations formalized as test cases.

This is the rationale for the test adequacy criterion that we call *base type coverage*, which can be formally stated as follows. Let:

$BaseTypes(TA_k) = \{T_i | T_i \in T_{base}$ and there are one or more instances of T_i in at least one of the IB states found consistent during the evaluation of $TA_k\}$

$$BaseTypes(TA) = \bigcup_{TA_k \in TA} BaseTypes(TA_k).$$

We say that a test set *TS* satisfies the *base type coverage* criterion if and only if T_{base} = *BaseTypes*(*TA*). Then, it is experimentally proved that all types $T_i \in T_{base}$ defined in a conceptual schema are relevant. It is important to remark, that the accomplishment of this criterion has the interesting property of ensuring the satisfiability of T_{base} (which is a necessary condition for the correctness of T_{base}). For further details about checking the satisfiability of base types, see Section 11.2.

The analysis of the set of uncovered base types (T_{base} - *BaseTypes*(*TA*)**)** allows us to identify which base types of the schema have not been exercised in any consistent scenario. Either they need more testing in order to satisfy the *base type coverage* criterion or they are irrelevant or incorrect.

In the test program of Fig. 68, the fixture initializes two municipalities (the *City of Brussels* and *Antwerp*) located in a country (*Belgium*). In a test program, the execution of any of its test cases implies the execution of its fixture. Therefore, the entity types *Municipality* and *Country*, its basic attributes and the relationship type between them are covered according to this criterion.

Moreover, the test case *familyWithoutChildren* registers the births of a woman (*audrey)* and a man (*alex*), the marriage between them and the death of *alex.* The execution of this test case implies that the entity types *Man, Woman, AlivePerson*, *DeadPerson* (and *Person* due to the taxonomy), their basic attributes, and the relationship types *IsRegisteredIn* and *husband-wife* are also covered.

However, by taking into account only the test case *familyWithoutChildren*, the coverage analysis identifies that the relationship types *father-child* and *mother-child* are not covered. If we consider that the test set also includes the test case *familyWithADaughter,* then all the basic types of the example become covered.

11.1.2 Derived type coverage

Entity and relationship types may be derived. For each derived type, the conceptual schema includes a derivation rule that defines the population of that type in terms of the population of other types. In UML, derivation rules are formally written in OCL.

Derived types defined in a schema are valid if they are relevant and correct (Olivé 2007, Lindland et al. 1994).

We denote by T_{der} the set of derived types defined in a conceptual schema. The relevance of a derived type can be checked by means of testing. The test set *TS* should include a test case that makes an assertion TA_k whose evaluation requires the derivation of at least one instance of that type.

We denote by *DerTypes*(TA_k) the set of derived entity types such that TA_k has derived one or more instances of them during its evaluation, and by *DerTypes*(*TA*) the set of derived entity types that have instances derived during the evaluation of *TA*.

Formally:

$$DerTypes(TA_k) = \{T_i \mid T_i \in T_{der} \text{ and the evaluation of } TA_k \text{ in a state found consistent has required the derivation of one or more instances of } T_i\}$$

$$DerTypes(TA) = \bigcup_{TA_k \in TA} DerTypes(TA_k)$$

We say that a test set *TS* satisfies the *derived type coverage* criterion if and only if T_{der} = *DerTypes*(*TA*). Then, it is experimentally proved that all types $T_i \in T_{der}$ defined in a conceptual schema are relevant. Moreover, the accomplishment of this criterion has the interesting property of ensuring the satisfiability of T_{der} (which is, in turn, a necessary condition for the correctness of T_{der}). For further details about checking the satisfiability of derived types, see Section 11.2.

The analysis of the set of uncovered derived types (T_{der} - *DerTypes*(*TA*)**)** allows us to identify which derived types have not been exercised in any consistent scenario. Either they need more testing in order to satisfy the *derived type coverage* criterion or they are irrelevant or incorrect.

The first test case (*familyWithoutChildren)* of the test program shown in Fig. 68, makes assertions about the population of the municipalities and the country initialized in the fixture.

```
assert equals brussels.population 1;
assert equals antwerp.population 1;
assert equals belgium.population 2;
```

The verdict of these assertions is *Pass* because the conceptual schema has the knowledge to derive the population as expected. Consequently, this test case

ensures that the derived attributes *Municipality::population* and *Country::population* have been correctly derived in a consistent state.

In contrast, the derived attributes *Municipality::lifeExpectancy*, *Country::lifeExpectancy* and *DeadPerson::ageAtDeath* are not covered if we only consider the test case *familyWithoutChildren*. However, if we add the test case *familyWithADaughter* all derived types become covered.

11.1.3 Valid type configuration coverage

In conceptual models that admit multiple classification (like the UML), an entity may be an instance of two entity types, E_1 and E_2, such that (1) E_1 does not subsume E_2; (2) E_2 does not subsume E_1; and (3) no E_3 is subsumed by both E_1 and E_2. In multiple-classification models, correctness and relevance do not only apply to the individual entity types, but also to the set of valid configurations of entity types (Olivé 2007). These configurations are completely determined by the entity types and the taxonomic constraints of the conceptual schema.

The example of Fig. 69 assumes multiple classification. There are six valid type configurations: {*Person, Man, AlivePerson*}, {*Person, Woman, AlivePerson*}, {*Person, Man, DeadPerson*}, {*Person, Woman, DeadPerson*}, {*Municipality*} and {*Country*}.

The relevance of a valid type configuration $VTC_i = \{E_1, \ldots, E_n\}$ can be checked by means of testing. The test set *TS* should include a test case TC_j such that it (1) builds a state of the IB having at least one entity that is an instance of VTC_i and (2) makes an assertion TA_k that can only *Pass* if the above IB state is consistent.

Therefore, if we want to experimentally prove that all valid type configurations $VTC_i \in VTC$ defined in a conceptual schema are relevant, we must require that for each of them there is at least one test assertion that checks the consistency of one or more IB states having at least one instance of VTC_i.

This is the rationale for the test adequacy criterion that we call *valid type configuration coverage*, which can be formally stated as follows. Let:

$$VTC(TA_k) = \{VTC_i \mid VTC_i \in VTC \text{ and there are one or more instances of } VTC_i \text{ in at least one of the IB states found consistent during the evaluation of } TA_k\}$$

$$VTC(TA) = \bigcup_{TA_k \in TA} VTC(TA_k)$$

We say that a test set *TS* satisfies the *valid type configuration coverage* criterion if and only if *VTC* = *VTC*(*TA*). Then, it is experimentally proved that all types $VTC_i \in VTC$ defined in a conceptual schema are relevant. Additionally, the accomplishment of this criterion has the interesting property of ensuring the satisfiability of *VTC* (which is, in turn, a necessary condition for the correctness of *VTC*) (Olivé 2007). For further details about checking the satisfiability of VTCs, you may read Section 11.2.

When $VTC \neq VTC(TA)$ then there is at least one $VTC_i \in$ VTC but $VTC_i \notin VTC(TA)$. The analysis of the set of uncovered valid type configurations (*VTC* - *VTC(TA)*) allows us to identify which type configurations may not be valid. This means that a valid type configuration VTC_i allowed by the conceptual schema has not been tested. If the domain experts confirm that VTC_i is valid in the domain, then the conceptual modeler must write more test cases. Otherwise, if VTC_i is invalid in the domain, then the conceptual modeler must change the taxonomy to prevent it.

The test case *familyWithoutChildren* proves that all the entity types are covered according to the *Base Type Coverage* (see Section 11.1.1). However, the CSUT example (Fig. 69) assumes multiple classification. Therefore, when analyzing the *Valid Type Configuration Coverage* satisfaction, we realize that {*Person, Woman, DeadPerson*} is not covered (no valid instances of a dead woman participate in the test case). If we also consider the test case *familyWithADaughter*, then all VTCs become covered.

In single-classification schemas, the satisfaction of the *base type coverage* criterion implies the satisfaction of the *valid type configuration coverage* criterion.

11.1.4 Domain event type coverage

Domain event types must be relevant and correct (Olivé 2007). We denote by *Dev* the set of domain event types. The relevance of a domain event type $Dev_i \in Dev$ can be checked by means of testing. The test set *TS* should include a test case TC_j such that it 1) builds a state of the IB, 2) creates an instance *d* of Dev_i, and 3) asserts the occurrence of *d*.

If the test set includes such test case TC_j, and its execution gives the verdict *Pass*, then it is experimentally proved that Dev_i is relevant. If Dev_i is not relevant, then the test set should not include any assertion stating the occurrence of Dev_i.

This is the rationale for the test adequacy criterion that we call *domain event type coverage*, which can be formally stated as follows. Let TA_k be the assertion of a domain event occurrence. We denote by *DevTypes*(TA_k) the type of the domain event whose occurrence is asserted and by *DevTypes*(*TA*) the set of domain event

types that have instances whose occurrence has been asserted during the evaluation of *TA*. Formally:

$$DevTypes(\mathrm{TA}) = \bigcup_{TA_k \in TA} DevTypes(TA_k)$$

We say that a test set *TS* satisfies the *domain event type coverage* criterion if and only if *Dev* = *DevTypes*(*TA*). Then, it is experimentally proved that all types $Dev_i \in Dev$ defined in a conceptual schema are relevant. Again, note that the accomplishment of this criterion has also the interesting property of ensuring the satisfiability of *Dev* (which is, in turn, a necessary condition for the correctness of *Dev*). Satisfiability comprises applicability (the initial IB state has been found consistent and the event constraints have been satisfied) and executability (the new IB state has been found consistent and the event postconditions have been satisfied). For further details about checking the satisfiability of domain event types, you may read Section 11.2.

The set of uncovered event types (*Dev* - *DevTypes*(*TA*)) allows us to identify which event types do not have valid occurrences in any test case. Either they need more testing to satisfy the criterion or they are irrelevant (or even incorrect).

Assume the existence of the ordinary domain events *Birth*, *Death*, *Marriage* and *Divorce*. Events that create countries and municipalities of a country are also considered (*MunicipalityCreation* and *CountryCreation)*. The example test case *familyWithoutChildren* (Fig. 68) exercises the valid execution of these domain events with the exception of the event *Divorce* that becomes covered if we also consider the test case *familyWithADaughter*. The satisfaction of the domain event type coverage criterion ensures that these domain events are satisfiable.

The structural and the behavioral subschema should be consistent (Pilskalns et al. 2007, Salay et al. 2009) between them. If the *domain event type coverage* criterion is satisfied but the *base type coverage* criterion is not satisfied, then either the uncovered base types are not relevant, or some event types are missing in the schema, or the existing ones must be instantiated in other test cases.

11.1.5 Coverage Criteria Satisfaction and Schema Validity

If there exists a test set *TS* that satisfies the four coverage criteria defined in sections 11.1.1, 11.1.2, 11.1.3 and 11.1.4, then we can ensure that all base and derived types, type configurations and domain event types are relevant and satisfiable, which is a necessary condition for correctness.

Formally, if we denote the relevance by *Rel* and the satisfiability by *Sat*, then:

Base type coverage:	$T_{base} = BaseTypes(TA) \rightarrow Rel(T_{base}, TS) \wedge Sat(T_{base})$
Derived type coverage:	$T_{der} = DerTypes(TA) \rightarrow Rel(T_{der}, TS) \wedge Sat(T_{der})$
Valid type configuration coverage:	$VTC = VTC(TA) \rightarrow Rel(VTC, TS) \wedge Sat(VTC)$
Domain event type coverage:	$Dev = DevTypes(TA) \rightarrow Rel(Dev, TS) \wedge Sat(Dev)$

The test program *PeopleRegistration* of Fig. 68 completely satisfies the proposed basic set of test adequacy criteria. Therefore, we can conclude that all the schema elements have been exercised by a test case in at least one consistent scenario, that there are no potentially irrelevant elements, and that these elements are satisfiable.

11.2 Checking Conceptual Schema Satisfiability by Testing

Satisfiability is one of the properties that all correct conceptual schemas must have. Satisfiability applies to both the structural and the behavioral parts of a conceptual schema. Structurally, a conceptual schema is satisfiable if each base or derived entity and relationship type of the schema may have a non-empty population at a certain time. An entity or relationship type is unsatisfiable when the set of constraints defined in the schema can only be satisfied if the population of that type is empty. Behaviorally, a conceptual schema is satisfiable if each event type is satisfiable, that is, there is at least one consistent state of the Information Base (IB) and one event of that type with a set of characteristics such that the event constraints are satisfied, and the effects of the event leave the IB in a state that is consistent and satisfies the event postconditions. A state of the IB is consistent if it satisfies all integrity constraints.

As we have seen in Chapter 3, there has been a lot of work on automated reasoning procedures for checking satisfiability, mainly for the structural part of a schema (a representative set of recent papers is (Queralt et al. 2009, Queralt et al. 2008, Formica 2003, Berardi et al. 2005, Jarrar 2007, Brambilla et al. 2009, Gogolla et al. 2009, Clavel et al. 2009)). However, it is well known that the problem of reasoning in conceptual schemas including general integrity constraints, derivation rules and event pre and postconditions is undecidable. Therefore, the available automated reasoning procedures are restricted to certain kinds of constraints, derivation rules, pre/postconditions or domains, or they may not terminate in some circumstances.

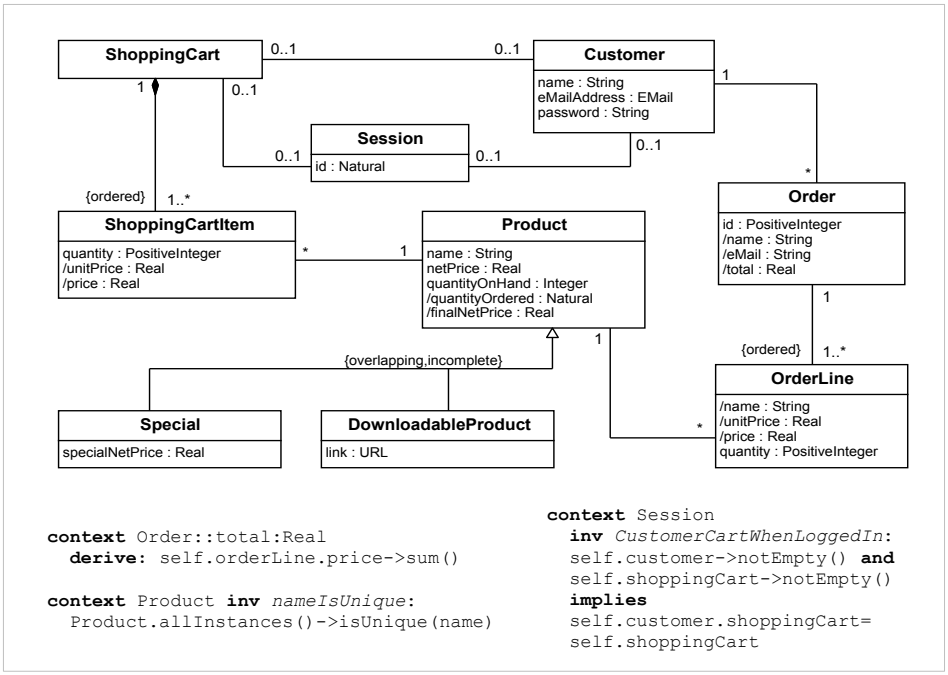


Fig. 70. Fragment of the osCommerce structural schema

In this section, we explore an alternative approach to satisfiability checking (Olivé et al. 2010), which can be used when conceptual schemas are developed in the context of a development environment that allows their testing.

When conceptual schemas can be tested, then their satisfiability can be proved by testing. The idea is that the conceptual modeler sets up a test case such that if its verdict is *Pass* then by definition the entity or relationship or event type under test is satisfiable. If the conceptual modeler is unable to set up such test case, then this is not a formal proof of unsatisfiability. We show that the unsatisfiability results obtained by testing are not as strong as those obtained by automated reasoning procedures when they are applicable, but in many practical cases testing provides a clue that helps to uncover a faulty schema.

In the following, we analyze first the satisfiability of base entity and relationship types (Section 11.2.1), then that of derived base and relationships types (Section 11.2.2), and finally that of domain event types (Section 11.2.3). All the examples of this chapter are taken from a fragment of the conceptual schema of the osCommerce system (Tort 2007), a popular industrial e-commerce system. The full details of the case study can be found in the report (Tort 2009a).

11.2.1 Base type satisfiability

Satisfiability (or liveliness) is a well-known property of base entity and relationship types. A base type is satisfiable (or lively) if it may have a non-empty finite population at certain time. In a conceptual schema, a base type is unsatisfiable when the set of constraints defined in that schema can only be satisfied if the population of that type is empty or infinite (Queralt et al. 2006). In conceptual modeling, it is usually required that all base types be satisfiable (Parsons et al. 1997, Calvanese et al. 1994, Costal et al. 1996).

Let T_i be a base type (entity types, attributes and associations) defined in a conceptual schema. The satisfiability of T_i can be checked by means of testing. The idea is to set up a test case TC_j such that it:

- builds a state of the IB having at least one instance of T_i, and
- makes an assertion TA_k that can only *Pass* if the above IB state is consistent (that is, it satisfies all constraints).

If the execution of TA_k gives the verdict *Pass*, then it is experimentally proved that T_i is satisfiable. Note that in a single test case we can instantiate several types and that a single assertion can experimentally prove that all of them are satisfiable.

Assume the conceptual schema fragment shown in Fig. 70. Moreover, Fig. 71 shows a test program aimed to test the conceptual schema. In this test program example, the fixture creates the customer *john* and the session *s*. It also initializes the online catalog with the product *shirt* and the special product *trousers*. The execution of any of the test cases of the test program example implies the execution of this fixture and ensures that the entity types *Customer*, *Session*, *Product* and *Special* (and also their attributes) are satisfiable.

Moreover, the test case *confirmOrder* adds a shopping cart item with two units of *shirt* and another item with a pair of *trousers*. The shopping cart is created when adding the first item. By this way, the entity types *ShoppingCart* and *ShoppingCartItem* (and also their relationship types, including attributes) are proved satisfiable. The relationship types *ShoppingCart-Session, Session-Customer* and *Customer-ShoppingCart* are also satisfiable when the *LogIn* event occurs (the session is assigned to a customer and the anonymous shopping cart becomes the shopping cart of the customer of the session). The entity types *Order* and *OrderLine* (and their relationship types) become satisfiable when the event *OrderConfirmation* occurs (the order and its order lines are created from the shopping cart). Finally, the occurrence of the instance *ndp* of the domain event type *NewDownloadableProduct* proves the satisfiability of the entity type *DownloadableProduct*.

```
testprogram PlaceOrder{

nc := new NewCustomer
      (name:='John', eMailAddress:='john@john.com', password:='pwd');
assert occurrence nc;
john := nc.createdCustomer;

ns := new NewSession;
assert occurrence ns;
s := ns.createdSession;

np1 := new NewProduct(name:='shirt', netPrice:=20, quantityOnHand:=5);
assert occurrence np1;
shirt := np1.createdProduct;
np2 := new NewSpecial (name:='trousers', netPrice:=80,
                       quantityOnHand:=25,specialNetPrice:=65);
assert occurrence np2;
trousers := np2.createdProduct;

test confirmOrder{
    apsc1 := new AddProductToShoppingCart(quantity:=2,
                                          session:=s, product:=shirt);
    assert occurrence apsc1;
    apsc2 := new AddProductToShoppingCart(quantity:=1,
                                          session:=s, product:=trousers);
    assert occurrence apsc2;
    assert equals s.shoppingCart.shoppingCartItem->at(1).price 40;
    assert equals s.shoppingCart.shoppingCartItem->at(2).price 65;

    li := new LogIn(customer:=john, session:=s);
    assert occurrence li;
    oc := new OrderConfirmation(shoppingCart:= s.customer.shoppingCart);
    assert occurrence oc;
    assert equals oc.createdOrder.total 105;
    assert equals shirt.quantityOrdered 2;
    assert equals oc.createdOrder.eMail 'john@john.com';
    assert equals oc.createdOrder.name 'John';
    assert equals oc.createdOrder.orderLine->at(1).name 'shirt';
    assert equals oc.createdOrder.orderLine->at(2).name 'trousers';
}

test productKindsInCatalog{
    ndp := new NewDownloadableProduct
           (name:='fashionDesigner', netPrice:=43, quantityOnHand:=85,
            link:='http://fashionshop.com/fashionDesigner.zip');
    assert occurrence ndp;
    nds := new NewDownloadableSpecial
           (name:='FashionTipsMagazine', netPrice:=3, quantityOnHand:=15,
            specialNetPrice := 2,
            link:='http://fashionshop.com/tips.pdf');
    assert occurrence nds;
}

}
```

Fig. 71. Test program example

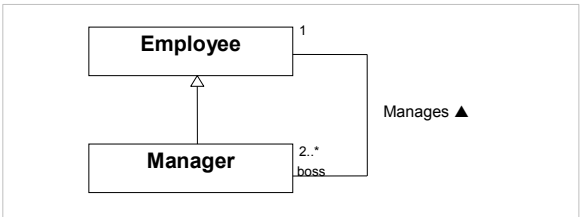


Fig. 72. Schema fragment with types that cannot be satisfied

If a conceptual schema includes a base type T_i that is unsatisfiable, then the conceptual modeler will be unable to set up a test case that builds a state of the IB with at least one instance of T_i, and an assertion that can only *Pass* if that state is consistent. This is not a formal proof that T_i is unsatisfiable, but in many practical cases it provides a clue that helps to uncover a faulty constraint.

For example, consider the schema example shown in Fig. 72 (adapted from (Calvanese et al. 1994)). The association *Manages* is satisfiable if we do not take into account that *Manager IsA Employee*. However, if we take this inclusion constraint into account then it cannot be satisfied. If the conceptual modeler writes a test case such as

```
test  EmployeeWithTwoBosses{
   emily := new Employee;
   john := new Manager(employee:=Emily);
   natalie := new Manager(employee:=Emily);
   assert consistency;
}
```

the assertion will *Fail* because *john* and *natalie* do not have (at least) two bosses. Any change of the instances of the three types will produce the same result, and the conceptual modeler will find out soon that the defined cardinality constraints are wrong.

11.2.2 Derived type satisfiability

Entity and relationship types may be derived. For each derived type, the conceptual schema includes a derivation rule that defines the population of that type in terms of the population of other types. In UML, the derivation rules are written in OCL. Derived types must be satisfiable too (Costal et al. 1996). Satisfiability of a derived type means that its derivation rule may derive at least one instance of it at a certain time.

The satisfiability of a derived type can be checked by means of testing. The idea is to write a test case that makes an assertion TA_k whose evaluation requires the derivation of at least one instance of that type.

In the example of Fig. 70 there are ten derived attributes. The assertions *"assert equals s.shoppingCart.shoppingCartItem->at(1).price 40"* and *"assert equals s.shoppingCart.shoppingCartItem->at(2).price 65"* (specified in the test case *confirmOrder* shown in Fig. 68) imply that the attribute *ShoppingCartItem::price* is satisfiable and also the attributes *ShoppingCartItem::unitPrice* and *Product::finalNetPrice*. This is because the derivation of the *price* of a shopping cart item implies the derivation of its *unitPrice* (its derivation rule expression is *unitPrice*quantity)*, and the *unitPrice* of a shopping cart item corresponds to the *finalNetPrice* of its associated product. Similarly, the assertion "*assert equals oc.createdOrder.total 105"* implies the satisfiability of the attributes *Order::total* (its derivation rule is shown in Fig. 70), *OrderLine::price* and *OrderLine::unitPrice*. Finally, the assertions *"assert equals shirt.quantityOrdered 2"*, *"assert equals oc.createdOrder.eMail 'john@john.com'"*, "*assert equals oc.createdOrder.name 'John'*", "*assert equals oc.createdOrder.orderLine->at(1).name 'shirt'*" and "*assert equals oc.createdOrder.orderLine->at(2).name 'trousers'*" make the attributes *Product::quantityOrdered*, *Order::name*, *Order::eMail* and *OrderLine::name* satisfiable.

11.2.3 Domain event type satisfiability

Domain event types must be satisfiable too. Domain event type satisfiability comprises the properties of applicability and executability defined in (Queralt et al. 2009, Costal et al. 1996): A domain event type Dev_i is applicable if there is a consistent IB state and one instance *d* of Dev_i with a set of characteristics such that the event constraints are satisfied, and Dev_i is executable if Dev_i is applicable and the effects of *d* leave the IB in a state that is consistent and satisfies the event postconditions.

The satisfiability of a domain event type Dev_i can be checked by means of testing. The idea is to set up a test case TC_j such that it:

- builds a state of the IB, and
- creates an instance *d* of Dev_i, and
- asserts the occurrence of *d*.

If the test set includes such test case TC_j, and its execution gives the verdict *Pass*, then it is experimentally proved that Dev_i is satisfiable: applicable (because the initial IB state has been found consistent and the event constraints have been satisfied) and executable (because the new IB state has been found consistent and the event postconditions have been satisfied).

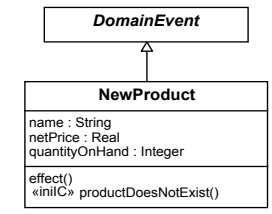
Assume that we consider that we have specified the following events: *NewCustomer*, *NewSession*, *NewProduct*, *NewDownloadableProduct*, *LogIn*, *OrderConfirmation AddProductToShoppingCart* and *NewDownloadableSpecial*. Fig. 73 shows the complete specification of one of these domain events (*New Product*). The test program of Fig. 71 exercises the valid execution of all the domain events considered in the example and this ensures that these domain events are satisfiable.

If a conceptual schema includes a domain event type Dev_i that is unsatisfiable, then the conceptual modeler will be unable to set ups a test case that builds a state of the IB, creates an instance of Dev_i and asserts its occurrence. Again, this is not a formal proof that Dev_i is unsatisfiable, but in many practical cases it provides a clue that helps to uncover a faulty constraint.

For example, related to the schema of Fig. 70, assume that there is a domain event type *RemoveOrder*, whose intended effect is to remove the order to which it is associated. If one of the constraints of the event is:

```
context RemoveOrder::thereAreNoOrderLines():Boolean
  body: self.order.orderLine->isEmpty()
```

then *RemoveOrder* is not applicable, because an instance of *Order* is always associated with at least one instance of *OrderLine*. Any assertion of the occurrence of an instance of *RemoveOrder* will *Fail*, and the conceptual modeler will find out that either the above event constraint or the cardinality constraint of Fig. 70 is incorrect.




```
context NewProduct::
productDoesNotExist():Boolean
  body: not Product.allInstances()
    -> exists (pr|pr.name = self.name)
```

```
context  NewProduct::effect()
   post:(Product.allInstances -
   Product.allInstances@pre) -> one(p:Product|
       p.oclIsNew()  and
       p.name = self.name and
       p.netPrice = self.netPrice and
       p.quantityOnHand = self.quantityOnHand)
```

```
method NewProduct::effect(){
p:=new Product;
p.name:=self.name;
p.netPrice:=self.netPrice;
p.quantityOnHand:=self.quantityOnHand;
self.createdProduct:=p;
}
```

Fig. 73. Domain event specification example

12

Conclusions and Further Work

Conceptual modeling is an essential activity in requirements engineering, which is aimed at eliciting, specifying and validating the conceptual schema of an information system (Chapter 1).

The main objective of conceptual schema validation is checking the alignment between the knowledge specified in the schema and the stakeholders' expectations. The validation of conceptual schemas is an important and relevant challenge in information systems development (Chapter 2). Conceptual schema validation enhances the semantic quality of the schema.

This thesis is a contribution to the challenge of conceptual schema validation and it addresses the following main research questions: (1) Can we test conceptual schemas in order to enhance its semantic quality?, and (2) can we develop conceptual schemas using a test-driven conceptual modeling method?

Conceptual Schema Testing

The first main research question has been explored by the development of answers to the following specific questions: (1.1) What it means to test conceptual schemas?, (1.2) Which kinds of tests are required to test conceptual schemas, (1.3.) Why do we want to test conceptual schemas?, (1.4) Which are the requirements of an environment for conceptual schema testing?, and (1.5) How can we determine the suitability of a test for testing a conceptual schema?

In order to address these questions, we have shown that executable conceptual schemas may be tested as a means for their validation. We have described the fundamentals of conceptual schema testing based on a list of five kinds of tests that can be applied to conceptual schemas. We have explained the similarities and differences between testing conceptual schemas and testing programs. Furthermore, we have presented a conceptual schema testing approach that allows conceptual schemas of information systems to be tested with the goal of uncover requirements errors (Chapter 4). We have seen that the semantic quality of conceptual schemas (Chapter 2) may be improved by testing, and that other quality factors such as pragmatic, social or organizational quality are also positively influenced. Moreover, we have explained that our approach may cooperate with existing conceptual schema validation and verification techniques (Chapter 3).

We have proposed the Conceptual Schema Testing Language (CSTL), a textual procedural language for writing tests of executable conceptual schemas written in UML/OCL (Chapter 4). As far as we know, this is the first proposal of a language for testing conceptual schemas designed in the style of the modern *xUnit* software testing frameworks.

We have implemented a supporting tool for automated testing of conceptual schemas (Chapter 5). This tool contains a test processor that manages and executes CSTL programs. It includes a test interpreter that coordinates the execution of the tests and invokes the services of an information processor. Tests written in CSTL may be automatically executed as many times as needed (regression testing). We have also shown that our test processor has been extended in order to be able to deal with predefined temporal constraints and derivation rules.

In the context of Design Science Research, which is the adopted framework in this Thesis, we have evaluated the feasibility of the approach by means of its application in two real-sized case studies. The objective of these case studies has been the application of our approach in order to test the conceptual schemas of two widely-used e-commerce systems (Chapter 6).

We have also seen that a test set whose verdict is *Pass* is not sufficient to ensure the necessary conditions for the validity of the conceptual schema under test. We have defined a set of four basic adequacy criteria for ascertaining conceptual schema validity (Chapter 11). The overall goals of these criteria is determining which parts of the schema have been exercised by a test case; analyzing which elements of the schema are potentially irrelevant (or even incorrect); and ensuring the satisfiability of the entity types, relationship types, integrity constraints and domain event types, which is a necessary property for correctness.

Test-Driven Conceptual Modeling

The second main research question has been explored by the development of answers to the following questions: (2.1) Which are the activities and the process that define TDCM?, (2.2) How TDCM may contribute to the quality of conceptual schemas?, (2.3) Can TDCM be integrated in existing requirements engineering and software development methods?, and (2.4) Which are the requirements of an environment to support TDCM?

In order to answer these specific research questions, in this Thesis we have presented the Test-Driven Conceptual Modeling (TDCM) method (Chapter 8), which is based on the principles of Test-Driven Development (Chapter 7) applied to conceptual modeling, and we have shown that it is possible and useful to develop a conceptual schema using it.

Using TDCM, conceptual modelers have at any time fully tested schemas. TDCM fosters the development of correct and complete schemas, which are fundamental quality properties in conceptual modeling (Chapter 2). We have seen that, using TDCM, a system's conceptual schema is incrementally obtained by performing three kinds of tasks: (1) Write a test the system should pass; (2) Change the schema to pass the test; and (3) Refactor the schema to improve its qualities.

We have also presented a set of guidelines on the use of TDCM. We have dealt with schemas written in UML/OCL, but TDCM can be adapted for the development of schemas in other languages.

We have explored the integration of TDCM (Chapter 10) into four well-known requirements engineering methods: the Unified Process development methodology, the MDD-based approaches, the storytest-driven agile methods and the goal and scenario-oriented methods.

Finally, we have reported four complementary experimental case studies on the application of TDCM (Chapter 9). The successful application of TDCM assisted by the use of our proposed testing environment (Chapter 5) concludes the feasibility of the method in practice. These experimental case studies have been useful to categorize the kinds of errors and failures that drive TDCM iterations. Furthermore, lessons about the resultant conceptual schema, the testing effort, the iterations productivity, the confidence on quality, and common iteration patterns have been learned and discussed, as a knowledge base for future TDCM applications.

Given that TDCM was developed in the context of Design Science Research (Section 1.4), the reported experiences were useful to improve-by-use the supporting tool.

A Work that Opens New Research Directions

As far as we know, the work presented in this Thesis is an innovative proposal for making feasible the automated testing of conceptual schemas. Based on the contributions described in this document, we believe that our work opens new directions for research and development in conceptual modeling.

Firstly, in this Thesis we have proposed a testing language and a test processor for testing conceptual schemas, and we have analyzed its feasibility by means of two e-commerce case studies. However, we need to study how to use both the language and the processor in professional projects in order to know how and when to get the maximum benefit from them.

Secondly, when the transformation from a conceptual schema to a Platform-Specific Model (PSM) is manual, there is the need of rewriting the conceptual schema test cases at the PSM level. In this context, it would be very interesting to save testing effort by developing automatic transformation procedures of the test cases.

Thirdly, taking into account that we have presented a basic set of adequacy criteria for testing conceptual schemas that specify both the structural and the behavioral knowledge, it is possible to define a variant of these criteria which is applicable when only the structural subschema is available. The idea in this context is that the test cases do not build the IB states by means of the occurrence of domain events, but by means of explicit insertion, deletion and update (CSTL) statements. This variant could be useful in projects that aim at developing only the structural schema, or in the initial phases of the development of a complete schema.

Fourthly, we remark that test sets should satisfy the proposed criteria given that they ensure the necessary conditions for conceptual schema validity. However, several additional criteria may be envisaged in order to enhance the confidence about the conceptual schema correctness and according to the testing strategy. As further work, we mention two of them here. The first may be inspired on the branch coverage criterion in program testing (Zhu et al. 1997), aiming at ensuring that all branches of the OCL integrity constraints have been tested. The second is a criterion that ensures that all integrity constraints that must be enforced by the system have at least one domain event precondition that prevents the occurrence of a domain event that could lead to its violation.

Fifthly, we think that conceptual schema testing should be integrated with other validation techniques, and the test processor should be integrated with the other tools of a comprehensive development environment (Bouzeghoub et al. 2000).

Sixthly, as far as we know, our work on Test-Driven Conceptual Modeling (TDCM) is the first work that explores the use of TDD in conceptual modeling, and naturally work remains to be done. We mention some research directions related to TDCM here:

- The first one is to further elaborate the integration approach of TDCM into the above mentioned requirements engineering methods, and analyze the results of its use in practice.

- The second direction aims at experimentally determining to what extent TDCM brings the advantages and drawbacks of TDD to conceptual modeling. This may depend on the requirements engineering method in which TDCM is applied. We have conjectured that TDCM may "inherit" several advantages and drawbacks and we have analyzed its feasibility and application patterns in four case studies (Chapter 9), but they must be confirmed by rigorous development experiments in business contexts. We are confident that TDCM will be very useful in those projects that follow the OMG's Model-Driven Development approach when the transformation from Platform-Independent Models to Platform-Specific Models is fully automatic and correct-by-construction, and we conjecture that TDCM will be useful in the other contexts too, even in projects that only develop the structural schema.

- Finally, the third direction aims at enlarging the set of guidelines on the use of TDCM, so that they provide useful advice to the conceptual modeler in at least the most common situations.

References

AMBLER, S. W. Agile Modeling. Wiley, 2002.

ARMSTRONG, W. W. Dependency Structures of Data Base Relationships. In: Congress of the International Federation of Information Processing (IFIP 1974). New York: Elsevier, 1974, pp. 580–583.

Association for Information Systems (AIS). Design Research in Information Systems. 2009. Available from <http://desrist.org/design-research-in-information-systems/>.

ASTELS, D. Test Driven Development: A Practical Guide. Prentice Hall, 2003.

Atlassian Pty. Clover: Java Code Coverage & Test Optimization. Available from <http://www.atlassian.com/software/clover/>.

BAKER, P., et al. Model-Driven Testing: Using the UML Testing Profile. Springer, 2008.

BECK, K. Test-Driven Development: By Example. Addison-Wesley, 2003.

BECK, K. Simple Smalltalk Testing: With Patterns. Smalltalk, 1994. Available from <http://www.xprogramming.com/ftp/TestingFramework/Testfram.rtf>.

BECK, K., et al. Manifesto for Agile Software Development. 2001. Available from <http://www.agilemanifesto.org/>.

BEIZER, B. Software Testing Techniques. Dreamtech Press, 2002.

BERARDI, D.; CALVANESE, D. and DE GIACOMO, G. Reasoning on UML Class Diagrams. Artificial Intelligence, 2005, vol. 168, no. 1-2, pp. 70-118.

BEYNON-DAVIES, P.; TUDHOPE, D. and MACKAY, H. Information Systems Prototyping in Practice. Journal of Information Technology, 1999, vol. 14, no. 1, pp. 107-120.

BHAT, T. and NAGAPPAN, N. Evaluating the Efficacy of Test-Driven Development: Industrial Case Studies. In: 2006 ACM/IEEE International Symposium on Empirical Software Engineering. ACM Press New York, NY, USA, 2006, pp. 356-363.

BICARREGUI, J., et al. Making the most of Formal Specification through Animation, Testing and Proof. Science of Computer Programming, 1997, vol. 29, no. 1-2, pp. 53-78.

BOEHM, B. Get Ready for Agile Methods, with Care. Computer, 2002, vol. 35, no. 1, pp. 64-69.

BOMAN, M., et al. Conceptual Modelling. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1997.

BOOCH, G.; RUMBAUGH, J. and JACOBSON, I. The Unified Modeling Language Reference Manual. Second ed. Addison-Wesley, 2005.

BOUZEGHOUB, M.; KEDAD, Z. and MÉTAIS, E. CASE Tools: Computer Support for Conceptual Modelling. In: DIAZ, O.; and PIATTINI, M. eds., Advanced Database Systems, Techniques and Design. Artech House, 2000, pp. 439-483.

BRAMBILLA, M. and TZIVISKOU, C. An Online Platform for Semantic Validation of UML Models. In: 9th International Conference on Web Engineering (ICWE 2009). Springer, 2009, pp. 477-480.

BRIAND, L. and LABICHE, Y. A UML-Based Approach to System Testing. In: 4th International Conference on the Unified Modeling Language (UML 2001). Springer, 2001, pp. 149-208.

BRINKKEMPER, S. Method Engineering: Engineering of Information Systems Development Methods and Tools. Information and Software Technology, 1996, vol. 38, no. 4, pp. 275-280.

British Computer Society. Vocabulary of Terms in Software Testing, British Standards, BS 7925-1. 2009. Available from <http://www.testingstandards.co.uk/bs_7925-1_online.htm>.

BURTON-JONES, A., et al. A Semiotic Metrics Suite for Assessing the Quality of Ontologies. Data & Knowledge Engineering, 2005, vol. 55, no. 1, pp. 84-102.

CABOT, J.; CLARISÓ, R. and RIERA, D. Verification of UML/OCL Class Diagrams using Constraint Programming. In: MoDeVVa 2008. ICST Workshop. Citeseer, 2008, pp. 73–80.

CABOT, J.; PAU, R. and RAVENTÓS, R. From UML/OCL to SBVR Specifications: A Challenging Transformation. Information Systems, 2010, vol. 35, no. 4, pp. 417-440.

CALVANESE, D. and LENZERINI, M. On the Interaction between ISA and Cardinality Constraints. In: 10th International Conference on Data Engineering (ICDE 1994). IEEE Computer Society Press, 1994, pp. 204-213.

CANFORA, G., et al. Evaluating Advantages of Test Driven Development: A Controlled Experiment with Professionals. In: 2006 ACM/IEEE International symposium on empirical software engineering. New York: ACM Press, 2006, pp. 364-371.

CHIKOFSKY, E. J. and CROSS II, J. H. Reverse Engineering and Design Recovery: A Taxonomy. IEEE Software, 1990, pp. 13-17.

CLAVEL, M.; EGEA, M. and DE DIOS, M. A. G. Checking Unsatisfiability for OCL Constraints. In: OCL Workshop MODELS 2009. Electronic Communications of the EASST, 2009. Available from <http://modeling-languages.com/events/OCLWorkshop2009/papers/3.pdf>.

COCKBURN, A. Writing Effective use Cases. Addison-Wesley, 1999.

CONESA, J.; OLIVE, A. and CABALLÉ, S. Refactoring and its Application to Ontologies. In: Semantic Web Personalization and Context Awareness. Information Science Reference, 2011, pp. 107-136.

CORREA, A. and WERNER, C. Applying Refactoring Techniques to uml/ocl Models. In: 7th International Conference on the Unified Modeling Language (UML 2004). Springer, 2004, pp. 173-187.

COSTAL, D., et al. Handling Conceptual Model Validation by Planning. In: 5th International Conference on Advanced Information Systems Engineering (CAiSE 1996). Springer, 1996, pp. 255-271.

DALIANIS, H. A Method for Validating a Conceptual Model by Natural Language Discourse Generation. In: 4th International Conference on Advanced Information Systems Engineering (CAiSE 1992). Springer, 1992, pp. 425-444.

DENG, C.; WILSON, P. and MAURER, F. Fitclipse: A Fit-Based Eclipse Plug-in for Executable Acceptance Test Driven Development. In: 8th International Conference on Agile Processes in Software Engineering and Extreme Programming (XP 2007). Springer, 2007, pp. 93-100.

DIGNUM, F., et al. Constraint Modelling using a Conceptual Prototyping Language. Data & Knowledge Engineering, 1987, vol. 2, no. 3, pp. 213-254.

DOKE, E. R. and SWANSON, N. E. Decision Variables for Selecting Prototyping in Information Systems Development: A Delphi Study of MIS Managers. Information & Management, 1995, vol. 29, no. 4, pp. 173-182.

DOOLAN, E. P. Experience with Fagan's Inspection Method. Software - Practice and Experience, 1992, vol. 22, no. 2, pp. 173-182.

Eclipse Foundation. Eclipse IDE. Available from <http://www.eclipse.org>.

EDWARDS, S. Using Test-Driven Development in the Classroom: Providing Students with Automatic, Concrete Feedback on Performance. In: International Conference on Education and Information Systems: Technologies and Applications (EISTA). Citeseer, 2003.

ENDRES, A. and ROMBACH, H. D. A Handbook of Software and Systems Engineering: Empirical Observations, Laws and Theories. Addison-Wesley, 2003.

ERICKSON, J.; LYYTINEN, K. and SIAU, K. Agile Modeling, Agile Software Development, and Extreme Programming: The State of Research. Journal of Database Management, 2005, vol. 16, no. 4, pp. 88-100.

FAGAN, M. E. Advances in Software Inspections. IEEE Transactions on Software Engineering, 1986, vol. 12, no. 7, pp. 744-751.

FAGAN, M. E. Design and Code Inspections to Reduce Errors in Program Development. IBM Systems Journal, 1976, vol. 15, no. 3, pp. 182-211.

FORMICA, A. Satisfiability of Object-Oriented Database Constraints with Set and Bag Attributes. Information Systems, 2003, vol. 28, no. 3, pp. 213-224.

FOWLER, M. and BECK, K. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, 1999.

FREDERIKS, P. J. M. and VAN DER WEIDE, T. P. Information Modeling: The Process and the Required Competencies of its Participants. Data & Knowledge Engineering, 2006, vol. 58, no. 1, pp. 4-20.

GAMMA, E. and BECK, K. JUnit: A cook's Tour. Java Report, 1999, vol. 4, no. 5, pp. 27-38. Available from <http://junit.sourceforge.net/doc/cookbook/cookbook.htm>.

GARGANTINI, A. and HEITMEYER, C. Using Model Checking to Generate Tests from Requirements Specifications. ACM SIGSOFT Software Engineering Notes, 1999, vol. 24, no. 6, pp. 146-162.

GEMINO, A. and WAND, Y. Complexity and Clarity in Conceptual Modeling: Comparison of Mandatory and Optional Properties. Data & Knowledge Engineering, 2005, vol. 55, no. 3, pp. 301-326.

GENERO, M.; POELS, G. and PIATTINI, M. Defining and Validating Metrics for Assessing the Understandability of entity–relationship Diagrams. Data & Knowledge Engineering, 2008, vol. 64, no. 3, pp. 534-557.

GEORGE, B. and WILLIAMS, L. An Initial Investigation of Test Driven Development in Industry. In: 2003 ACM symposium on applied computing. ACM Press New York, NY, USA, 2003, pp. 1135-1139.

GERAS, A.; SMITH, M. and MILLER, J. A Prototype Empirical Evaluation of Test Driven Development. In: 10th International Symposium on Software Metrics (METRICS'04). IEEE CS Press, 2004, pp. 405-416.

GLASS, R. L. An Ancient (but Still Valid?) Look at the Classification of Testing. IEEE Software, 2008, vol. 25, no. 6, pp. 111-112.

GLINZ, M. A Lightweight Approach to Consistency of Scenarios and Class Models. In: 4th International Conference on Requirements Engineering (ICRE 2000). IEEE, 2000, pp. 49-58.

GLOVER, A. Jump into JUnit 4. 2007. Available from <http://www.ibm.com/developerworks/edu/j-dw-java-junit4.html>.

GOGOLLA, M.; BOHLING, J. and RICHTERS, M. Validating UML and OCL Models in USE by Automatic Snapshot Generation. Software & Systems Modeling, 2005, vol. 4, no. 4, pp. 386-398.

GOGOLLA, M.; BÜTTNER, F. and RICHTERS, M. USE: A UML-Based Specification Environment for Validating UML and OCL. Science of Computer Programming, 2007, vol. 69, no. 1-3, pp. 27-34.

GOGOLLA, M., et al. A Development Environment for an Object Specification Language. IEEE Transactions on Knowledge and Data Engineering, 1995, vol. 7, no. 3, pp. 505-508.

GOGOLLA, M.; KUHLMANN, M. and HAMANN, L. Consistency, Independence and Consequences in UML and OCL Models. In: 2nd International Conference on Tests And Proofs (TAP 2009). Springer, 2009, pp. 90-104.

GOODENOUGH, J. B. and GERHART, S. L. Toward a Theory of Test Data Selection. In: International Conference on Reliable Software. ACM, 1975, pp. 493-510.

GRÜNINGER, M. and FOX, M. S. Methodology for the Design and Evaluation of Ontologies. In: Workshop on Basic Ontological Issues in Knowledge Sharing. Citeseer, 1995.

GULLA, J. A. A General Explanation Component for Conceptual Modeling in CASE Environments. ACM Transactions on Information Systems, 1996, vol. 14, no. 3, pp. 297-329.

GULLA, J. A. and WILLUMSEN, G. Using Explanations to Improve the Validation of Executable Models. In: 5th International Conference on Advanced Information Systems Engineering (CAiSE 1993). Springer, 1993, pp. 118-142.

HALPIN, T. A. Information Modeling and Relational Databases. Morgan Kaufmann, 2001.

HALPIN, T. A.; MORGAN, A. J. and MORGAN, T. Information Modeling and Relational Databases. Morgan Kaufmann, 2008.

HAMILL, P. Unit Test Frameworks. O'Reilly, 2004.

HARTMAN, A. and NAGIN, K. The AGEDIS Tools for Model Based Testing. ACM SIGSOFT Software Engineering Notes, 2004, vol. 29, no. 4, pp. 129-132.

HARTMANN, S.; LINK, S. and TRINH, T. Constraint Acquisition for Entity-Relationship Models. Data & Knowledge Engineering, 2009, vol. 68, no. 10, pp. 1128-1155.

HEVNER, A. R., et al. Design Science in Information Systems Research. Mis Quarterly, 2004, pp. 75-105.

HIERONS, R. M., et al. Using Formal Specifications to Support Testing. ACM Computing Surveys, 2009, vol. 41, no. 2.

IEEE. IEEE Guide for Developing System Requirements Specifications, IEEE Std 1233. 1998a.

IEEE. Standard for Software Verification and Validation, Std 1012-1998, 1998b.

IEEE. Standard Glossary of Software Engineering Terminology, Std.610.12, 1990.

IEEE. IEEE Guide to Software Requirements Specification, Std 830. 1984.

INSFRÁN, E.; PELECHANO, V. and PASTOR, O. Conceptual Modeling in the eXtreme. Information and Software Technology, 2002, vol. 44, no. 11, pp. 659-669.

International Standards Organization (ISO). ISO 9000-2000: Quality Management Systems: Fundamentals and Vocabulary. 2000.

International Standards Organization (ISO). ISO TC97/SCS/WG3: Concepts and Terminology for the Conceptual Schema and the Information Base. 1982.

JAFFE, M. S., et al. Software Requirements Analysis for Real-Time Process-Control Systems. IEEE Transactions on Software Engineering, 1991, vol. 17, no. 3, pp. 241-258.

JANZEN, D. and SAIEDIAN, H. Does Test-Driven Development really Improve Software Design Quality?. IEEE Software, 2008, vol. 25, no. 2, pp. 77-84.

JANZEN, D. and SAIEDIAN, H. Test-Driven Development: Concepts, Taxonomy, and Future Direction. Computer, 2005, vol. 38, no. 9, pp. 43-50.

JARRAR, M. Towards Automated Reasoning on ORM Schemes. In: 26th International Conference on Conceptual Modeling (ER 2007). Springer, 2007, pp. 181-197.

JAVED, A. Z.; STROOPER, P. A. and WATSON, G. N. Automated Generation of Test Cases using Model-Driven Architecture. In: 2nd International Workshop on Automation of Software Test (AST 2007). Washington, DC, USA: IEEE Computer Society, 2007.

JESUS, L. and CARAPUCA, R. Automatic Generation of Documentation for Information Systems. In: 4th International Conference on Advanced Information Systems Engineering (CAiSE 1992). Springer, 1992, pp. 48–64.

JUDSON, S. R.; CARVER, D. L. and FRANCE, R. B. A Metamodeling Approach to Model Transformation. In: Conference on Object Oriented Programming Systems Languages and Applications. ACM New York, NY, USA, 2003, pp. 326-327.

KALYANPUR, A., et al. Debugging Unsatisfiable Classes in OWL Ontologies. Web Semantics: Science, Services and Agents on the World Wide Web, 2005, vol. 3, no. 4, pp. 268-293.

KAVAKLI, E. and LOUCOPOULOS, P. Goal Modeling in Requirements Engineering: Analysis and Critique of Current Methods. Information Modeling Methods and Methodologies, 2005, vol. 3273, pp. 173-187.

KOSKELA, L. Test Driven: Practical TDD and Acceptance TDD for Java Developers. Manning Publications, 2007.

KROGSTIE, J.; SINDRE, G. and JØRGENSEN, H. Process Models Representing Knowledge for Action: A Revised Quality Framework. European Journal of Information Systems, 2006, vol. 15, no. 1, pp. 91-102.

KÜHNE, T. What is a Model? In: BEZIVIN, J.; and HECKEL, R. eds., Dagstuhl Seminar in Language Engineering for Model-Driven Software Development. Dagstuhl: Schloss Dagstuhl, 2005.

LARMAN, C. Applying UML and Patterns. Third ed. Prentice Hall, 2005.

LARMAN, C. and BASILI, V. R. Iterative and Incremental Developments. A Brief History. Computer, 2003, vol. 36, no. 6, pp. 47-56.

LEITE, J. C. S. P., et al. Scenario Inspections. Requirements Engineering, 2005, vol. 10, no. 1, pp. 1-21.

LEITE, J. C. S. P., et al. Enhancing a Requirements Baseline with Scenarios. Requirements Engineering, 1997, vol. 2, no. 4, pp. 184-198.

LINDLAND, O. I. and KROGSTIE, J. Validating Conceptual Models by Transformational Prototyping. In: 5th International Conference on Advanced Information Systems Engineering (CAiSE 1993). Springer, 1993, pp. 165-183.

LINDLAND, O. I.; SINDRE, G. and SOLVBERG, A. Understanding Quality in Conceptual Modeling. IEEE Software, 1994, vol. 11, no. 2, pp. 42-49.

LINGS, B. and LUNDELL, B. On Transferring a Method into a Usage Situation. In: KAPLAN, B., et al. eds., Information Systems Research: Relevant Theory and Informed Practice. Boston: Springer, 2004, pp. 535-553.

LOY, M. and ECKSTEIN, R. Java Swing. O'Reilly Media, Inc., 2002.

LUI, K. M. and CHAN, K. C. Test Driven Development and Software Process Improvement in China. In: 5th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP 2005). Springer, 2005, pp. 219–222.

LUTZ, R. R. Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems. In: IEEE International Symposium on Requirements Engineering. Citeseer, 1993, pp. 35–46.

MADEYSKI, L. and SZAŁA, Ł. The Impact of Test-Driven Development on Software Development Productivity-an Empirical Study. In: ABRAHAMSSON, P., et al. eds., Software Process Improvement. Springer, 2007, pp. 200–211.

MARTIN, R. C.; MELNIK, G. and INC, O. M. Tests and Requirements, Requirements and Tests: A Möbius Strip. IEEE Software, 2008, vol. 25, no. 1, pp. 54-59.

MAXIMILIEN, E. M. and WILLIAMS, L. Assessing Test-Driven Development at IBM. In: 25th International Conference on Software Engineering (ICSE'03). Washington DC, USA: IEEE Computer Society, 2003, pp. 564-569.

MELLOR, S. J. and BALCER, M. J. Executable UML. A Foundation for Model-Driven Architecture. Addison-Wesley, 2002.

MELNIK, G. and MAURER, F. Multiple Perspectives on Executable Acceptance Test-Driven Development. In: International Conference on Agile Processes in Software Engineering and Extreme Programming (XP 2007). Springer, 2007, pp. 245-249.

MELNIK, G.; MAURER, F. and CHIASSON, M. Executable Acceptance Tests for Communicating Business Requirements: Customer Perspective. In: Agile 2006 Conference. IEEE Computer Press, 2006, pp. 35-46.

MENS, T. and TOURWÉ, T. A Survey of Software Refactoring. IEEE Transactions on Software Engineering, 2004, vol. 30, no. 2, pp. 126-139.

MÉTAIS, E. Enhancing Information Systems Management with Natural Language Processing Techniques. Data & Knowledge Engineering, 2002, vol. 41, no. 2-3, pp. 247-272.

MEYER, B. Seven Principles of Software Testing. IEEE Computer, 2008, vol. 41, no. 8, pp. 99-101.

MIYASHITA, K.; MASUDA, K. and HIGASHITANI, F. Coordinated Service Allocation through Flexible Reservation. IEEE Transactions on Services Computing, 2008, vol. 1, no. 2, pp. 117-128.

MOODY, D. and SHANKS, G. What Makes a Good Data Model? Evaluating the Quality of Entity Relationship Models. In: LOUCOPOULOS, P. ed., 13th International Conference on the Entity Relationship Approach (ER '94). Springer, 1994, pp. 94-111.

MOODY, D. L. Theoretical and Practical Issues in Evaluating the Quality of Conceptual Models: Current State and Future Directions. Data & Knowledge Engineering, 2005, vol. 55, no. 3, pp. 243-276.

MOODY, D. L., et al. Evaluating the Quality of Information Models: Empirical Testing of a Conceptual Model Quality Framework. In: 25th International Conference on Software Engineering (ICSE 2003). Washington, DC, USA: IEEE Computer Society, 2003, pp. 295-305.

MOORE, I. Jester-a JUnit Test Tester. In: 2nd XP Universe and First Agile Universe Conference (XP 2001). Citeseer, 2001, pp. 84–87.

MUGRIDGE, R. Managing Agile Project Requirements with Storytest-Driven Development. IEEE Software, 2008, vol. 25, no. 1, pp. 68-75.

MUGRIDGE, R. and CUNNINGHAM, W. Fit for Developing Software: Framework for Integrated Tests. Upper Saddle River, NJ, USA: Prentice Hall, 2005.

MULLER, M. and HAGNER, O. Experiment about Test-First Programming. IEE Proceedings Software, 2002, vol. 149, no. 5, pp. 131-136.

MURPHY-HILL, E. and BLACK, A. P. Refactoring Tools: Fitness for Purpose. IEEE Software, 2008, vol. 25, no. 5, pp. 38-44.

MYERS, G. J., et al. The Art of Software Testing. Wiley, 2004.

NEBUT, C., et al. Automatic Test Generation: A use Case Driven Approach. IEEE Transactions on Software Engineering, 2006, vol. 32, no. 3, pp. 140-155.

NELSON, H. J., et al. A Conceptual Modeling Quality Framework. Software Quality Journal, 2011, pp. 1-28.

NIETO, P.; COSTAL, D. and GÓMEZ, C. Enhancing the Semantics of UML Association Redefinition. Data & Knowledge Engineering, 2010, vol. 70, no. 2, pp. 182-207.

NoUnit team. NOUnit Project. Available from <http://nounit.sourceforge.net/>.

NUSEIBEH, B. and EASTERBROOK, S. The Process of Inconsistency Management: A Framework for Understanding. In: First International Workshop on the Requirements Engineering Process (REP'99). IEEE, 1999, pp. 364-368.

Object Management Group (OMG). Action Language for Foundational UML (Alf). FTF-Beta 1, ptc/2010-10-05. 2010a. Available from <http://www.omg.org/spec/ALF/1.0/Beta1/>.

Object Management Group (OMG). Object Constraint Language Specification. Version 2.2., formal/2010-02-01. 2010b. Available from <http://www.omg.org/spec/OCL/2.2/>.

Object Management Group (OMG). UML Superstructure Version 2.2, formal/2009-02-02. 2009. Available from <http://www.omg.org/spec/OCL/2.2/>.

Object Management Group (OMG). UML Testing Profile. 2005. Available from <http://www.omg.org/spec/UTP/1.0/>.

Object Management Group (OMG). MDA Guide. 2003. Available from <www.omg.org/docs/omg/03-06-01.pdf>.

OLIVÉ, A. Conceptual Modeling of Information Systems. Berlin: Springer, 2007.

OLIVÉ, A. A Method for the Definition of Integrity Constraints in Object-Oriented Conceptual Modeling Languages. Data & Knowledge Engineering, 2006, vol. 59, no. 3, pp. 559-575.

OLIVÉ, A. Conceptual Schema-Centric Development: A Grand Challenge for Information Systems Research. In: 17th Conference on Advanced Information Systems Engineering (CAiSE'05). Springer, 2005, pp. 1-15.

OLIVÉ, A. Derivation Rules in Object-Oriented Conceptual Modeling Languages. In: Proceedings of CAiSE 2003. Berlin: Springer, 2003, pp. 404-420.

OLIVÉ, A. and RAVENTÓS, R. Modeling Events as Entities in Object-Oriented Conceptual Modeling Languages. Data & Knowledge Engineering, 2006, vol. 58, no. 3, pp. 243-262.

OLIVÉ, A. and SANCHO, M. R. Validating Conceptual Specifications through Model Execution. Information Systems, 1996, vol. 21, no. 2, pp. 167-186.

OLIVÉ, A. and TORT, A. Testing Conceptual Schema Satisfiability. In: NURCAN, S., et al. eds., Intentional Perspectives on Information Systems Engineering. Springer, 2010.

OMG. Object Constraint Language. 2006. Available from <http://www.omg.org/spec/OCL/2.0/>.

osTicket. OsTicket Website. Available from <http://http://www.osticket.com/>.

OSTROFF, J. S., et al. E-Tester: A Contract-Aware and Agent-Based Unit Testing Framework for Eiffel. Journal of Object Technology, 2005, vol. 4, no. 7, pp. 97-114.

OSTROFF, J. S. and TORSHIZI, F. A. Testable Requirements and Specifications. In: 3rd International Conference on Tests And Proofs (TAP 2007). Springer, 2007, pp. 17.

PANCUR, M., et al. Towards Empirical Evaluation of Test-Driven Development in a University Environment. In: International Conference on Computer as a Tool (EUROCON 2003). IEEE, 2003.

PARR, T. The Definitive ANTLR Reference: Building Domain-Specific Languages. Pragmatic Bookshelf, 2007.

PARSONS, J. and WAND, Y. Choosing Classes in Conceptual Modeling. Communications of the ACM, 1997, vol. 40, no. 6, pp. 63-69.

PASTOR, O. and MOLINA, J. C. Model-Driven Architecture in Practice. Springer, 2007.

PEFFERS, K., et al. A Design Science Research Methodology for Information Systems Research. Journal of Management Information Systems, 2007, vol. 24, no. 3, pp. 45-77.

PICKIN, S., et al. System Test Synthesis from UML Models of Distributed Software. In: 22nd International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2002). Springer, 2002, pp. 97-113.

PILSKALNS, O., et al. Testing UML Designs. Information and Software Technology, 2007, vol. 49, no. 8, pp. 892-912.

POHL, K. Requirements Engineering. Fundamentals, Principles, and Techniques. Berlin: Springer, 2010.

PORRES, I. Model Refactorings as Rule-Based Update Transformations. In: 6th International Conference on the Unified Modeling Language (UML 2003). Springer, 2003, pp. 159-174.

PORTER, A. A.; VOTTA JR, L. G. and BASILI, V. R. Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment. IEEE Transactions on Software Engineering, 1995, vol. 21, no. 6, pp. 563-575.

POSNER, R. Charles Morris and the Behavioral Foundations of Semiotics. New York and London: Plenum Press, 1987.

QUERALT, A. Validation of UML Conceptual Schemas with OCL Constraints and Operations. Universitat Politècnica de Catalunya, 2009.

QUERALT, A. and TENIENTE, E. Reasoning on UML Conceptual Schemas with Operations. In: The 21st International Conference on Advanced Information Systems Engineering (CAiSE'09). Springer, 2009, pp. 47-62.

QUERALT, A. and TENIENTE, E. Decidable Reasoning in UML Schemas with Constraints. In: The 20th International Conference on Advanced Information Systems Engineering (CAiSE'08). Springer, 2008, pp. 281-295.

QUERALT, A. and TENIENTE, E. Reasoning on UML Class Diagrams with OCL Constraints. In: 26th International Conference on Conceptual Modeling (ER 2006). Springer, 2006, pp. 497-512.

RAMIREZ, A. Esquema Conceptual De Magento, Un Sistema De Comerç Electrònic. UPC, 2011. Available from <http://hdl.handle.net/2099.1/12294>.

REGNELL, B.; KIMBLER, K. and WESSLEN, A. Improving the use Case Driven Approach to Requirements Engineering. In: Second International Symposium on Requirements Engineering. Citeseer, 1995, pp. 40-47.

REGNELL, B.; RUNESON, P. and THELIN, T. Are the Perspectives really Different?–further Experimentation on Scenario-Based Reading of Requirements. Empirical Software Engineering, 2000, vol. 5, no. 4, pp. 331-356.

RICHTERS, M. and GOGOLLA, M. Validating UML Models and OCL Constraints. In: 3rd International Conference on the Unified Modeling Language (UML 2000). York, UK: Springer, 2000, pp. 265-277.

ROBERTSON, J. and ROBERTSON, S. Volere: Requirements Specification Template. Atlantic Systems Guild, 1997.

ROBINSON, W. N.; PAWLOWSKI, S. D. and VOLKOV, V. Requirements Interaction Management. ACM Computing Surveys (CSUR), 2003, vol. 35, no. 2, pp. 132-190.

ROLLAND, C. and PROIX, C. A Natural Language Approach for Requirements Engineering. In: The 5th International Conference on Advanced Information Systems Engineering (CAiSE'93). Springer, 1992, pp. 257-277.

ROLLAND, C. and SALINESI, C. Modeling Goals and Reasoning with them. In: AURUM, A.; and WOHLIN, C. eds., Engineering and Managing Software Requirements. Springer, 2005, pp. 189-217.

RTI International. The Economic Impacts of Inadequate Infrastructure for Software Testing. North Carolina, USA: National Institute of Standards and Technology & US Department of Commerce, 2002.

SALAY, R. and MYLOPOULOS, J. Improving Model Quality using Diagram Coverage Criteria. In: 21st International Conference on Advanced Information Systems Engineering (CAiSE 2009). Springer, 2009, pp. 186-200.

SANGWAN, R. S. and LAPLANTE, P. A. Test-Driven Development in Large Projects. IT Professional, 2006, vol. 8, no. 5, pp. 25-29.

SANTOS NETO, P.; RESENDE, R. and PADUA, C. A Method for Information Systems Testing Automation. In: The 17th International Conference on Advanced Information Systems Engineering (CAiSE'05). Springer, 2005, pp. 504-518.

SAUVÉ, J. P.; NETO, A. and LOPES, O. Teaching Software Development with ATDD and EasyAccept. ACM SIGCSE Bulletin, 2008, vol. 40, no. 1, pp. 542-546.

SAUVÉ, J. P., et al. EasyAccept: A Tool to Easily Create, Run and Drive Development with Automated Acceptance Tests. In: International workshop on Automation of software test. ACM, 2006, pp. 117.

SCHEWE, K. D. and THALHEIM, B. Conceptual Modelling of Web Information Systems. Data & Knowledge Engineering, 2005, vol. 54, no. 2, pp. 147-188.

SEYBOLD, C.; MEIER, S. and GLINZ, M. Scenario-Driven Modeling and Validation of Requirements Models. In: International Workshop on Scenarios and State Machines: models, algorithms, and tools. ACM, 2006.

SOMMERVILLE, I. Software Engineering. 9th ed. Addison-Wesley, 2010.

SOMMERVILLE, I. and KOTONYA, G. Requirements Engineering: Processes and Techniques. John Wiley & Sons, Inc. New York, NY, USA, 1998.

STEPHENS, M. and ROSENBERG, D. Design Driven Testing. New York: Apress, 2010.

STEPHENS, M. and ROSENBERG, D. Extreme Programming Refactored: The Case Against XP. New York: Apress, 2003.

SUNYÉ, G.; LE GUENNEC, A. and JÉZÉQUEL, J. M. Using UML Action Semantics for Model Execution and Transformation. Information Systems, 2002, vol. 27, no. 6, pp. 445-457.

SUNYÉ, G., et al. Refactoring UML Models. In: 4th International Conference on the Unified Modeling Language (UML 2001). Springer, 2001, pp. 134-148.

SUTCLIFFE, A. Scenario-Based Requirements Analysis. Requirements Engineering, 1998, vol. 3, no. 1, pp. 48-65.

TER HOFSTEDE, A. H. M.; PROPER, H. A. and VAN DER WEIDE, T. P. Exploiting Fact Verbalisation in Conceptual Information Modelling. Information Systems, 1997, vol. 22, no. 6-7, pp. 349-385.

THALHEIM, B. Entity-Relationship Modeling: Foundations of Database Technology. Springer, 2000.

TORT, A. Development of the Conceptual Schema of a Bowling Game System by Applying TDCM. UPC, 2011a. Available from <http://hdl.handle.net/2117/11196>.

TORT, A. Development of the Conceptual Schema of the osTicket System by Applying TDCM. UPC, 2011b. Available from <http://hdl.handle.net/2117/12369>.

TORT, A. A Basic Set of Test Cases for a Fragment of the osCommerce Conceptual Schema. UPC, 2009a. Available from <http://hdl.handle.net/2117/6130>.

TORT, A. Testing the osCommerce Conceptual Schema by using CSTL. UPC, 2009b. Available from <http://hdl.handle.net/2117/6289>.

TORT, A. The osCommerce Conceptual Schema. 2007. Available from <http://guifre.lsi.upc.edu/OSCommerce.pdf>.

TORT, A. The CSTL Processor Project Website. Available from <http://www.essi.upc.edu/~atort/cstlprocessor>.

TORT, A. and OLIVÉ, A. An Approach to Testing Conceptual Schemas. Data & Knowledge Engineering, 2010, vol. 69, no. 6, pp. 598-618.

TORT, A.; OLIVÉ, A. and SANCHO, M. R. An Approach to Test-Driven Development of Conceptual Schemas. Data & Knowledge Engineering, 2011a, vol. 69, no. 6, pp. 598-618.

TORT, A.; SANCHO, M. R. and OLIVÉ, A. The CSTL Processor: A Tool for Automated Conceptual Schema Testing. In: DE TROYER, O., et al eds., Advances in Conceptual Modeling. Recent Developments and New Directions. ER 2011 Workshops FP-UML, MoRE-BI, Onto-CoM, SeCoGIS, Variability@ER, WISM. Springer, 2011b.

TRONG, T. D., et al. UMLAnT: An Eclipse Plugin for Animating and Testing UML Designs. In: 2005 OOPSLA workshop on Eclipse technology eXchange. ACM Press New York, NY, USA, 2005, pp. 120-124.

UUSITALO, E. J., et al. Linking Requirements and Testing in Practice. In: 16th IEEE International Requirements Engineering (RE'08). Springer, 2008, pp. 265-270.

VAN EMDEN, E. and MOONEN, L. Java Quality Assurance by Detecting Code Smells. In: 9th Working Conference on Reverse Engineering. Citeseer, 2002.

VAN GORP, P., et al. Towards Automating Source-Consistent UML Refactorings. UML 2003, 2003, vol. 2863, pp. 144-158.

VAN LAMSWEERDE, A. Requirements Engineering: From System Goals to UML Models to Software Specifications. Wiley, 2009.

VLIET, H. Software Engineering: Principles and Practice. 2nd ed. New York, USA: John Wiley & Sons, 2000.

VONK, R. Prototyping. Prentice-Hall, 1990.

WAND, Y. and WANG, R. Y. Anchoring Data Quality Dimensions in Ontological Foundations. Communications of the ACM, 1996, vol. 39, no. 11, pp. 86-95.

WHALEN, M. W., et al. Coverage Metrics for Requirements-Based Testing. In: 2006 International Symposium on Software Testing and Analysis. ACM, 2006, pp. 36.

WOHLIN, C. Experimentation in Software Engineering: An Introduction. Springer Netherlands, 2000.

YANG, Q.; LI, J. J. and WEISS, D. M. A Survey of Coverage-Based Testing Tools. In: International Workshop on Automation of Software Test (AST 2006). New York: ACM, 2007.

ZAVE, P. Classification of Research Efforts in Requirements Engineering. ACM Computing Surveys, 1997, vol. 29, no. 4, pp. 321.

ZHANG, J.; LIN, Y. and GRAY, J. Generic and Domain-Specific Model Refactoring using a Model Transformation Engine. In: BEYDEDA, S.; BOOK, M.and GRUHN, V. eds., Model-Driven Software Development. Springer, 2005, pp. 199–218.

ZHANG, Y. Test-Driven Modeling for Model-Driven Development. IEEE Software, 2004, vol. 21, no. 5, pp. 80-86.

ZHU, H.; HALL, P. A. V. and MAY, J. H. R. Software Unit Test Coverage and Adequacy. ACM Computing Surveys, 1997, vol. 29, no. 4, pp. 366-427.

Index

I

K

M

O

P

Q

R

S

T

U

V

W

X

Appendix A

In this appendix, we provide the whole grammar of the CSTL language described in Section 4.4.

testProgram :
testprogram <programID> { *fixture fixtureComponent* testCase** }

fixture :
*statement**

fixtureComponent :
fixturecomponent <fixtureComponentID> { *statement** }

testCase :
concreteTest
| *abstractTest*
| *abstractTestInvocation*

concreteTest :
test <testID> { *statement** }

abstractTest :
abstract test <abstractTestID> *paramList* { *statement** }

paramList :
(*parameter* [**,** *parameter*]* **)**

parameter :
parameterType **<parameterID>**

type :
<oclPrimitiveType>
| **<entityTypeID>**

parameterType :
type
| Fixture

abstractTestInvocation :
test <abstractTestID> *parametersAssignment*

parametersAssignment :
(*parameterAssignment* [**,** *parameterAssignment*]* **)**

parameterAssignment :
  <parameterName> := *expression*

expression :
  <oclExpressionWithVariableIDs>

statement :
  stateStatement **;**
    | *variableStatement* **;**
    | *assertion* **;**
    | *controlFlowStatement*

stateStatement :
  entityCreation
    | *entityDeletion*
    | *binaryPropertySetting*
    | *nAryRelationshipCreation*
    | *fixtureComponentLoading*

variableStatement :
  variableDeclaration
    | *variableAssignment*

assertion :
  assertTrue
    | *assertFalse*
    | *assertEquals*
    | *assertNotEquals*
    | *assertConsistency*
    | *assertInconsistency*
    | *assertDomainEventOccurrence*
    | *assertDomainEventNonOccurrence*

controlFlowStatement :
  conditional
    | *whileLoop*
    | *forLoop*
    | *forEachLoop*

entityCreation :
  new <entityTypeID> [**,** **<entityTypeID>**]* *propertiesAssignment*?

propertiesAssignment :
  (*propertyAssignment* [**,** *propertyAssignment*]* **)**

propertyAssignment :
  <propertyID> := *expression*

entityDeletion :
  delete *expression*

binaryPropertySetting :
  expression **:=** *expression*

nAryRelationshipCreation :
  new <assocID> *participantsAssignment*

participantsAssignment :
(*participantAssignment* [**,** *participantAssignment*]+ **)**

prarticipantAssignment :
<roleID> := *expression*

fixtureComponentLoading :
load <fixturecomponentID>

variableDeclaration :
type **<varID>**

variableAssignment :
[**<varID>** | *varDeclaration*] **:=** [*expression* | *entityCreation* | *nAryRelationshipCreation*]

assertTrue :
assert true *expression*

assertFalse :
assert false *expression*

assertEquals :
assert equals *expression expression*

assertNotEquals :
assert not equals *expression expression*

assertConsistency :
assert consistency

assertInconsistency :
assert inconsistency

assertDomainEventOccurrence :
assert occurrence <domainEventID>

assertDomainEventNonOccurrence :
assert non-occurrence <domainEventID>

assertDomainEventNonOccurrence :
assert non-occurrence <domainEventID>

condition :
if *expression* **then** *statement**
[**else if** *expression* **then** *statement**]*
[**else** *statement**]?
endif

whileLoop :
while *expression* **do** *statement** **endwhile**

forLoop :
for *variableAssignment* **to** *expression* **step** *expression* **do** *statement** **endfor**

forEachLoop :

for each [*variableDeclaration* | *varID*] **in** *expression* **do** *statement** **endfor**