

Testing-Based Conceptual Schema Validation in a Model-Driven Environment

PhD Thesis

Maria Fernanda Granda



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Advisors:

Dra. Nelly Condori-Fernández

Dra. Tanja E. J. Vos

Prof. Dr. Óscar Pastor López

September 2017

Testing-Based Conceptual Schema Validation in a Model-Driven Environment

María Fernanda Granda Juca



A thesis submitted in partial fulfilment of the requirements for
the Ph.D. degree by the Universitat Politècnica de València

Advisors:

Dra. Nelly Condori-Fernández

Dra. Tanja E. J. Vos

Prof. Dr. Óscar Pastor López

September 2017

This report was prepared by:

María Fernanda Granda
fgranda@pros.upv.es
PROS Research Centre
Univèrsitat Politècnica de València
Camino de Vera s/n, Edificio 1F, DISC
46022, Valencia, Spain

Advisors

- Dra. Nelly Condori Fernández, Universidade da Coruña, Spain and Vrije Universiteit Amsterdam, The Netherlands
- Dra. Tanja E. J. Vos, Universitat Politècnica de València, Spain
- Prof. Dr. Óscar Pastor, Universitat Politècnica de València, Spain

External reviewers of the thesis

- Prof. Javier Dolado Cosin, University of the Basque Country, Spain
- Prof. Haralambos Mouratidis, University of Brighton, United Kingdom
- Prof. Jolita Ralyté, University of Geneva, Switzerland

Members of the Thesis committee

- *President:* Prof. Roel Wieringa, University of Twente, The Netherlands
- *Secretary:* Prof. Jolita Ralyté, University of Geneva, Switzerland
- *Speaker:* Prof. Javier Tuya González, University of Oviedo, Spain

*To God and my family,
For their love, care and support in all my experiencies.*

ACKNOWLEDGEMENTS

I am grateful for nights that became mornings, friends who became family and dreams that became true. *Anonymous.*

This thesis is the result of several years of hard work, during which I have received the professional and personal support of many people. My thanks to all of them.

First of all, I would like to thank my thesis supervisors, Nelly Condori Fernández, Tanja Vos and Oscar Pastor, for giving me the opportunity to work with and learn from them. To Nelly, my diary supervisor, thanks for the knowledge imparted on Requirements and Empirical Evaluation, for the time dedicated to discussing my doubts and advances, and demanding me to my utmost. To Tanja, for her guidance and support in the area of Testing, for her enthusiasm at all times. To Oscar, for believing in me from the beginning and pointing me in the right direction in the world of Model-driven Development, for his friendship and constant interest. To all three for their guidance and support that have allowed me to get here.

Many thanks to all the people who have collaborated directly with the development of this thesis. Special thanks to Sergio España and Marcela Ruiz for allowing me to use their Communication Analysis method and tool. To the Spanish everis company for allowing me to evaluate and validate my tool using one of their cases. To Ivette Vilar and Tania González for their time and valuable comments. To Ignacio Panach for translating the Abstract into the Valencian language.

To all the colleagues of the PROS research group specially José Reyes, Carlos Iñiguez, Francisco Valverde, Urko Rueda, Didier Bueno, Raúl Soriano and Ana Ciudad. Thank you for all the good moments.

I'm also going to thank all my friends who helped me disconnect from work and inspired me with the motivation to move on: Sonia

León, Cristina Castillo, Gloria Mora, Sonia Cárdenas, Mauricio Loachamin, Julio Sandobalín, Alejandro Catalá, Miguel Zúñiga, Priscila Andrade, Priscila Cedillo, Freddy Serrano and Angel Cuenca.

Special thanks to Patricia Lago and Nelly Condori-Fernández for allowing me to take part in their Software and Services group (S2) and for the useful discussions we had during my stay in the Faculty of Sciences at the Vrije Universiteit in Amsterdam. Also many thanks to the rest of the people in the group particularly to Mojca, Maryam, Damian and Giuseppe.

I also want to thank the reviewers of this thesis for agreeing to be a part of the court and the external reviewers who will read a preliminary version of the thesis and suggest improvements.

I want to thank my family for their love, prayers and constant support, my mother, sisters, mother-in-law, father-in-law, sisters-in-law, brothers-in-law and my nieces and nephews. Finally, and very important to me, many thanks to my husband Otto and our three children Maria Paula, Luis Felipe and Maria Emilia for their patience and understanding in hard times, for supporting me and encouraging me in all the adventures undertaken, but especially for their unconditional love and company.

This work has been supported by the Universidad de Cuenca, and SENESCYT (Secretaría Nacional de Educación Superior, Ciencia y Tecnología) of the Republic of Ecuador.

ABSTRACT

Despite much scepticism and problems for its adoption, the Model-Driven Development (MDD) is being used and improved to provide many inherent benefits for industry. One of its greatest benefits is the ability to handle the complexity of software development by raising the abstraction level. Models are expressed using concepts that are not related to a specific implementation technology (e.g. Unified Modelling Language -UML, Object Constraint Language –OCL, Action Language for Foundational UML -ALF), which means that the models can be easier to specify, maintain and document. As in Model-Driven Engineering (MDE), the primary artefacts are the conceptual models, efforts are focused on their creation, testing and evolution at different levels of abstraction through transformations because if a conceptual schema has defects, these are passed on to the following stages, including coding. Thus, one of the challenges for researchers and developers in Model-Driven Development is being able to identify defects early on, at the conceptual schema level, as this helps reduce development costs and improve software quality.

Over the last decade, little research work has been performed in this area. Some of the causes of this are the high theoretical complexity of testing conceptual schemas and the lack of adequate software support. This research area thus admits new methods and techniques, facing challenges such as generation of test cases using information external to the conceptual schemas (i.e. requirements), the measurement of possible automation, selection and prioritization of test cases, the need for an efficient support tool using standard semantics, the opportune feedback to support the software quality assurance process and facilitate making decisions based on the analysis and interpretation of the results.

The aim of this thesis is to mitigate some of the problems that affect conceptual schema validation by providing a novel testing-based validation framework based on Model-Driven Development. The use of MDD improves abstraction, automation and reuse, which allows us to alleviate the complexity of our validation framework. Furthermore, by leveraging MDD techniques (such as metamodeling, model transformations, and models at runtime), our framework supports four phases of the testing process: test design, test case generation, test case execution and the evaluation of the results, unlike traditional testing approaches, which, in general, only support some of these phases.

In order to provide software support for our proposal, we developed the CoSTest ALF-based testing environment. To ensure that CoSTest offers the necessary functionality, we first identified a set of functional requirements. Then, after these requirements were identified, we defined the architecture and testing environment of the validation framework, and finally we implemented the architecture in the Eclipse context. CoSTest has been developed to test several properties on the executable model, such as syntactic correctness (i.e. all the elements in the model conform to the syntax of the language in which it is described), consistency between the structural and behavioural parts (its integrity constraints) and completeness (i.e. all possible changes on the system state can be performed through the execution of the operations defined in the executable model). For defective models, the CoSTest report returns a meaningful feedback that helps locate and repair any defects detected.

The work involved in the thesis was validated by means of six studies using cases found in the literature, as well as in a practical industrial case. The first four studies were laboratory experiments to validate and evaluate some CoSTest components such as mode-driven generation of test cases, the mutant generator used to prioritize and select test cases, as well as the generator of an ALF-based executable conceptual schema. In the fifth study, the mutation analysis was

applied to evaluate the effectiveness and adequacy of CoSTest' test cases when detecting different defects in mutated CSs. In the last study, CoSTest was assessed by means of the Technology Acceptance Model (TAM) and the interview method. While the TAM allowed us to subjectively measure usefulness and ease-of-use, the interview method allowed us to identify its limitations and consider possible improvements to be implemented in the tool. Overall, the results were favourable. CoSTest was highly rated in perceived usefulness and ease-of-use and also obtained positive results in the effectiveness of test cases.

RESUM

A pesar de l'escepticisme i les dificultats en la seua adopció, el Desenvolupament Orientat per Models (MDD, segons les sigles en anglès) està sent usat i millorat per tal de proveir molts beneficis potencials inherents a l' indústria. Un dels majors beneficis és la capacitat de manejar la complexitat del desenvolupament del programari elevat el nivell d'abstracció. Els models s'expressen mitjançant conceptes que no estan relacionats amb una tecnologia d'implementació específica (per exemple, el Llenguatge de Modelat Unificat – UML, Llenguatge de Restricció d'Objectes –OCL, Llenguatge d'Acció per al Foundational UML – ALF), el que significa que els models poder ser més fàcils d'especificar, mantindre i documentar. A causa de que en una Enginyeria dirigida per models (MDE), els artefactes primaris són els models conceptuals, els esforços es centren en la seua creació, prova i evolució a diferents nivells d'abstracció mitjançant transformacions, perquè si un esquema conceptual té defectes, aquestos es passen a les següents etapes, inclosa la codificació. Per tant, un del reptes per als investigadors i desenvolupadors en MDD és poder identificar els defectes des del principi, a nivell de esquemes conceptuals, perquè açò ajudaria a reduir els costos de desenvolupament i millora de la qualitat del programari.

Durant l'última dècada, pocs treballs d'investigació s'han fet en aquesta àrea. Algunes de les causes d'aquesta realitat són l'alta complexitat teòrica de provar esquemes conceptuals i la falta de suport de programari adequat. Per tant, aquesta àrea d'investigació admet nous mètodes i tècniques, enfrontant reptes com la generació de casos de prova mitjançant informació externa als esquemes conceptuals (es a dir, requisits), la medició de una possible automatització, selecció i prioritització de casos de prova, la necessitat de una ferramenta de suport rentable que utilitze una semàntica estàndard, la retroalimentació oportuna per suportar el procés d'assegurament de la

qualitat del programari i la facilitat per a prendre decisions basades en l'anàlisi i la interpretació dels resultats.

En aquesta tesi intentem mitigar alguns dels problemes que afecten a la validació dels esquemes conceptuals, proporcionant un nou marc de validació basat en proves que va ser construït mitjançant un desenvolupament dirigit per models. L'ús de MDD permet un augment en l'abstracció, automatització i reutilització que ens permet alleujar la complexitat del nostre marc de validació. A més a més, al aprofitar les tècniques MDD (com el metamodelat, les transformacions de models i els models en temps d'execució), el nostre marc suporta quatre fases del procés de prova: disseny, generació i execució de casos de prova, així com l'avaluació de resultats del procés de prova. Açò és diferent als enfocaments de proves tradicionals, que en general només admeten algunes d'estes fases.

Amb la finalitat de proporcionar suport de programari per a la nostra proposta, hem desenvolupat un entorn de proves basat en el llenguatge ALF que s'anomena CoSTest. Per tal d'assegurar que CoSTest ofereix la funcionalitat necessària, identifiquem un conjunt de requisits funcionals abans de desenvolupar la ferramenta. Després d'identificar aquests requisits, definim l'arquitectura i l'ambient de proves del nostre marc de validació, i finalment, implementem l'arquitectura en el context Eclipse. CoSTest ha sigut desenvolupat per provar diverses propietats sobre el model executable com la correcció sintàctica (és a dir, tots els elements del model s'ajusten a la sintaxi del llenguatge en el que es descriu), consistència entre la part estructural i el comportament (les seues restriccions d'integritat) i completitud (és a dir, tots els canvis possibles en l'estat del sistema es poden realitzar mitjançant l'execució de les operacions definides en el model executable). Per als models defectuosos, l'informe de CoSTest retorna una retroalimentació significativa que ajuda a localitzar i reparar els defectes detectats.

El treball de tesi va ser avaluat mitjançant sis estudis usant casos trobats a la literatura, així com un cas industrial. Els quatre primers varen ser experiments de laboratori per validar y avaluar alguns components de CoSTest tals com la generació dirigida per models dels casos de prova, el generador de mutants usat per prioritzar i seleccionar casos de prova, així com també el generador d'un esquema conceptual executable basat en ALF. En el quart estudi, es va aplicar l'anàlisi de mutacions per avaluar l'efectivitat i l'adequació dels casos de prova de CoSTest al detectar defectes en esquemes conceptuals mutats amb diferents tipus de defectes. En l'últim estudi, CoSTest va ser avaluat amb la participació d'usuaris finals mitjançant el Model d'Acceptació de Tecnologia (TAM) i el mètode d'entrevistes. Mentre que el TAM ens va permetre mesurar l'utilitat i facilitat d'ús d'una manera subjectiva, el mètode d'entrevistes ens va permetre identificar les limitacions i possibles millores que es poden implementar en la ferramenta. En general, els resultats varen ser favorables. CoSTest va ser altament valorat en la utilitat percebuda i la facilitat d'ús; també varem obtindre resultats positius amb respecte a l'efectivitat dels casos de prova.

RESUMEN

A pesar del escepticismo y dificultades en su adopción, el Desarrollo Orientado por Modelos (MDD, por sus siglas en inglés) está siendo usado y mejorado para proveer muchos beneficios inherentes a la industria. Uno de sus mayores beneficios es la capacidad de manejar la complejidad del desarrollo de software elevando el nivel de abstracción. Los modelos se expresan utilizando conceptos que no están relacionados con una tecnología de implementación específica (por ejemplo, Lenguaje de Modelado Unificado -UML, Lenguaje de Restricción de Objetos -OCL, Lenguaje de Acción para el Foundational UML - ALF), lo que significa que los modelos pueden ser más fáciles de especificar, mantener y documentar. Debido a que en una Ingeniería dirigida por modelos (MDE), los artefactos primarios son los modelos conceptuales, los esfuerzos se centran en su creación, prueba y evolución a diferentes niveles de abstracción a través de transformaciones, porque si un esquema conceptual tiene defectos, éstos se pasan a las siguientes etapas, incluida la codificación. Por lo tanto, uno de los retos para los investigadores y desarrolladores in MDD es poder identificar los defectos temprano, a nivel de esquemas conceptuales, ya que esto ayudaría a reducir los costos de desarrollo y mejorar la calidad del software.

Durante la última década, pocos trabajos de investigación se han realizado en esta área. Algunas de las causas de esta realidad son la alta complejidad teórica de probar esquemas conceptuales y la falta de soporte de software adecuado. Por lo tanto, este área de investigación admite nuevos métodos y técnicas, enfrentando retos como la generación de casos de prueba utilizando información externa a los esquemas conceptuales (es decir, los requisitos), la medición de una posible automatización, selección y priorización de casos de prueba, la necesidad de una herramienta de soporte eficiente que utilice una semántica estándar, la retroalimentación oportuna para apoyar el

proceso de aseguramiento de la calidad del software y facilitar la toma de decisiones basadas en el análisis y la interpretación de los resultados.

El objetivo de esta tesis es mitigar algunos de los problemas que afectan la validación de los esquemas conceptuales, proporcionando un nuevo marco de validación basado en pruebas que fue construido usando un desarrollo dirigido por modelos. El uso de MDD permite un aumento en la abstracción, automatización y reutilización que nos permite aliviar la complejidad de nuestro marco de validación. Además, al aprovechar las técnicas MDD (como el metamodelado, las transformaciones de modelos y los modelos en tiempo de ejecución), nuestro marco soporta cuatro fases del proceso de prueba: diseño de pruebas, generación de casos de prueba, ejecución de casos de prueba y la evaluación de los resultados. Esto es diferente a los enfoques de pruebas tradicionales, que, en general, sólo admiten algunas de estas fases.

Con el fin de proporcionar soporte de software para nuestra propuesta, hemos desarrollado CoSTest, un entorno de pruebas basado en el lenguaje ALF. Para asegurar que CoSTest ofrece la funcionalidad necesaria, primero identificamos un conjunto de requisitos funcionales. Luego, después de identificar estos requisitos, definimos la arquitectura y el ambiente de pruebas de nuestro marco de validación y, finalmente, implementamos la arquitectura en el contexto de Eclipse. CoSTest ha sido desarrollado para probar varias propiedades sobre el modelo ejecutable como la corrección sintáctica (es decir, todos los elementos del modelo se ajustan a la sintaxis del lenguaje en el que se describe), consistencia entre la parte estructural y el comportamiento (sus restricciones de integridad) y completitud (es decir, todos los cambios posibles en el estado del sistema se pueden realizar a través de la ejecución de las operaciones definidas en el modelo ejecutable). Para los modelos defectuosos, el informe de CoSTest devuelve una retroalimentación significativa que ayuda a localizar y reparar los defectos detectados.

El trabajo involucrado en la tesis fue validado mediante seis estudios usando casos encontrados en la literatura, así como un caso industrial. Los cuatro primeros fueron experimentos de laboratorio para validar y evaluar algunos componentes de CoSTest tales como la generación dirigida por modelos de los casos de prueba, el generador de mutantes usado para priorizar y seleccionar casos de prueba, así como también el generador de un esquema conceptual ejecutable basado en ALF. En el quinto estudio, se aplicó el análisis de mutaciones para evaluar la efectividad y la adecuación de los casos de prueba de CoSTest al detectar defectos en esquema conceptuales mutados con diferentes tipos de defectos. En el último estudio, CoSTest fue evaluado con la participación de usuarios finales a través del Modelo de Aceptación de Tecnología (TAM) y el método de entrevista. Mientras que el TAM nos permitió medir la utilidad y facilidad de uso de una manera subjetiva, el método de entrevista nos permitió identificar las limitaciones y posibles mejoras que se pueden implementar en la herramienta. En general, los resultados fueron favorables. CoSTest fue altamente valorado en la utilidad percibida y facilidad de uso; también obtuvimos resultados positivos con respecto a la efectividad de los casos de prueba.

CONTENTS

PART I. PREFACE	1
1. Introduction.....	1
1.1 Motivation	2
1.2 Problem Statement	4
1.3 Objectives	4
1.4 Thesis Context.....	5
1.5 Means of Achieving the Proposed Objectives.....	6
1.6 Thesis Outline	7
2. Research Methodology	11
2.1 Framework for the CoSTest Design Science Project.....	12
2.2 Statement of Research Goals and the Design Problem	13
2.3 Research Questions	17
2.4 Engineering, Design and Empirical Cycles	18
2.5 Summary.....	28
PART II. PROBLEM INVESTIGATION	31
3. Theoretical Framework	33
3.1 Concepts of Requirements Engineering	35
3.1.1 Modelling Requirements based on Communicational Analysis.....	37
3.2 Concepts of the Conceptual Schema Quality	38
3.2.1 Model Quality for Conceptual Schemas	39
3.2.2 Practices to improve the Quality of Conceptual Schemas ...	40

3.3 Concepts of the Model-driven Environment	44
3.3.1 MDA Definitions and Assumptions	45
3.3.2 Overview of the Metamodeling Architecture.....	46
3.3.3 UML CD-based Conceptual Schemas	47
3.3.4 Executable UML Conceptual Schema Under Test.....	48
3.3.5 Defect Types in UML-based Conceptual Schemas.....	50
3.4 Summary and Conclusions	53
4. Related Work of Conceptual Schema Validation	55
4.1 Dimensions of the Related Work	56
4.1.1 Domain.....	57
4.1.2 Quality Goal	57
4.1.3 Method	58
4.2 Generation, Selection, Prioritization and Execution of Test Cases	
61	
4.2.1 Test Case Generation	61
4.2.2 Test Case Selection and Prioritization.....	62
4.2.3 Test Case Execution	63
4.3 Comparison of Related Works.....	66
4.3.1 Dimension-Based Comparison.....	67
4.3.2 Testing feature Comparison.....	71
4.4 Summary and Conclusions.....	76
PART III. TREATMENT DESIGN.....	79
5. Validation Framework for Conceptual Schemas.....	81
5.1 Framework Overview	82
5.1.1 Phases of the Methodological Framework	83
5.2 Test Analysis	83

5.2.1 Requirements Specification based on Communicational Analysis.....	84
5.2.2 Modelling Requirements based on Communicational Analysis.....	85
5.3 Test Design	90
5.3.1 Test Data.....	91
5.4 Test Generation.....	91
5.4.1 Test Case Selection.....	91
5.4.2 Addressed Quality Goals	92
5.4.3 Test Types.....	92
5.4.4 Test Generation Criteria	94
5.4.5 Deriving test goals	94
5.4.6 Concrete and Executable Test Cases.....	95
5.5 Test Prioritization.....	106
5.6 Test Execution	111
5.6.1 Executable Conceptual Schema based on UML Class Diagram	111
5.6.2 Architecture and Testing Environment	116
5.6.3 Execution Trace	118
5.7 Test Evaluation.....	119
5.7.1 Verifying the Syntax Correctness.....	119
5.7.2 Validating the Semantic Correctness	120
5.7.3 Verifying the Unnecessary Elements	121
5.7.4 Validating the Completeness.....	122
5.8 Overview of the CoSTest Testing Process	126
5.9 Summary and Conclusions	127
6. Transformation Rules	131

6.1 An Overview of the MDT Process	132
6.2 Metamodels.....	133
6.2.1 Test Metamodel.....	134
6.2.2 Test Scenario Metamodel	136
6.2.3 Test Data Metamodel	137
6.3 Transformations	139
6.3.1 Transformation from Requirements Model to Test Model 139	
6.3.2 Transformation from Test Model to Test Scenario Model	147
6.3.3 Transformation from Test Model to Test Data Model	148
6.3.4 Transformation from Test Scenario Model to Test Scenario Model with Abstract Test Cases.....	148
6.3.5 Transformation from Test Data Model and Abstract Test Cases to Executable and Concrete Test Cases	150
6.3.6 Transformation from UML CD-based CS to Executable CS under test.....	151
6.4 Summary and Conclusions.....	156
PART IV. TREATMENT VALIDATION.....	157
7. CoSTest Tool Implementation.....	159
7.1 General Overview and Architecture	159
7.2 The Test Model Manager.....	162
7.2.1 Presentation Manager	163
7.2.2 Test Model Generator.....	164
7.2.3 Graph and Tree Builder.....	165
7.2.4 Element Report Generator	165
7.3 The Test Scenario Model Manager.....	165
7.3.1 Presentation Manager	166

7.3.2	Test Model Generator	167
7.3.3	Tree Builder	167
7.3.4	Element Report Generator	167
7.4	The Test-Data Manager.....	167
7.4.1	Presentation Manager.....	169
7.4.2	Web-based Generator.....	170
7.4.3	Requirement-based Generator	170
7.4.4	Database Manager	170
7.5	The CSUT Processor.....	171
7.5.1	Presentation Manager.....	171
7.5.2	CSUT Manager	174
7.6	The Test Processor	174
7.6.1	Presentation Manager.....	175
7.6.2	Test Manager.....	178
7.7	The Mutant Generator.....	179
7.7.1	Presentation Manager.....	180
7.7.2	Mutant Manager	181
7.8	The Batch Testing Processor	182
7.8.1	Presentation Manager.....	183
7.8.2	Batch Test Manager	184
7.9	Summary and Conclusions	184
8.	Validation and Evaluation of the CoSTest Tool	187
8.1	Validating the Effectiveness of CoSTest CSUT Processor.....	190
8.1.1	Experimental Design.....	190
8.1.2	Conclusions and Changes on the CoSTest CSUT Processor	

8.2 Validating the CoSTest Transformation Rules	194
8.2.1 Definition of Basic and Derived Metrics with Rule Scope.	196
8.2.2 Definition of Basic and Derived Metrics with Transformation scope	197
8.2.3 Experimental Design	201
8.2.4 Results and Discussion	205
8.3 Evaluating the CoSTest Mutant Generator.....	214
8.3.1 Experiment No 1: Evaluating the Mutation Operators Implemented in CoSTest.....	215
8.3.2 Experiment No 2: Validating the Effectiveness and Efficiency of Mutant Generator of CoSTest	218
8.4 Validating of the Effectiveness of CoSTest' Test Cases	222
8.4.1 Experiment Goal and Questions	222
8.4.2 Variables	223
8.4.3 Metrics	223
8.4.4 Hypotheses	224
8.4.5 Experimental Material	225
8.4.6 Procedure.....	226
8.4.7 Analysis of Results.....	230
8.4.8 Discussion.....	238
8.4.9 Analysis of the Threats to the Validity of the Results	240
8.4.10 Conclusions and Changes to the Tool	241
8.5 Evaluating CoSTest User Perceptions	242
8.5.1 Experiment Research Goal.....	242
8.5.2 Research Methodology	242
8.5.3 Experiment Context: The everis' Study Case	243
8.5.4 Experiment Reseach Questions	244

8.5.5 Case Selection.....	244
8.5.6 Methods of Data Collection	245
8.5.7 Experimental Subjects.....	246
8.5.8 Instrumentation	246
8.5.9 Experimental Procedure.....	247
8.5.10 Pilot Test.....	248
8.5.11 Analysis of the Threats to Validity.....	251
8.5.12 Answers to Experiment Research Questions	253
8.5.13 Discussion	254
8.6 Summary and Conclusions	256
PART V. FINAL DISCUSSION	259
9. Final Discussion	261
9.1 Summary of the Contributions of this Thesis.....	261
9.2 Thesis Impact	265
9.2.1 Publications	265
9.2.2 Academic Project Participation	268
9.2.3 Research Stay	268
9.3 A Work that Opens New Research Lines.....	268
9.3.1 Domain	268
9.3.2 Quality Goal.....	269
9.3.3 Method.....	269
REFERENCES	271
APPENDICES.....	287
Appendix A	289
Mutation Operators for UML CD-based Conceptual Schemas	289
Appendix B	291

Case Study: The Incident Management System	291
B.1 Test Analysis.....	291
B.1.1 Event Description Templates	291
B.1.2 Events Diagram	307
B.2 Test Design.....	307
B.2.1 Test Model	309
B.2.2 Test Scenario Model	309
B.2.3 Test Data	315
B.3 Test Case Generation.....	315
B.4 Mutant Generation.....	320
B.5 Test Execution.....	323
B.5.1 Generation of the Executable Conceptual Schema Under Test	323
B.5.2 Generation of the Execution Trace	327
B.6 Test Evaluation.....	328
B.7 Conclusions	330
Appendix C.....	331
Supplementary Material on the Evaluation Study.....	331
C1. Characterization Form	331
C2. CoSTest Tool Installation Guide	334
C3. Guideline with Task Template for VideoClub Case	336
C4. User Acceptance Form	336

LIST OF TABLES

Table 3.1 Quality Goals based on 6C quality model from Mohaghehi et al. [9].....	40
Table 3.2. Validation relevant methods for Conceptual Schemas	42
Table 3.3. Defect types in a UML-based model (excerpt taken from [43])	52
Table 4.1. Related approach comparison.....	68
Table 4.2. Testing features comparison	72
Table 5.1. Example of an Event Specification Template	88
Table 5.2. Test generation criteria for UML CD-based Conceptual Schema	94
Table 5.3. Mutation operators for CS FOM taken from [116].....	109
Table 5.4. Relationship between fault and reported defect	125
Table 6.1. Transformation rules for generation of the Test Model	141
Table 6.2. Transformation rules for generation of the Test Cases.....	141
Table 6.3. Transformation rules for generation of the Precedence relations.....	142
Table 6.4. Transformation rules for generation of the test items Assertions	142
Table 6.5. Transformation rules for generation of the test items Triggers.....	142
Table 6.6. Transformation rules for generation of the test items Services.....	143
Table 6.7. Transformation rules for generation of the test items Links	143
Table 6.8. Transformation rules for generation of the test items Parameters	144
Table 6.9. Requirements Metamodel constructs used in this transformation	145
Table 6.10. Transformation rules for generation of the Test Scenario Model	147

Table 6.11. Transformation rules for generation of the Test Scenario	147
Table 8.1. Elements of the Subject Conceptual Schemas.....	191
Table 8.2. Basic metrics for Semantic and Syntactic Correctness of a Rule	197
Table 8.3. Derived metrics for Semantic and Syntactic Correctness of a Rule	197
Table 8.4. GQM for M2M transformation validation	202
Table 8.5. Elements of the CSs.....	203
Table 8.6. Elements of the requirements model included in the five examples	206
Table 8.7. Elements of the Test Model generated for the five example	206
Table 8.8. Results of SyC_T1 and SeC_T1 for the five cases	208
Table 8.9. Elements of the Test Scenario Model generated for the five examples	211
Table 8.10. Results of SyC_T2 and SeC_T2 for the five cases	212
Table 8.11. Specification of hypotheses	224
Table 8.12. Elements of the Subject Conceptual Schemas.....	225
Table 8.13. Faults and Fault Types detected by Mutant type	231
Table 8.14. Shapiro-Wilk Normality Tests.....	232
Table 8.15. Mann-Whitney U Test for Rate of Fault Detection by Mutant Type.....	233
Table 8.16. Tests of Normality of Shapiro-Wilk	233
Table 8.17. Mann-Whitney U Test for Rate of Fault Type Detection ^a	234
Table 8.18. Mutation Score by Mutant type.....	235
Table 8.19. Mutation Score of CoSTest Test Suites for First Order Mutants.....	236
Table 8.20. Mutation Score of CoSTest Test Suites for High Order Mutants.....	237
Table 8.21. Shapiro-Wilk Normality Tests.....	238
Table 8.22. Mann-Whitney U Test for Mutation Score by Mutant Type ^a	238
Table 8.23. Detail of the Activities	248

Table 8.24. Specification of hypotheses.....	253
Table A.1. Mutation Operators defined for a UML CD-based CS taken from [115].....	289
Table B.1. Communication Structure for TECH1	292
Table B.2. Communication Structure for USR1	293
Table B.3. Communication Structure for PLAN1	293
Table B.4. Communication Structure for INC1	294
Table B.5. Communication Structure for INC2	295
Table B.6. Communication Structure for INC3	295
Table B.7. Communication Structure for INC4	296
Table B.8. Communication Structure for INC5	297
Table B.9. Communication Structure for INC6	297
Table B.10. Communication Structure for INC7	298
Table B.11. Communication Structure for INC8	298
Table B.12. Communication Structure for INC9	299
Table B.13. Communication Structure for INC10	300
Table B.14. Communication Structure for INC11	300
Table B.15. Communication Structure for INC12	301
Table B.16. Communication Structure for INC13	302
Table B.17. Communication Structure for INC14	302
Table B.18. Communication Structure for INC15	303
Table B.19. Communication Structure for INC16	303
Table B.20. Communication Structure for INC17	304
Table B.21. Communication Structure for INC18	305
Table B.22. Communication Structure for INC19	305
Table B.23. Communication Structure for INC20	306
Table B.24. Communication Structure for INC21	306
Table B.25. Values for variables of test model for Incident Management	316
Table B.26. List of First Order Mutants generated for the case study	322
Table B.27. Testing results for the mutants of Table B.26	328

LIST OF FIGURES

Figure 1.1 Context of research work	6
Figure 2.1. Framework for design science of the CosTest project	13
Figure 2.2 Goal Structure of the design science research project for CoSTest	16
Figure 2.3. Design cycle for the CoSTest project (part 1)	19
Figure 2.4. Design cycle for the CoSTest project (part 2)	20
Figure 2.5. Design cycle for the CoSTest project (part 3)	21
Figure 3.1. Research areas involved in this work	34
Figure 3.2. Communication Analysis requirements levels and workflow [21]	37
Figure 3.3. Excerpt of the Metamodel of an UML Class Diagram [40] ..	47
Figure 3.4. Excerpt of UML-CD-based CS for Video Club case	49
Figure 3.5. Example of constraints for the Video Club system	50
Figure 3.6. Relationships among conceptual entities	51
Figure 4.1. Related Work dimensions	56
Figure 5.1. Overview of the validation Framework.....	83
Figure 5.2. Excerpt of a CA model for the Video Club case.	86
Figure 5.3. Examples of test goals generated for Video Club CS.....	95
Figure 5.4. Test Case Structure.....	96
Figure 5.5. UML class diagram for Video Club CS.....	97
Figure 5.6. VideoClub CS with examples of pre, post-conditions and invariants	102
Figure 5.7. Example for validating pre-, post-conditions and invariants	102
Figure 5.8. Example of an invariant.....	103
Figure 5.9. Example of test case for asserting the non-occurrence of events	104
Figure 5.10. Example of test case validating a derivation rule	106
Figure 5.11. Selection process of the mutation operators.....	108

Figure 5.12. Excerpt of the Metamodel of an UML Class Diagram [40]	111
Figure 5.13. Textual definition for the package VideoClub by using ALF language	114
Figure 5.14 Overview to generate an executable CSUT	114
Figure 5.15. Testing environment to test Conceptual Schemas	116
Figure 5.16. Example of an execution trace for Video Club CS	119
Figure 5.17. Excerpt of the CS with a syntactically incorrect code	120
Figure 5.18. Example of a CS with the corrected Alf code	120
Figure 5.19. Excerpt of the VideoClub CS with a semantic incorrect defect	121
Figure 5.20. Example of the VideoClub CS with the corrected semantic defect	121
Figure 5.21. Example of comparison of elements used in a coverage analysis	122
Figure 5.22. Excerpt of the CS with a missing defect	123
Figure 5.23. Excerpt of a corrected CS	123
Figure 5.24. Example of the test case	124
Figure 5.25. Example of execution trace	124
Figure 5.26. Extended UML class diagram for Video Club CS	125
Figure 5.27. Overview of the testing process	126
Figure 6.1. An overview of our MDT approach	132
Figure 6.2. Overview of the sequence of proposed transformations.	133
Figure 6.3. Metamodels for the first transformation adapted from [121]	135
Figure 6.4. OCL Constraints for Test Metamodel	136
Figure 6.5. Test Scenario Metamodel adapted from [121]	137
Figure 6.6. Test Data Metamodel	138
Figure 6.7. Structure of T1 Transformation	140
Figure 6.8. Examples using graphical concrete syntax of (a) RM, (b) TM, and (c) modified TM	146
Figure 6.9. Example of the first rule of the ATL transformation CA2TM	146
Figure 6.10. Structure of T2 Transformation	148

Figure 6.11. Structure of T3 transformation	148
Figure 6.12. Partial Acceleo code of transformation	149
Figure 6.13. Test Scenario with abstract test cases	150
Figure 6.14. Example of a concrete and executable test case for VideoClub CS	151
Figure 6.15. Acceleo transformation rule for UML package	151
Figure 6.16. Partial definition for the class VideoClub by using ALF language	152
Figure 6.17. Association and Aggregation of Order example using ALF language	152
Figure 6.18. Partial view of the ALF unit including an inheritance relation	153
Figure 6.19. Example of a constraint translated to ALF code.....	154
Figure 6.20. Example of a derived association using ALF code	155
Figure 6.21. An example of class association	155
Figure 7.1. Screenshot of the CoSTest tool support.....	162
Figure 7.2. The CoSTest tool architecture	162
Figure 7.3. Test Model Manager design.....	163
Figure 7.4. Screenshot with a test configuration example of the CoSTest tool	164
Figure 7.5. Screenshot with a test model example of the CoSTest tool	165
Figure 7.6. Test Scenario Model Manager design	166
Figure 7.7. Screenshot of a test scenario model example in the CoSTest tool	168
Figure 7.8. Test Data Manager design.....	168
Figure 7.9. Screenshot for the data concretization in the CoSTest tool	171
Figure 7.10. CSUT Processor design	172
Figure 7.11. Screenshot for editing an executable CSUT in the CoSTest tool	172
Figure 7.12. Screenshot for showing the parser results in the CoSTest tool	173

Figure 7.13. Screenshot for showing the CSUT elements in the CoSTest tool.....	173
Figure 7.14. Test Processor design	175
Figure 7.15. Screenshot of the test configuration in the CoSTest tool.....	175
Figure 7.16. Screenshot of a test suite management example in the CoSTest tool	176
Figure 7.17. Screenshot of a test execution report in the CoSTest tool	176
Figure 7.18. Screenshot of the Summary Generation tab of the CoSTest tool.....	177
Figure 7.19. Screenshot of a log and coverage report in the CoSTest tool.....	178
Figure 7.20. The Mutation UML tool architecture.....	180
Figure 7.21. Application of five mutation operators for our CS example	182
Figure 7.22. Batch Testing Processor design.....	183
Figure 7.23. Screenshot for Batch Testing of the CoSTest tool	184
Figure 8.1. i-th iteration of the experiment applying the CoSTest tool	193
Figure 8.2. Example of the calculation of the metrics SyC_T1 and SeC_T1.....	200
Figure 8.3. Process to evaluate a M2M _i in our proposal	204
Figure 8.4. Structure of T1 Transformation with the identified problems	210
Figure 8.5. i-th iteration of the experiment applying the CoSTest tool	217
Figure 8.6. i-th iteration of the experiment applying the CoSTest tool	220
Figure 8.7. Steps taken in experimental process	227
Figure 8.8. Application of five mutation operators on Video Club CS	229
Figure 8.9. Excerpt of a Constraint mutated by WCO8 operator	230
Figure 8.10. Box-plot for Rate of Fault Detection by Mutant Type	232
Figure 8.11. Box-plot for Rate of Fault Type Detection by Mutant Type	234

Figure 8.12. Box-plot for Mutation Score by Mutant Type	235
Figure 8.13. Example of an assertion conditional	239
Figure 8.14. Experimental Procedure.....	247
Figure B.1. Partial view of the message structure in the GREAT tool [141]	307
Figure B.2. Event Diagram using Communication Analysis	308
Figure B.3. Test Model for IM case study	309
Figure B.4. Test cases of the test scenario #2	320
Figure B.5. Excerpt of the Conceptual Schema for Incident Management System	321
Figure B.6. ALF unit for PMO class	323
Figure B.7. ALF unit for Incident_external_company association.....	323
Figure B.8. ALF unit for EXTERNAL_COMPANY_ANALYSYS class.....	324
Figure B.9. ALF unit for incident_resource_allocation association.....	324
Figure B.10. ALF unit for technician_resource_allocation association.....	324
Figure B.11. ALF unit for INCIDENT class	325
Figure B.12. ALF unit for RESOLUTION_PLAN class	326
Figure B.13. ALF Unit for RESOURCE_ALLOCATION class	326
Figure B.14. ALF unit for TECHNICIAN class	326
Figure B.15. ALF unit for USER class	326
Figure B.16. ALF unit for user_incident association	327
Figure B.17. Example of Execution Trace for the MAS_2 mutant	327
Figure B.18. Defect report obtained in the testing process for MAS_2 CS	329
Figure B.19. Coverage report obtained in the testing process for MAS_2 CS	329
Figure C.1. Characterization form: Demographic data.....	331
Figure C.2. Characterization form: Experience (1)	332
Figure C.3. Characterization form: Experience (2)	333
Figure C.4. Characterization form: Experience (3)	334
Figure C.5. Guideline for VideoClub case (1).....	337
Figure C.6. Guideline for VideoClub case (2).....	338
Figure C.7. User Acceptance Form: Perceived Usefulness	338
Figure C.8. User Acceptance Form: Perceived Ease-of-Use	339

PART I.

PREFACE

Chapter 1

INTRODUCTION

A wide range of software engineering methods supports the development of information systems (IS) by considering requirements engineering as an essential activity, which specifies general knowledge about the IS domain and the functions it has to perform. *In the **Information Systems** field, this knowledge is called a conceptual schema¹ [1]. According to Johnson and Henderson [2] a **Conceptual Schema** or **Conceptual Model** is “a high-level description of an application. It enumerates all concepts in the application that users can encounter, describes how those concepts relate to each other, and how those concepts fit into tasks that users perform with the application”.*

In Model-Driven Development, the main artefacts are conceptual schemas (CS) or models, and efforts are focused on their creation, testing and evolution at different levels of abstraction through transformations. If a conceptual schema has defects, these are passed on to the following stages, including coding. Therefore, techniques for improving the quality of conceptual schemas must be implemented to

¹In this thesis the terms "conceptual schema", "conceptual model" and "model" are considered similar.

ensure the correct generation of final software products. One of the challenges of Model-Driven Development is to be able to generate test cases from the requirements, not only to identify defects, as well as to validate requirements early on, at the level of conceptual schemas, so that appropriate decisions can be taken based on the results of the validation process, to help reduce development costs and improve software quality. In this work we designed an approach for testing-based conceptual schema validation in order to improve quality.

The rest of this chapter is organized as follows: Section 1.1 gives an explanation of why this research is important. Section 1.2 summarizes the problem resolved in the present thesis. Section 1.3 details the defined thesis objectives. Section 1.4 presents the context of this work. Section 1.5 summarizes the means of achieving the main objective. Finally, Section 1.6 gives an overview of the structure of this document.

1.1 Motivation

Despite much scepticism and many problems [3], **Model-Driven Development** (MDD) is being used and improved in order to provide multiple inherent potential benefits for industry [4], [5]. One of its greatest benefits is the ability to handle the complexity of software development by raising the abstraction level. Models are expressed using concepts that are not related to a specific implementation technology (e.g. Unified Modelling Language -UML, Object Constraint Language -OCL, Action Language for Foundational UML -ALF), which means that the models can be easier to specify, understand, maintain and document. As in Model-Driven Engineering (MDE), the primary artefacts are the conceptual models, and ensuring their quality at an optimum level is still challenging for researchers and developers.

Although verification² and validation³ (V&V) are highly related to the concepts of quality and **software quality** assurance, very few MDD

² Verification is to check that the conceptual schema meets its stated functional and non-functional requirements [1].

tools incorporate these activities into their development process. The OO-Method (OOM) [6], a Model Driven Architecture (MDA) approach, is a model-driven initiative with a technical multi-view (structural model, dynamic model, functional model and presentation model), where the structural view is the basis for the automatic derivation of the other views, and this feature helps to minimize problems such as multi-view specifications and synchronization, integration and change propagation. The OO-Method has been successfully implemented in industry through the Integranova⁴ commercial tool (previously known as OLIVANOVA). This tool manages the syntactic verification of conceptual schemas (e.g. syntactic correctness) [6], but it still does not validate whether the model built meets the requirements and expectations of the stakeholders.

With the ever-increasing complexity of software systems, the ability to identify the vast majority of defects early on at the model level is a challenge that if met could help to reduce development costs and improve software quality [7]. The list of open problems presented in [8] by Olivé includes the Complete and Correct Conceptual Schemas.

However, to assess the quality of a conceptual schema, we need a quality model. In the literature, we can find several proposals, e.g. [9], [10]. Although, Genero et al. [11] suggest that more work is needed on model quality assessment. We will aim to set the quality properties that can be improved using testing techniques.

Testing is part of a process of V&V, where the conceptual schema operates under controlled conditions, (1) to verify that it behaves as specified; (2) to detect defects, and (3) to validate user requirements [12]. Therefore, (i) the close integration between model and code in a model-driven development, (ii) the development of high-level

³ Validation is to ensure that the conceptual schema meets the customer's expectations [1].

⁴ <http://www.integranova.com/>

languages suited for modelling CS (like UML/OCL with the ALF language), generate the need to develop verification and validation strategies to be applied early in the software life cycle (e.g. at CS level) and to locate and point out defects in realistic schemas with minimum cost.

This work aims to define a testing-based validation framework for multi-view conceptual schemas (i.e. structural and behavioural). We will focus on adapting testing techniques for Model-driven environments, such as the OO-Method approach, because we believe that testing can be a very effective and efficient way to identify defects early on, and can play an important role in the validation of conceptual schemas.

1.2 Problem Statement

Requirements errors are the most common cause of defects in system development projects [13]. This suggests that it would be more effective and efficient to focus quality assurance efforts on the early phases, in order to catch defects as soon as they occur. In MDD, the ability to identify defects early on is still a challenge that, if it were met, could help to reduce development costs and improve the quality of delivered software systems [7] [8]. Lightweight testing techniques for improving the quality of the conceptual schemas must be implemented. These techniques should be able to find defects with minimum effort, and without the need for a strong testing background.

The starting point of this PhD Thesis begins with the statement of the research problem “Improve the quality of the conceptual schemas built in a model-driven environment in order to reduce the development costs and improve the quality of delivery software systems”.

1.3 Objectives

The main objective of this PhD thesis is to “Design a testing-based validation framework to improve the quality of conceptual schemas

built in a Model-driven environment”. This main objective is dependent on the achievement of the following specific objectives:

- Define the conceptual framework related to the conceptual schema validation by using testing techniques in a Model-driven environment.
- Design a framework for testing-based validation of conceptual schemas integrated into a Model-driven environment.
- Validate the contribution of the testing-based framework in ensuring the quality of conceptual schemas.

1.4 Thesis Context

This thesis aims to validate conceptual schemas by using model-based testing techniques. Our approach contributes to improving the quality of conceptual schemas built in a Model-driven environment, by detecting and correcting defects at an earlier phase than traditional testing techniques used successfully at code level.

Figure 1.1 shows an overview of the design of the proposed solution, which is based on a series of model transformations for automatically generating test cases from a requirements model. These test cases are used for testing the conceptual schema, previously prepared for use as a testing artefact (conceptual schema under test). Then, the output will be the list of defects properly classified, which will serve as feedback for relevant stakeholders like the analyst, modeller or project manager.

This thesis has been developed in the context of the STAQ (Software Testing and Quality) research group of the PROS Center (Centro de Investigación en Métodos de Producción de Software), Department of Information Systems and Computation (DSIC: Departamento de Sistemas de Información y Computación) of the Universitat Politècnica de València, Spain.

The work has been supported by Universidad de Cuenca and Secretaría de Nacional de Educación Superior, Ciencia y Tecnología - SENESCYT of Ecuador, and has been received financial support from

the SHIP (SMEs and HEIs in Innovation Partnerships, ref: EACEA/A2/UHB/CL 554187), PERTEST (TIN2013-46928-C3-1-R), European Commission (CaaS project) and Generalitat Valenciana (PROMETEOII/2014/039).

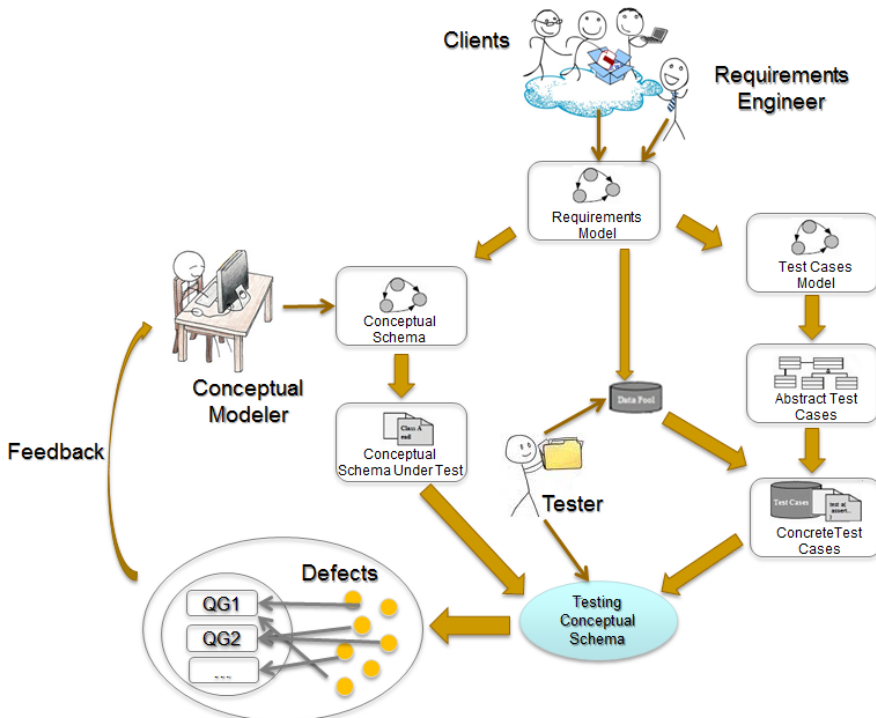


Figure 1.1 Context of research work

1.5 Means of Achieving the Proposed Objectives

In order to achieve the main objective, we identify three means:

- a) Software resources. Software tools and standards will be required to perform the proposed approach through the process such as 1) generate a test cases model from user's requirements, 2) generate concrete test cases, 3) specify and transform the conceptual schema to an executable form, 4) execute the testing, 5) report the results of this process and 6) validate the solution design.

- b) Expert support. Due to the multidisciplinary nature of this PhD project proposal, it is supervised by three senior researchers, who are respectively experts in Requirements Engineering, Model-Driven Engineering and Software Testing & Quality. Their advice and valuable feedback will be very helpful to accomplish the research goal of this work.
- c) Financial resources. This work is being supported by the Secretary of Higher Education, Science and Technology (SENESCYT: Secretaría Nacional de Educación Superior, Ciencia y Tecnología), and University of Cuenca, both public bodies of the Republic of Ecuador. Additionally, the work is being developed at the STAQ (Software Testing and Quality) research group of the PROS Center (Centro de Investigación en Métodos de Producción de Software), Department of Information Systems and Computation (DSIC: Departamento de Sistemas de Información y Computación) of the Universitat Politècnica de València, Spain.

1.6 Thesis Outline

We have divided the thesis into five parts and three appendices. Part I is the preface, Part II presents the problem investigation, Part III provides the treatment design, Part IV presents the treatment validation, and finally, Part V provides the final discussion. Here we describe the outline of the thesis.

Part I – Preface

Chapter 1 presents an overview of the research including the motivation, problem statement, hypothesis and objectives addressed in this PhD thesis as well as the context and means to achieve the proposed objectives.

Chapter 2 describes the framework for the design science project applied in this thesis, as well as the research goals, research questions and the methodology followed.

Part II – Problem Investigation

Chapter 3 provides the reader with the theoretical framework (knowledge) that is required for understanding the overall work.

Chapter 4 summarizes the main research efforts that have been carried out in Validation of Conceptual Schemas.

Part III –Treatment Design

Chapter 5 describes the phases of the construction process of a model-driven validation framework for conceptual schemas.

Chapter 6 details the metamodels and transformations rules used to generate the test scenarios model from the requirements models, which contains the test suite with the abstract test cases.

Part IV – Treatment Validation

Chapter 7 presents the tool support that has been developed to support the methodological detailed in Chapters 5 and 6 as well as its validation. This chapter presents the architecture and functionality of the CoSTest tool.

Chapter 8 summarizes (1) a validation study of the two first model-to-model transformations for the purpose of validating them with respect to their syntactic and semantic correctness, (2) two laboratory experiments for the purpose of evaluating the mutation operators and the effectiveness and efficiency of CoSTest to generate mutants that are used to evaluate the effectiveness of CoSTest' test cases and that also served to prioritize the test cases; (3) a comparative experiment for the purpose of measuring CoSTest' test cases in terms of effectiveness; and (4) evaluation of user perceptions during the defect correction process using the CoSTest' report in an industrial case.

Part V- Final Discussion

Chapter 9 draws some conclusions about the present thesis and summarizes the main contributions and publications that we obtained. It also discusses future lines of research, which are the in line with the limitations of the present work.

Appendix A includes the list of mutation operators used during the build of the CoSTest' mutant generator (Chapter 7) and during the validation and evaluation of CoSTest described in Chapter 8.

Appendix B describes a case study aimed to exemplify our model-driven validation framework. The appendix applies the CoSTest tool to an example of a conceptual schema that represents an excerpt of the Incident Management system defined by the everis company.

Appendix C includes material used during the evaluation study described in Chapter 8 (see Section 8.5).

Chapter 2

RESEARCH METODOLOGY

The nature of this research work lends itself to the use of the design science framework [14] in the form of a new artefact, the CoSTest framework.

Design science is the design and investigation of artefacts in context [15]. In this PhD thesis, we design CoSTest to support stakeholders (e.g. modellers and testers) in their tasks of modelling and validating conceptual schemas in the requirements, analysis and design stages during the development of an information system. CoSTest is therefore an artefact in the context of validating the stakeholder's requirements at the conceptual schema level.

In this chapter, we introduce the Design Science Research framework and describe the methodology applied. The chapter is organized as follows: Section 2.1 presents the methodological framework used in this thesis. Section 2.2 defines the research goals. Section 2.3 describes the research questions. Section 2.4 presents the methodology followed by summarizing and grouping the activities in the design cycle and empirical cycle applied. Section 2.5 gives a summary of the entire chapter.

2.1 Framework for the CoSTest Design Science Project

Since we conceived this PhD Thesis as a design science project, it consists of two activities (i.e. design and investigation). It iterates over two issues involved in solving design problems (e.g. related to the design of artefact CoSTest to improve a problem context) and answering knowledge questions (e.g. related to knowledge of CoSTest and the interaction between CoSTest and the context in which it is applied). However, these problems can create new problems (e.g. building a prototype of the artefact, simulating its context, or designing a measurement instrument) because an artefact may interact differently in different contexts.

These interactions may even contribute to stakeholder goals in one context but create obstacles to goal achievement in another. Therefore, *a design science project is never restricted to one kind of problem only and the design researcher should therefore study the interaction between artefacts and contexts rather than artefacts alone or contexts alone* [15].

Figure 2.1 shows the framework for the design science of the proposed testing framework, in which the interactions between design and investigation are extended to the social and knowledge contexts.

CoSTest's social context consists of stakeholders, who may either affect or may be affected by the project, potential users like modellers, testers, researchers, etc. who are part of organizations that need to validate conceptual schemas during the development of an information system, and sponsors that provide the financial support for this PhD thesis.

In the knowledge context, CoSTest is involved with very diverse theories, such as model-based engineering, particularly founded on the model-driven development paradigm, requirements engineering for analysis of information systems from a communicational perspective

supported by the Communication Analysis method, software quality for defining the test automation framework, lessons learned from the experience of researchers in earlier design science projects, practical knowledge in Eclipse Modelling Framework for implementing model-driven and model-based tools, and several conceptual schemas taken from different testing domains.

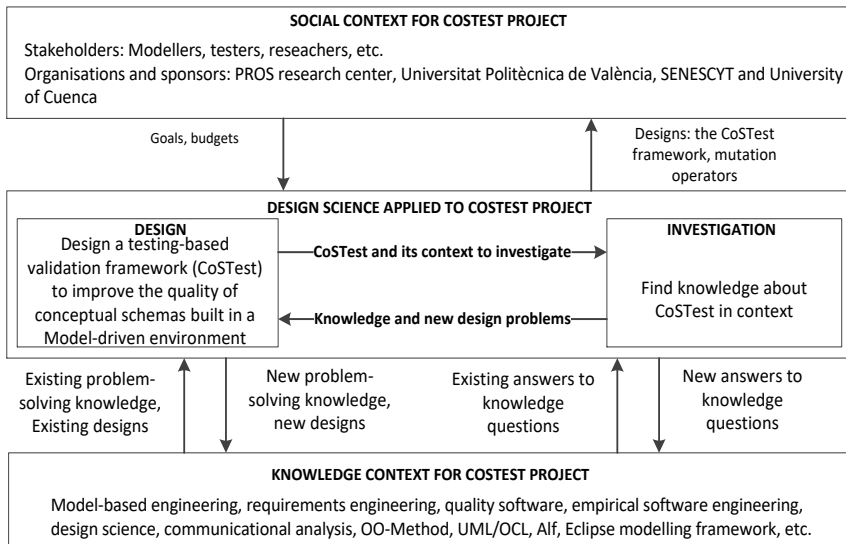


Figure 2.1. Framework for design science of the CoStest project

This framework is used to define the thesis' research goals.

2.2 Statement of Research Goals and the Design Problem

In this work, we can distinguish research goals from the external stakeholder's goals (sponsors and potential end-users).

In addition to our intrinsic motivation as researchers to answer the knowledge questions, as well as to design and test the new artefact (CoStest), we want to improve the way in which the quality assurance of conceptual schemas is performed in an early phase of the software lifecycle.

We therefore promote the use of conceptual schemas as a high level analysis of information systems to specify the functionality of an IS and to generate the respective test cases.

The use of different artefacts by requirements analysts, modellers, testers and developers is avoided, thus making their work easier. By using models it is possible to automate the testing process and reduce the cost, increase the effectiveness of the tests and optimize the testing cycle.

In addition, the Software Engineering community (potential external stakeholder) proposes the use of testing techniques as a mechanism to contribute to ensuring the software product quality [12]. Following this proposal, our motivation is to define a testing-based validation approach to support quality assurance process of conceptual schemas in a Model-driven environment.

The sponsors (academics) of this PhD thesis supported our research, which is not a market-oriented project (with a well-defined possible utility of the designs and knowledge that will come out of this project).

This thesis is thus an exploratory project in which the aim of the researchers was to explore the possibility of a model-driven testing framework for conceptual schemas.

This exploratory research is therefore motivated by the research goals regardless of whether or not it satisfies a set of specific stakeholders or end-user needs.

Figure 2.2 shows the goal hierarchy of the CoSTest design science research project. Since this project is an exploratory research, we focus on the design science research goals.

However, we have also included some speculative social context goals. Starting from the bottom up in Figure 2.2 the lowest level goals (instruments design goals) are to define the requirements for the

model-driven testing framework (G1), and build prototypes of the CoSTest testing framework (G2).

These instruments were used to answer knowledge questions such as: (a) how can available treatments detect defects in conceptual schemas? (G3), (b) what is the effectiveness and quality of test suites generated by the CoSTest prototype? (G4) and (c) what are the effects of the prototype's implementation as regards stakeholder's perceptions of its usefulness, user experience and user satisfaction? (G5).

This knowledge is generalizable and could be used to predict the effectiveness and efficiency of the CoSTest framework to detect defects in conceptual schemas (G7). Answering these questions also contributed to the artefact design goal of designing a model-driven testing framework to improve conceptual schema quality in a Model-driven environment (G6). This in turn contributes to the goal of problem context improvement.

The CoSTest framework will be part of a software testing lifecycle to be used in model-driven/based software development projects.

The sponsor's goal (speculative) is to reduce development costs and improve the quality of the delivered software system (G8). G7 and G8 are high-level goals (speculative) that would be achieved in the future.

However, we include these goals in this document because the social and prediction goals are part of the goal structure of the CoSTest framework.

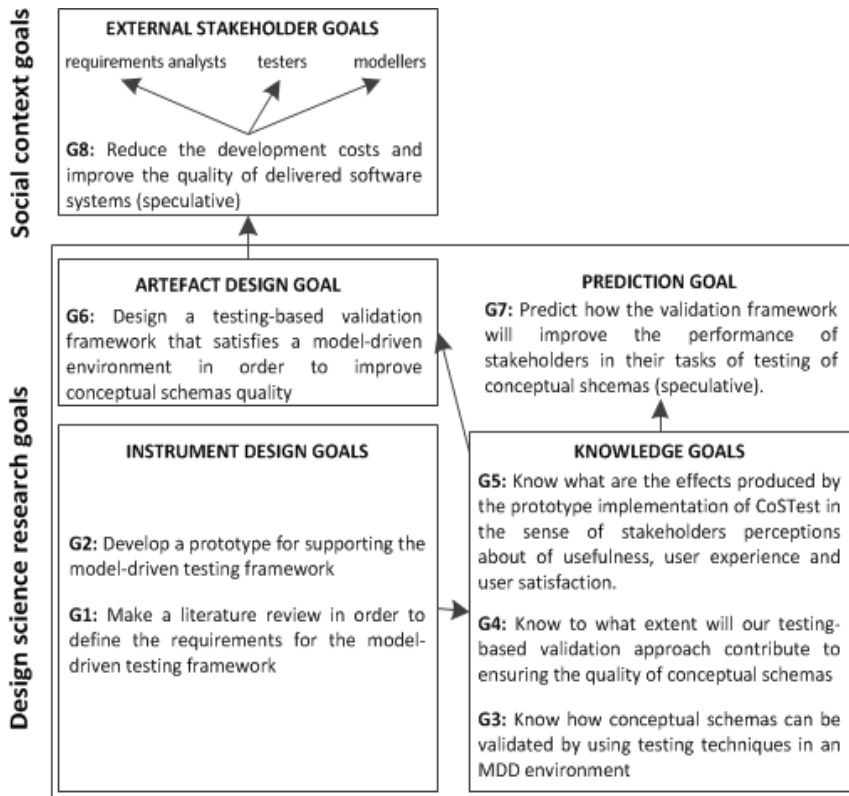


Figure 2.2 Goal Structure of the design science research project for CoSTest

The goal structure has various challenges (design problems and knowledge questions) that need to be overcome.

Below we introduce the main design problem statement (a.k.a. technical research problem -TRP) derived from the artifact design goal (G6):

Improve the quality of the conceptual schemas by designing a testing-based validation framework that satisfies a model-driven environment in order to reduce the development costs and improve the quality of delivery software systems.

2.3 Research Questions

In order to provide a solution to the technical research problem mentioned above, we present the list of research questions (RQ) derived from design goals (DP) and knowledge questions (KQ).

RQ₁ (KQ): *What are the testing-based validation techniques that can be used on conceptual schemas in an MDD environment?* This research question is related to G1. In order to answer this question, we have considered the following sub-questions:

- RQ_{1.1} (KQ): Which testing techniques can be effectively used or adapted for conceptual schemas?
- RQ_{1.2} (KQ): What kind of defects can be detected in the conceptual schemas using a testing strategy?
- RQ_{1.3} (KQ): Which Model-driven environment requirements should be considered when developing the testing-based approach?
- RQ_{1.4} (KQ): How can an approach for testing-based validation of conceptual schemas be integrated into a *Model-driven* environment?
- RQ_{1.5} (KQ): Which of the existing quality assurance frameworks is the most suitable for use in *Model-driven* environments?
- RQ_{1.6} (KQ): What quality properties can be improved using testing techniques in conceptual schemas?

If conceptual schema testing is feasible, as implied in the first question of this research work, then another main research question arises:

RQ₂ (TRP): *How to build a testing framework that detects defects at conceptual schema level so that it contributes to the achievement of the quality of software systems in a Model-driven environment?* This research question is related to G6, which refers to the main research goal. In order to answer RQ₂, the following specific research questions must be addressed:

- RQ_{2.1} (DP): How to build a prototype tool that supports the CoSTest framework so that researchers can validate the proposed treatment on UML-CD based conceptual schemas? This research question is based on G2.

— RQ_{2.2} (KQ): How can the treatment detect defects in conceptual schemas? This research question is related to G3.

RQ₃ (KQ): *To what extent will our testing-based validation approach contribute to ensuring the quality of conceptual schemas?* In order to answer this question, we have considered the following sub-questions:

— RQ_{3.1} (KQ): What is the effectiveness and adequacy of test suites generated by the CoSTest prototype? This is an empirical research question related with G4.

— RQ_{3.2} (KQ): What effects are produced by the prototype as regards stakeholder’s perceptions about its usefulness, user experience and user satisfaction? This is an empirical research question related with G5.

2.4 Engineering, Design and Empirical Cycles

Since the development of CoSTest (RQ1) is a design science research project, it follows the design cycle proposed by Wieringa [15] to describe design and research activities.

For tasks related to the *design problem*, Wieringa’s **design cycle** describe the activities related to the following three tasks: problem investigation (T1), treatment design (T2) and treatment validation (T3). The design cycle is part of the **engineering cycle**, in which a designed and validated treatment is implemented in the problem context, and the implementation is evaluated [15] (see Figures 2.3 -2.5).

In **problem investigation**, we seek to understand how to validate and verify conceptual schemas by using testing techniques, and the what current approaches that have been proposed to achieve this. To do this, from the existing surveys and systematic reviews concerning software testing, we select some testing strategies as possible candidates to implement our approach (RQ1.1, RQ1.2).

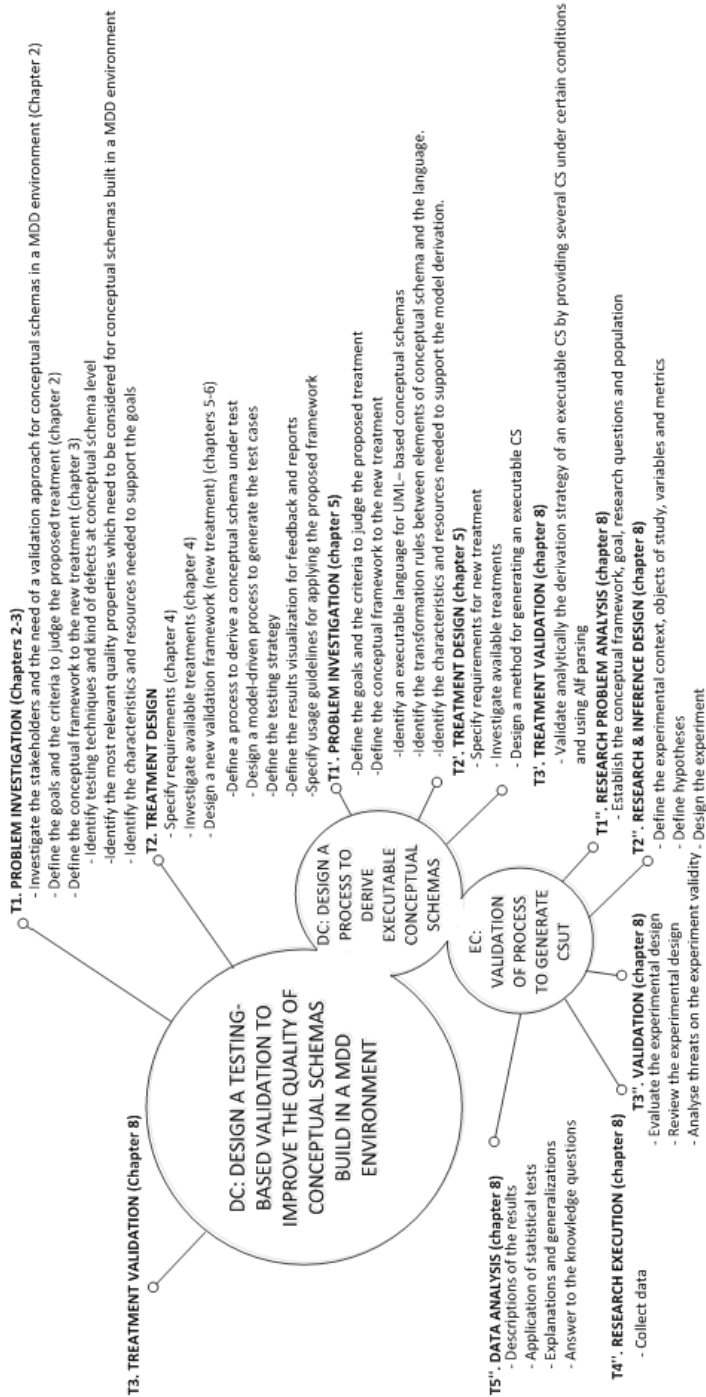


Figure 2.3. Design cycle for the CoStest project (part 1)

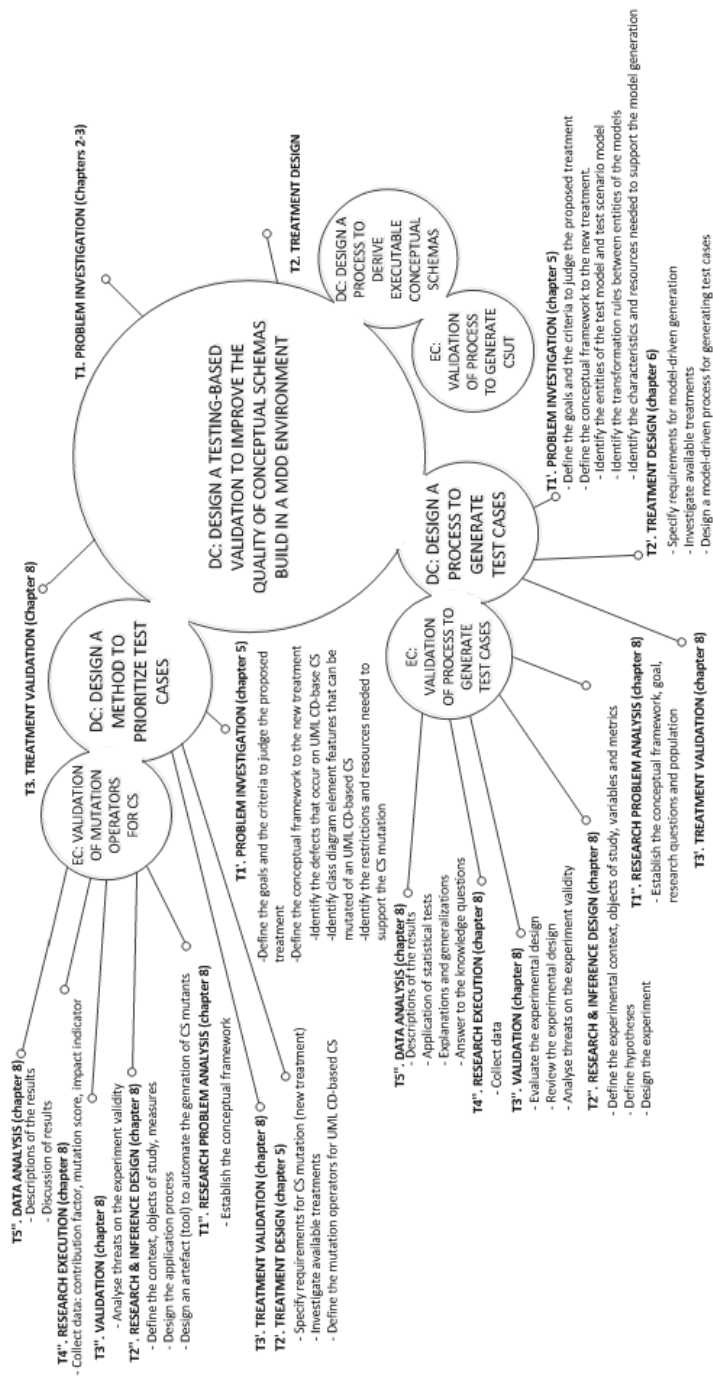


Figure 2.4. Design cycle for the CoSTest project (part 2)

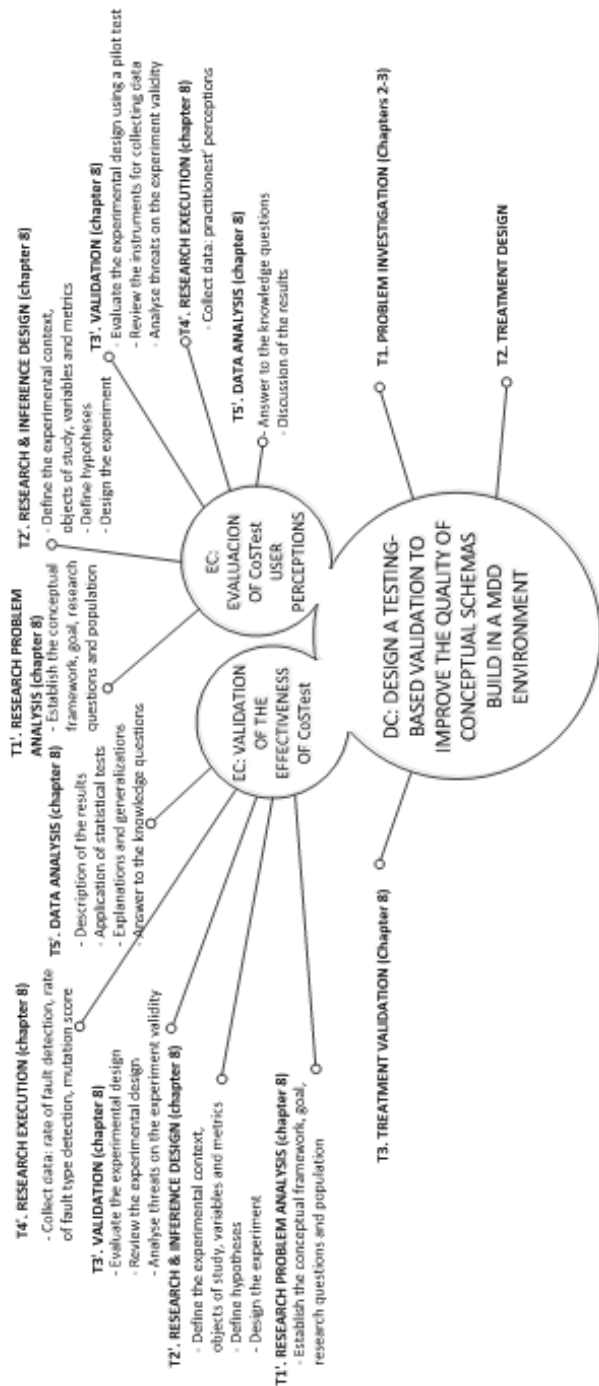


Figure 2.5. Design cycle for the CoStest project (part 3)

We also identify the most relevant quality properties which need to be considered for conceptual schemas built in a Model-driven environment (RQ1.5, RQ1.6) as well as the characteristics and resources needed to support the testing-based validation approach (RQ1.3). By considering these identified quality properties, we can analyse and identify the properties that are affected by defects that have so far been detected in conceptual models.

Based on the relationships between quality properties and defects, we can thus evaluate the selected testing techniques in order to identify those that can be effectively used or adapted for our purpose (RQ1.4). One of the outcomes of this phase will be a conceptual framework that should aid our understanding of the proposed approach, as well as identify the stakeholders, their goals and any problems with the existing solutions. This will provide the criteria to judge the treatment design.

Treatment design is characterized by its iterative nature. In this task, we specified the requirements and context assumptions for the new treatment based on design specifications identified in the previous phase, and our own logical reasoning. Based on this and on the results of the problem investigation, the researchers designed several versions of the CoSTest model-driven based testing framework for conceptual schemas in order to answer RQ2.1 and RQ2.2. The next iteration refines the solution by adding insights from several interviews with academic and industrial experts to improve the approach. Further iterations use inputs from the analysis of laboratory experiments and the results of treatment validation tasks.

Finally, the **treatment validation** task solves a knowledge problem which asks if the treatment design (prototype) is effective (e.g. finding defects capability, functional coverage). We then build a prototype: (i) a tool that supports the CoSTest framework, so that researchers can validate our treatment by conducting experiments to answer RQ3.1 and RQ3.2.

For the tasks related to the investigation (i.e. treatment validation), we followed the *empirical research cycle*, which has the structure of a rational decision cycle, just like the engineering cycle: research problem analysis (T1'), research and inference design (T2'), validation (T3'), research execution (T4') and data analysis (T5'). Then, our proposed framework was validated by (i) a comparative experiment of the results obtained in the two first model-to-model transformations in order to evaluate their syntactic and semantic correctness, (ii) laboratory experiments to validate the different methods used in CoSTest as well as to validate the effectiveness of the CoSTest test cases, (iii) interviewing some IT practitioners to ask their opinion about the usefulness and ease-of-use of the CoSTest tool. Several empirical cycles were thus included in this thesis in order to validate the different parts of the treatment.

For each validation task different protocols were applied according to the subjects, knowledge questions and goals of the study. For example, for the validation of CoSTest feasibility we decided to perform laboratory experiments with a mutation analysis. Figures 2.3-2.5 show some tasks of the empirical cycles applied to different parts of CoSTest.

As the engineering and design cycle do not prescribe a rigid sequence of activities [15] we conceived a *system engineering* execution sequence for CoSTest in which the activities are iterated and may even be performed simultaneously for different aspects of the problem and for alternative treatments. After each iteration a decision is made to stop or to go ahead with the next iteration. Throughout the engineering cycle each iteration uses knowledge about the problem and treatment generated by the previous ones.

Since the work involved in this thesis was carried out over six years, we do not describe all the iterations performed on the design and empirical cycle tasks; instead we present below a list of the tasks and

some research methods (RM) used in each phase of the proposed regulative cycle.

T1. PROBLEM INVESTIGATION

- Investigate the stakeholders and the need for the validation approach for conceptual schemas in a Model-driven environment. RM: literature review and conceptual analysis
- Define the goals and the criteria to judge the proposed treatment. RM: conceptual analysis.
- Define conceptual framework of the new solution. RM: literature review, conceptual analysis.
 - Identify testing techniques and type of defects at conceptual schema level. RM: literature review, application of the defect classification scheme, conceptual analysis.
 - Identify the most relevant quality properties which need to be considered for conceptual schemas built in a Model-driven environment. RM: literature review.
 - Identify the characteristics and resources needed to support the goals. RM: literature review and conceptual analysis

T2. TREATMENT DESIGN

- Specify requirements for new treatment. RM: conceptual analysis.
- Investigate available treatments. RM: literature review.
- Design a new validation framework for conceptual schemas integrated into a Model-driven environment.
 - Design a process to derive an executable conceptual schema under test.

T1'. PROBLEM INVESTIGATION

- Define the goals and the criteria to judge the proposed treatment. RM: conceptual analysis
- Define the conceptual framework to the new treatment. RM: literature review, conceptual analysis.
 - Identify an executable language for UML-based conceptual schemas
 - Identify the transformation rules between elements of the conceptual schema and the language.
 - Identify the characteristics and resources needed to support the derivation of the executable CS.

T2'. TREATMENT DESIGN

- Specify requirements for new treatment. RM: conceptual analysis.
- Investigate available treatments. RM: literature review.

- Design an approach for generating an executable conceptual schema integrated into a Model-driven environment.

T3'. TREATMENT VALIDATION

- Validate analytically the derivation strategy of an executable CS by providing several CS under certain conditions.

- Design the process to generate the test cases.

T1'. PROBLEM INVESTIGATION

- Define the goals and the criteria to judge the proposed treatment. RM: conceptual analysis.
- Define the conceptual framework to the new treatment. RM: literature review, conceptual analysis.
 - Identify the entities of the test model and test scenario model.
 - Identify the transformation rules between entities of the models.
 - Identify the characteristics and resources needed to support the model generation.

T2'. TREATMENT DESIGN

- Specify requirements for the model-driven generation (new treatment). RM: conceptual analysis.
- Investigate available treatments. RM: literature review.
- Design an approach for generating the test model from requirements and then the test scenario model from test model into a Model-driven environment.

T3'. TREATMENT VALIDATION

- Validate analytically the model-driven generation strategy (metamodels and transformation rules) by providing requirements and CS.

T1''. PROBLEM ANALYSIS

- Establish the conceptual framework, goal, experiment research questions and population.

T2''. RESEARCH & INFERENCE DESIGN

- Define the experimental context, objects of study, variables and metrics
- Define hypotheses and to design the experiment.

T3''. VALIDATION

- Evaluate the experimental design
- Review the instruments for collecting data
- Analyse threats on the experiment validity

T4''. RESEARCH EXECUTION

- Collect data: constructs for both models (test model and test scenario)

T5''. DATA ANALYSIS

- Descriptions of the results
- Application of statistical tests and corroboration of hypotheses
- Explanations and generalizations.
- Answer to the knowledge questions.

- Design a method to prioritize test cases

T1'. PROBLEM INVESTIGATION

- Define the goals and the criteria to judge the proposed treatment. RM: conceptual analysis
- Define the conceptual framework to the new treatment. RM: literature review, conceptual analysis.
 - Identify the defects that occur on UML CD-based CS.
 - Identify the class diagram element features that can be mutated of an UML CD-based CS.
 - Identify the restrictions and resources needed to support the CS mutation

T2'. TREATMENT DESIGN

- Specify requirements for new treatment. RM: conceptual analysis.
- Investigate available treatments. RM: literature review.
- Define the mutation operators for CS. RM: literature review and conceptual analysis

T3'. TREATMENT VALIDATION

- Validate some properties of the mutation operators for conceptual schemas.

T1''. RESEARCH PROBLEM ANALYSIS

- Establish the conceptual framework.

T2''. RESEARCH & INFERENCE DESIGN

- Define the context, objects of study, measures and procedure
- Design an artefact (tool) to automate the generation of CS mutants

T3''. VALIDATION

- Analyse threats on the experiment validity

T4''. RESEARCH EXECUTION

- Collect data: contribution factor, mutation score, impact indicator for each mutation operator

T5''. DATA ANALYSIS

- Description and discussion of results

- Define the testing strategy.

T1'. PROBLEM INVESTIGATION

- Define the goals and the criteria to judge the proposed treatment. RM: conceptual analysis
- Define the conceptual framework to the new treatment. RM: literature review, conceptual analysis.
 - Identify the commands to run test cases against the CS
 - Identify the faults generated in Alf
 - Relate the faults generated by Alf with the defect to be reported.

- Relate the faults generated by Alf with the defect to be reported.
- Identify the characteristics and resources needed to support the testing process. RM: literature review and conceptual analysis

T2'. TREATMENT DESIGN

- Specify requirements for new treatment. RM: conceptual analysis.
- Investigate available treatments. RM: literature review.
- Design an approach for executing the test cases against the CS and detect the defects using Alf.

T3'. TREATMENT VALIDATION

- Validate the testing process on executable CS mutants. RM: laboratory experiments using mutation testing.

T1'. PROBLEM INVESTIGATION

- Define the goals and the criteria to judge the proposed treatment. RM: conceptual analysis
- Define the conceptual framework to the new treatment. RM: literature review, conceptual analysis.
 - Identify the commands to run test cases against the CS
 - Identify the faults generated in Alf
 - Relate the faults generated by Alf with the defect to be reported.
 - Identify the characteristics and resources needed to support the testing process. RM: literature review and conceptual analysis

T2'. TREATMENT DESIGN

- Specify requirements for new treatment. RM: conceptual analysis.
- Investigate available treatments. RM: literature review.
- Design an approach for executing the test cases against the CS and detect the defects using Alf.

T3'. TREATMENT VALIDATION

- Validate the testing process on executable CS mutants. RM: laboratory experiments using mutation testing.

- Define the results visualization for feedback and reports. RM: conceptual analysis and defect classification schema.
- Specify usage guidelines for applying the proposed validation approach. RM: conceptual analysis.

T3. TREATMENT VALIDATION

- Validate the effectiveness of CoSTest by means of a comparative experiment.

T1'. PROBLEM ANALYSIS

-Establish the conceptual framework, goal, experiment research questions and population.

T2'. RESEARCH & INFERENCE DESIGN

- Define the experimental context, objects of study, variables and metrics
- Define hypothesis.
- Design the experiment.

T3'. VALIDATION

- Evaluate the experimental design
- Review the instruments for collecting data
- Analyse threats on the experiment validity

T4'. RESEARCH EXECUTION

- Collect data: rate of fault detection, rate of fault type detection, mutation score.

T5'. DATA ANALYSIS

- Description of the results
- Application of statistical tests
- Explanations and generalizations
- Answer to knowledge questions.
- Discussion of the results

- Evaluate CoSTest user perceptions by means of interviewing potential stakeholders and using an observational case study.

T1'. PROBLEM ANALYSIS

-Establish the conceptual framework, goal, experiment research questions and population.

T2'. RESEARCH & INFERENCE DESIGN

- Define the experimental context, objects of study, variables and metrics
- Define hypothesis.
- Design the experiment.

T3'. VALIDATION

- Evaluate the experimental design using a pilot test
- Review the instruments for collecting data
- Analyse threats on the experiment validity

T4'. RESEARCH EXECUTION

- Collect data: practitioners' perceptions

T5'. DATA ANALYSIS

- Answer knowledge questions.

2.5 Summary

Since this PhD thesis was conceived as an exploratory research project, this chapter summarizes the methodology followed in this

work, which has been taken from Wieringa's design science [15]. The different motivations and goals were then identified according to the stakeholders and potential end-users. Also, the goal structure and related research questions were presented to derive the different tasks of regulative cycles (i.e. design and empirical) to overcome a design problem and knowledge problem. A brief description is given of the different tasks in the design and empirical cycles.

In the next chapter, we will discuss related work on the validation of conceptual schemas from different standard, industrial and academy view points.

PART II.

PROBLEM

INVESTIGATION

Chapter 3

THEORETICAL FRAMEWORK

In the context of MDD, where conceptual schemas (models) are the basis of the whole development process, the quality of the CSs has a high impact on the final quality of the software systems derived from them [20]. Hence, CSs may directly affect both the efficiency (time, cost, effort) and the effectiveness (quality of the results) of information systems development.

Conceptual Schemas are developed using a modelling language. The de-facto standard for analysis and design of object-oriented software systems is the Unified Modelling Language (UML) [16], which is extended with OCL (Object Constraint Language) constraints [17]. The variety of UML diagrams provide flexibility and applicability to modellers to create CSs in the different spaces where they can be used (problem, solution and background) [18]. However, since the modelling process is a human task, it is difficult to avoid introducing defects into the CSs (e.g. inconsistency, incorrect, redundant and imprecise elements).

Although defects may be inevitable, we should minimize their number and impact on software quality through testing and/or inspecting the CS. Testing aims to detect defects in a system by

comparing the expected results (expressed in system requirements) to the observed results (the behaviour of the implementation of the System Under Test (SUT)). In many organizations testing processes begin after the code has been completed [19]. In order to detect defects before they become extremely expensive to fix and manage inevitable changes during software lifecycle, testing activities should start as soon as possible (the requirements level) in the software lifecycle and the Information on the defect types that occur in the earlier stages of the software development life cycle can be used to give feedback to stakeholders (e.g. modellers, developers, testers) about detecting defects and how they can be tracked, reduced and resolved. If the purpose is to get a good quality CS, the information on each defect must be related to the quality goals affected, according to an appropriate quality model for models in an MDD context, as proposed in [9].

The purpose of this chapter is to provide the basic knowledge required to understand the overall thesis. As shown in Figure 3.1, this work is placed in the intersection of three research areas that have some aspects in common. These disciplines are: Requirements Engineering, Software Quality (focused on Conceptual Schema Quality), and Model-driven Development.

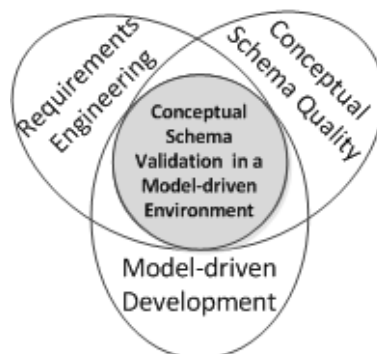


Figure 3.1. Research areas involved in this work

The rest of this chapter is organized as follows: Section 3.1 briefly describes the modelling requirements based on Communicational

Analysis used in this thesis. Section 3.2 describes the concepts of the Conceptual Schema Quality, such as the quality model for conceptual schemas taken as reference to our work and the testing terminology and the testing artefacts involved in our proposal. Section 3.3 summarizes the model-driven development concepts related to this research, and Section 3.4 summarizes and presents the conclusions of the chapter.

3.1 Concepts of Requirements Engineering

A general definition defines Requirements Engineering as a particular research discipline in the fields of software engineering. Following this definition, the discipline searches for, defines, and provides new techniques, instruments, and methods to support the requirements document process of a software system.

A more specific definition states that requirements engineering is the first phase of the software engineering life cycle, which is responsible for a systematic development of a requirements document that describes what, a system shall do. Loucopoulos and Karakostas define Requirements Engineering as “a systematic process of developing requirements through an iterative co-operative process of analyzing the problem, documenting the resulting observations in a variety of representation formats, and checking the accuracy of the understanding gained” [20].

The second definition of requirements engineering is assumed in this research, by which requirements engineering consists basically of the elicitation, analysis, documentation (specification), verification and validation. During elicitation, the requirements are elicited from all possible sources, e.g. from input documents or through interviews with the customer and users. The output of this first activity is the raw requirements. These are analyzed in the second step for consistency, feasibility, incompleteness, and ambiguity. If problems are detected in the analysis activity, the problems must be re-negotiated among the stakeholders until all the stakeholders agree upon the set of

requirements. Then, in the third step these requirements are documented at an appropriate level of detail (e.g. requirements model), and are integrated into the requirements document in the fourth activity. The requirements document is then validated with respect to correctness and completeness of the requirements to the customer and user needs. The validated requirements then serve as the major input for the system development and the acceptance as well as for the test case generation as they are required in our proposal.

Several techniques can be used to specify the requirements in the system requirements document. The most popular techniques to specify the user requirements are natural language and use cases. Even though natural language presents several disadvantages such as ambiguities, unclearness, and redundancies, it is the most frequently used technique to describe requirements in industry. On the other hand, the use cases focus on the more structured description of the interaction between the different users and the system by using a graphical and textual representation, however, they are also in form of natural language and also present the above disadvantages.

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 describe in Section 3.2.2, some V&V techniques require semi-formal or formal models to be applied, and models are often used for specification purposes and as a base for design and implementation (including automatic generation of code in MDD context). In our research, we need to capture the functional requirements in a clear and concise manner, which is typically not possible with natural language. We therefore use a Requirements Engineering method called Communication Analysis (CA) to specify requirements models, which minimizes the disadvantages of writing the requirements in natural language.

3.1.1 Modelling Requirements based on Communicational Analysis

Communication Analysis is a Requirements Engineering method that analyses the communicative interactions between company's Information Systems (CIS) and their environment [21].

The methodological core of Communication Analysis is the information system analysis stage, the result of which is an analysis specification, a communication-oriented documentation that describes the information system. For this purpose, CA proposes a requirements structure with five levels (see Figure 3.2):

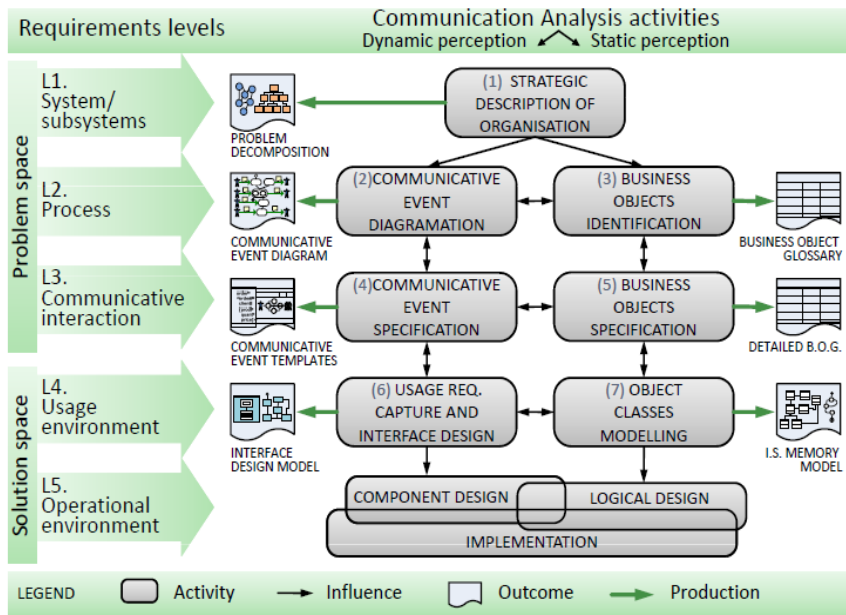


Figure 3.2. Communication Analysis requirements levels and workflow [21]

(i) System/subsystems level (L1) refers to an overall description of the organisation and its environment (Organisational System and Subject System, respectively) and also involves decomposing the problem in order to reduce its complexity, (ii) Process level (L2) refers to business process description both from the dynamic viewpoint (by

identifying flows of communicative interactions, a.k.a. communicative events) and the static viewpoint (by identifying business objects), (iii) Communicative interaction level (L3) refers to the detailed description of each communicative event (e.g. the description of its associated message) and each business object, (iv) Usage environment level (L4) refers to capturing requirements related to the usage of the Computer Information System (CIS), the design of user interfaces, and the modelling of object classes that will support IS memory, and (v) Operational environment level (L5) refers to the design and implementation of CIS software components and architecture (further information can be obtained in [21]).

3.2 Concepts of the Conceptual Schema Quality

The meaning of quality has been widely discussed and everybody agrees that quality is an important property of products. ISO/IEC 9126 [22] (an international standard for the evaluation of software quality consistent with ISO 9000 [23], a family of standards related to quality management) define the quality of a software as: “The totality of features and characteristics of a product or service that bear on its ability to satisfy stated or implied needs”. Most approaches to quality evaluation therefore decompose the concept of quality into a set of lower level quality properties (also called “goals”, “attributes” or quality characteristics) which may be precisely measured.

In the context of modelling, the quality of a CS or model is the degree to which a set of model quality properties is present. Therefore, the set of quality goals with their relations, accompanied by a set of practices or means to achieve the quality goals and evaluation methods for evaluating quality goals define a Quality Model [9].

This section describes the quality model taken as reference for this research project as well as the basic concepts and testing artefacts used in the testing-based validation process of the conceptual schemas.

3.2.1 Model Quality for Conceptual Schemas

Different quality models can be found in the literature for describing the quality of CSs such as that developed by Lindland, Sindre, and Sjølvberg [16] (1994), and its numerous extensions and refinements (e.g. Krogstie and Sjølvberg [17], 2003; Krogstie et al. [18], 2006; and Krogstie [19], 2012). This quality framework classifies model quality into three categories (i) the syntax quality (relationship between the model and modelling language norms, i.e. syntax), (ii) semantic quality (relationship between the model and problem domain); and, (iii) pragmatic quality (comprehensibility by the stakeholders). Krogstie [19] added some quality goals for the understanding and assessment of models quality to the Lindland's framework; such as physical quality (model is persistent, current and available), empirical quality (model has features visual or textual communication to help minimal error frequency), social quality (relationship agreements between different model interpretations) and deontic quality (if the model meets the objectives of modelling). Other quality models such as those found in [3] and [20] also discuss the concept of model quality within the context of UML.

However, the MDD approach allows many activities to be automated in software development. Conceptual schemas in MDD are expected to get progressively more complete, precise and executable and be used to generate the code and other artefacts such as test cases. Therefore, MDD add new requirements to the development process such as consistency between models, technical comprehension by tools and support changeability. In [21] Mohagheghi et al. describe a quality model (6C) oriented to Model Driven Engineering (MDE). They perform a combination of quality models and identify the following six classes of conceptual schema quality goals (see Table 3.1).

For the purpose of conceptualizing the quality properties considered in this thesis, we adopted the quality model proposed by Mohagheghi et al. [9].

Table 3.1 Quality Goals based on 6C quality model from Mohaghehi et al. [9]

Quality Goal (QG)	Description
Correctness QG1	Including correct elements and relations between them, and including correct statements about the domain; not violating rules and conventions; for example adhering to language syntax. Thus it covers both syntactic correctness (right syntax or well-formedness) and semantic correctness (right meaning and relations relative to the knowledge about the domain).
Completeness QG2	Having all the necessary information that is relevant and being detailed enough according to the purpose of modelling. It is a semantic quality.
Consistency QG3	Having no contradictions in the models, related to syntactic quality. It covers consistency between views that belong to the same level of abstraction or development phase (horizontal consistency), and between views that model the same aspect, but at different levels of abstraction or in different development phases (vertical consistency). It also covers semantic consistency between models; i.e, the same element does not have multiple meanings in different diagrams or models.
Comprehensibility QG4	Being understandable by the intended users, either human users or tools. It is related with the pragmatic quality.
Confinement QG5	Being in agreement with the purpose of modelling and the type of system, and being restricted to the modelling goals; such as including relevant diagrams and being at the right abstraction level. It is related with the semantic quality.
Changeability QG6	Supporting changes or improvements so that models can be changed or revolved rapidly and continuously. It is related with the pragmatic quality.

3.2.2 Practices to improve the Quality of Conceptual Schemas

In order to assess whether a CS meets the above quality goals, several methods can be employed. All these methods aim to validate and verify (V&V) the CS according to a quality model.

The IEEE [24] defines validation as the “confirmation by examination and provisions of objective evidence that the particular requirements for a specific intended use are fulfilled”. On the other hand, the same standard defines verification as the “confirmation by

examination and provisions of objective evidence that specified requirements have been fulfilled”.

Applying the above definitions in the modelling context, validation is an activity that answers the question: “Are we developing the right model?”, i.e. whether all the knowledge in the model is sufficiently correct and relevant to the problem domain. On the other hand, verification is an activity that answers the question “Are we developing the model right?”, i.e. whether the model satisfies quality properties such as consistency. According to the ISO/IEC 9126 [25] classification [25], validation aims to check the external quality and verification aims to check the internal quality.

The contributions of this thesis are aimed at enhancing conceptual schema validation in the context of a model-driven development. The application of techniques aimed at validating requirements may depend on the formalization level of the requirements specifications.

The methods applicable to validation are suitable for software validation in general, and for models validation (e.g. requirements models, design models, test model, etc.) in particular.

In this thesis, we classify methods from two perspectives: (1) the way in which the analysis is performed; and (2) the level of formalization. This classification is an oversimplification for the purpose of this thesis.

First, regarding the way in which the analysis is performed, we classify methods into two categories:

Static methods. Static methods examine a model and reason over all the possible behaviours that might arise at run time [26]. It means that the model is read by humans, or pursued by a computer, but not executed as a program. Hence, static methods work at “compile time”.

Dynamic methods. Dynamic methods operate by executing a program (in our case, a CS or model) and observing its executions [27].

It means that the model is run (or executed) by means of a computer. Hence, dynamic methods work at “run time”.

Second, regarding their level of formalization, we classify methods into two categories:

Formal methods. Wing [28] describes formal methods as “mathematically based techniques for describing system properties. Such formal methods provide frameworks within which people can specify, develop, verify and validate systems in a systematic, rather than ad-hoc manner”.

Non-formal methods. Unlike formal methods, non-formal methods do not try to follow a rigorous approach but to use informal techniques. Non-formal methods have the advantage that the user does not need be an expert in understanding mathematical models. They are easy to illustrate and can be used to validate models written in natural language, increasing the participation of non-technical stakeholders. As a drawback, given their non-formality, they can be ambiguous and provide a non-precise result.

In the following, we briefly review some of the existing methods, classifying them into the above categories (see Table 3.2). Note that, again, our classification is an oversimplification which only includes a subset of the many existing methods devoted to validation.

Table 3.2. Validation relevant methods for Conceptual Schemas

	Static Methods	Dynamic Methods
Non-formal methods	Reviews Inspections	
Formal methods		Testing Simulation and Animation

Review

The IEEE [29] standard defines a review as “a process or meeting during which a software product is presented to project personnel, managers, users, customers, user representatives, or other interested

parties for comment or approval". The IEEE [29] also defines a technical review as "a systematic evaluation of a software product by a team of qualified personnel that examines the suitability of the software product for its intended use and identifies discrepancies from specifications and standards. Technical reviews may also provide recommendations of alternatives and examination of various alternatives". The purpose of a technical review is to achieve at a technically superior version of the software product reviewed, whether by correction of defects or by recommendation or introduction of alternative approaches.

Inspection

The IEEE [29] standard defines an inspection as "a visual examination of a software product to detect and identify software anomalies, including errors and deviations from standards and specifications. Inspections are peer examinations led by impartial facilitators who are trained in inspection techniques. Determination of remedial or investigative action for an anomaly is a mandatory element of a software inspection, although the solution should not be determined in the inspection meeting". Compared to the technical reviews and walkthroughs, inspections are more structured. The IEEE standard [29] states that inspections should be done according to the project plan.

The above techniques have mainly been applied to analyse source code [30]. However, these techniques can also be applied in earlier phases of software development such as requirements specification [31] or design [32] [33].

Paraphrasing [34] and ***Explanation Generation*** [35] techniques are attempts to verbalize and provide explanations about the behaviour of conceptual schemas in order to facilitate their comprehension and validation, supporting the conceptual modelling activity.

Simulation and Animation

According to Bicarregui J. et al. [36], 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. The method we propose to test conceptual schemas belongs to this category of validation techniques. Formal requirements specifications (like conceptual schemas defined in a formal modelling language) can be validated by using these techniques that execute them through animation (e.g. [37], [38]).

Testing

Testing is probably the most popular method used for the dynamic verification and validation of a software artefact and is done by running a discrete set of test cases, where a test case consists of input values and their expected output. The test cases are suitably selected from a finite but very large input domain. During testing the actual behaviour is compared with the intended or expected behaviour. The emphasis of software testing is to validate and to verify the design and the initial construction.

Testing could be categorized as functional and non-functional testing. Functional testing is concerned with what the software artefact does its features or functions. Non-functional testing is concerned with examining how well the software artefact does its job and includes performance, usability, portability, maintainability, etc. However, testing is an expensive practice to improve the quality of CS and requires stop criteria because a complete testing is infeasible [12].

3.3 Concepts of the Model-driven Environment

As mentioned before, this research focuses on the design of a testing based validation framework that satisfies a model-driven environment in order to improve conceptual schema quality. This section provides the reader with the lexicon and tools used throughout model-driven testing. First, we introduce MDA definitions and assumptions as well as the concepts of the metamodeling architecture

used in our work. Then, we summarize the concepts of the UML CD-based Conceptual Schemas and also of an executable UML CS. Finally, we summarize the concepts of the OMG standards for specifying executable models.

3.3.1 MDA Definitions and Assumptions

The Object Management Group (OMG) has defined its own proposal for applying MDE practices to system' development, which is called MDA (Model-Driven Architecture). The entire MDA infrastructure is based on few core definitions and assumptions. The main elements of interest for MDA are the following [39]:

- A **System** is the subject of any MDA specification. It can be a program, a single computer system, some combination of parts of different systems, or a federation of systems.
- **Problem Space** (or domain) is the context where the system operates.
- **Solution Space** is the spectrum of possible solutions that satisfy the system requirements.
- **Architecture** is the specification of the parts and connectors of the system and the rules for the interactions of the parts using the connectors.
- **Platform** is a set of subsystems and technologies that provide a coherent set of functionalities oriented towards the achievement of a specified goal.
- **Viewpoint** is a description of a system that focuses on one or more particular concerns.
- **View** is a model of system seen under a specific viewpoint.
- **Metamodel** constitutes the definition of a modeling language, which provides a way of describing the whole class of models that can be represented by that language. Therefore, we can define models of the reality, and then models that describe models (called metamodels) and recursively models that describe metamodels (called **meta-metamodels**). Then, a model *conforms* a metamodel in the way that a computer

program conforms to the grammar of the programming language in which it is written [39].

- **Transformation** is a correspondence relation between elements in a source metamodel and elements in a target metamodel. It is defined at metamodel level, and then applied at the model level, upon models that conforms to those metamodels. Therefore, executing a Model-to-Model (M2M) transformation transforms a source model M_a conforming to a metamodel MM_a into a target model M_b conforming to a metamodel MM_b (where MM_a and MM_b can be the same or different metamodels).

3.3.2 Overview of the Metamodeling Architecture

Since our proposal complies with the principles of Model-Driven Architecture, it distinguishes different types of models at various levels of abstraction, as follows [39]:

Computation-Independent Model (CIM) is the most abstract modelling level and represents the requirements of the solution without any binding to computational implications.

Platform-Independent Model (PIM) is the level that describes the behaviour and structure of the system, regardless of the implementation platform.

Platform-Specific Model (PSM) contains all the required information regarding the behaviour and structure of an application on a specific platform that developers may use to implement the executable code.

A set of mappings between each level and the subsequent one can be defined through model transformations. Typically, every CIM can map to different PIMs, which in turn can map to different PSMs.

In this thesis, UML (Unified Model Language) [40] class diagrams define the metamodels presented in Chapter 6, while ATL (ATLAS

Transformation Language) [41] defines the model transformations. We select both solutions, as they are well-known languages. UML is proposed by OMG (Object Management Group) and is frequently used in MDE for defining metamodels. ATL is one of the most popular and widely used model transformation languages [41]. ATL is a hybrid transformation language that contains a mixture of declarative and imperative constructs. Helpers and transformation rules are the constructs used to specify the transformation functionality.

3.3.3 UML CD-based Conceptual Schemas

The aim of this work is to design test cases to find faults in a Conceptual Schema during the analysis and design of the software by deliberately changing a UML CD-based CS, resulting in wrong behaviour and possibly causing a failure. The CS of a system should describe its structure and behaviour (constraints). In this paper a UML-based class diagram is used to represent such a CS.

A class diagram (see Figure 3.3) is the UML's main building block that shows elements of the system at an abstract level (e.g. class, association class), their properties (ownedAttribute), relationships (e.g. association and generalization) and operations.

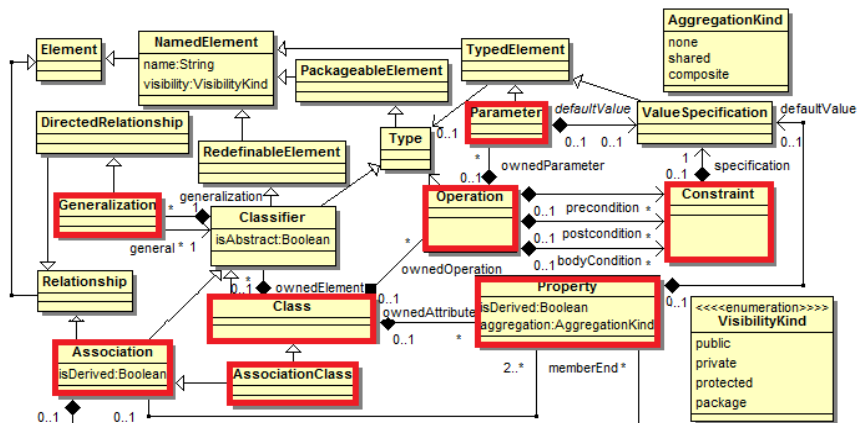


Figure 3.3. Excerpt of the Metamodel of an UML Class Diagram [40]

In UML an operation is specified by defining pre- and post-conditions. Figure 3.3 shows an excerpt of the UML structure for a class diagram and highlights eight elements of interest for this work.

3.3.4 Executable UML Conceptual Schema Under Test

If we want to dynamically test models to detect potential misconceptions expressed in it, we need to be able to execute the models. An executable model is a model with a structure (what is it?) and behavioural specification (what does it do?) detailed enough to be systematically executed in a production environment.

Structural model

The structural model specifies the static part of an information system [1], which is formed by a set of classes, a set of attributes of each class, a set of associations among classes, a set of generalizations among classes and a set of integrity constraints (i.e. conditions that must be satisfied in all states of an information system).

All elements in the class diagram are assumed to be correct instances of the corresponding metaclasses of the UML metamodel [40].

Some integrity constraints (mainly cardinalities) may be graphically represented in the CD, while the rest of them may be textually specified in OCL [17]. Figure 3.4 shows an excerpt of structural model of our Video Club CS.

Behavioural model

The behavioural model specifies the dynamic part of an information system, i.e. the valid changes in the system state, as well as the functions that the system can perform [1]. In UML there are several models to specify the behaviour of a system at a high level of abstraction, for instance, using use case diagrams, activity diagrams, state chart diagrams, etc. However, as we have introduced, in order to be executable, the behavioural models must be detailed enough. For

this reason, in this thesis, in order to define a detailed behavioural model, we use operations. Operations are sequences of atomic steps that users may execute to query and/or modify the information modelled in the structural model. The Operations are attached to UML classes.

Figure 3.4 shows three operations related to the Rental class (i.e. new rental, RENTAL_INFO and set_return_date).

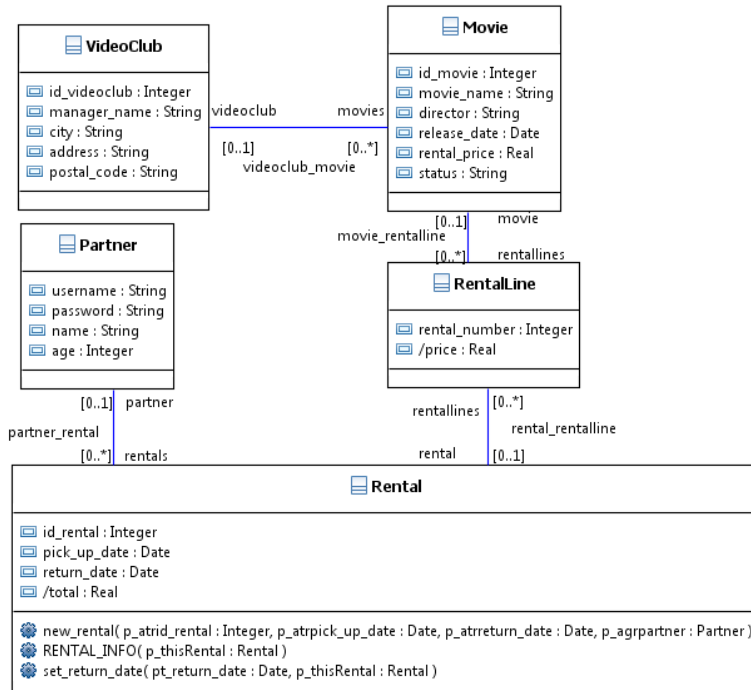


Figure 3.4. Excerpt of UML-CD-based CS for Video Club case

In addition, the pre and post conditions and invariants included in the class diagram are also operations or part of operations (i.e. pre and post condition). For example Figure 3.5 shows some constraints attached to the class Rental of the Video Club CS.

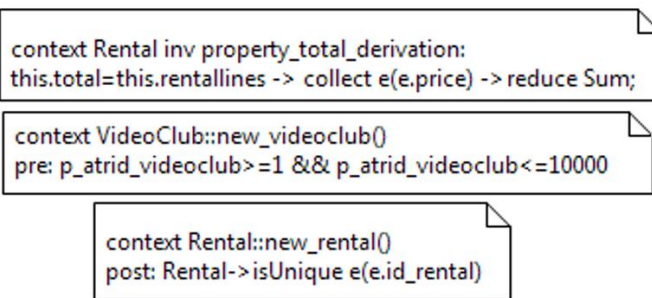


Figure 3.5. Example of constraints for the Video Club system

3.3.5 Defect Types in UML-based Conceptual Schemas

A conceptual schema may not always represent the functionality it is intended for. The causes and consequences of deviations from the expected function in conceptual schema are factors that affect the dependability and quality of a software product. The terminology presented below was adapted from IEEE std. 1044-2009 [42] for executable conceptual schemas.

- **Defect:** An imperfection or deficiency in a work product where that work product does not meet its requirements or specifications and needs to be either repaired or replaced.
- **Fault:** A manifestation of a defect in a conceptual schema.
- **Failure:** An event in which the conceptual schema does not perform a required function within specified limits.

When a defect is encountered during model execution it is called a fault, but it is not a fault if it is detected by inspection or static analysis. Therefore, a fault is a subtype of defect and may cause a failure when it is encountered. We adjusted the description of the scope of the relationships between conceptual entities proposed by the standard IEEE on one hand, with the conceptual entities of our study (UML-based conceptual models) on the other. This resulted in Figure 3.6, where these relationships are depicted graphically. The red frame directly corresponds to the IEEE standard.

The defects at the conceptual level can be located in several ways through V&V techniques, which use a detection mechanism (based on rules, metrics, and modelling conventions) for this purpose.

According to the nature of the technique, this can be statically or dynamically supported by a tool and can have a type of scope that depend on its purpose (i.e. detect, prevent and resolve).

The defects have insertion activity, severity, priority and probability of occurrence. They are detected at any specific time by noticing a specific description (symptom) using a detection mechanism. Each of these aspects is relevant for the purpose of the required analysis and also allows a classification of the defects. In previous work [43] we classified UML model defects reported in the literature and related the types of the defects with the CS quality goals (see Section 3.2) affected by them. Table 3.3 summarizes the CS defect types.

Table 3.3. Defect types in a UML-based model (excerpt taken from [43])

Defect Cause	Sub modes	Affected Quality Goal
MISSING	Something is absent that should be present.	QG2, QG4
WRONG Something is incorrect, inconsistent or ambiguous.	Inconsistent: There are contradictions in the models (1) vertical inconsistency (i.e. contradictions between model versions) and (2) horizontal inconsistency (i.e. contradictions between different model views).	QG1, QG3, QG4, QG5
	Incorrect: There is a misrepresentation of modelling concepts, their attributes and their relationships, as well as the violation of the rules by combining of these concepts at the time of building partial or complete models.	QG1, QG4
	Ambiguous (wrong wording): The representation of a concept in the model is unclear, and could cause a user (e.g. modeller) to misinterpret its meaning.	QG1, QG3
UNNECESSARY (Extra) Something is present that need not be.	Redundant: If an element has the same meaning that other element in the model.	QG5
	Extraneous: If there are items that should not be included in the model because they belong to another level of abstraction, e.g. details of implementation, which are decisions (e.g. type of data structure used at code level) that are left to be made by the developers, and is not specified at an earlier level (e.g. CS).	QG5, QG6

Missing and unnecessary elements (i.e. redundant and extraneous) and incorrectly modelled requirements are the main causes of a design model inaccuracy that can be detected by requirements testing. Inconsistent defects can only be found by comparing CS versions, so that testing is not required in this case. Ambiguous elements require user (e.g. modeller, low-level designer) criteria to find defects.

In this thesis we face the challenge of detecting defects (missing, correctness and unnecessary elements) on conceptual schemas by testing.

3.4 Summary and Conclusions

This thesis aims at enhancing conceptual schema validation in the context of model-driven development. In this chapter we describe the concepts related to three research areas on which our research is based: Requirements Engineering (Section 3.1), Conceptual Schema Quality (Section 3.2) and Model-driven Environment (Section 3.3).

Requirements Engineering and Quality both aim to support the development of software products to meet stakeholder's expectations regarding functionality and quality at different stages of the software development life cycle (e.g. conceptual schema used in both analysis and design phases). We adopted the quality model proposed in Mohagheghi et al. [21] (see Section 3.2) for the purpose of contextualizing the CS quality goals considered. In order to assess whether a Conceptual Schema meets the desired quality goals, several methods can be employed. In this chapter we have reviewed and classified a subset of the most relevant analytical methods used in several fields of computer science, both in hardware and software (mainly in source code) verification and validation:

- Static and non-formal methods: Walkthroughs, reviews and inspections.
- Static and formal methods: Data-Flow Analysis, Constraint-Based Analysis and Abstract Interpretation.
- Dynamic and non-formal methods: Testing.

- Dynamic and formal methods: Model Checking.

The testing of conceptual schemas may be an important and practical means of validation because it allows checking correctness and completeness according to stakeholders' needs and expectations. In conjunction with the automatic checking of basic test adequacy criteria, it can also contribute to improving the consistency, comprehensibility, confinement and changeability of the elements defined in the schema.

As we explain in Section 3.3, the Model-driven theoretical framework is indeed a vital base on which the testing-based framework for validation of UML CD-based conceptual schemas is built.

The theoretical framework for Communication Analysis [21] (a Requirements Engineering method) (see Section 3.1.1) is important for the purpose of modelling the functional requirements of the CS considered in this thesis and also defines the artefacts that are part of the input of our proposal.

Chapter 4

RELATED WORK OF CONCEPTUAL SCHEMA VALIDATION

In software engineering the requirements are usually elicited and specified before implementing them. Requirements can be specified in different kinds of artefacts and in different levels of formalization (i.e. unrestricted natural language, disciplined documentation or formal notation) [44]. The application of techniques aimed at validating requirements may depend on the formalization level of their specification. In particular, conceptual schemas defined in UML are formal specifications of functional requirements and their validation is the main objective of the conceptual schema testing approach proposed in this thesis (Chapter 5).

Validation of software conceptual schemas has been a topic addressed in the literature. The work related to this thesis can be analysed in three dimensions: (1) the domain, i.e. the kind of model to be validated; (2) the type of method employed to perform the validation; and (3) the CS quality goal improved by the validation.

In Chapter 3, we explained the general knowledge related to the problem of conceptual schema validation in a model-driven environment.

In this chapter, we review a representative set of existing techniques on validating conceptual schemas according to the above dimensions. Firstly, Section 4.1 describes the three dimensions and briefly cites the related works. Section 4.2 reviews the most representative works for the generation, selection, prioritization and execution of test cases, which are fundamental challenges addressed by the main contribution of this thesis. Section 4.3 compares the related works and Section 4.4 summarizes and presents the conclusions of the chapter.

4.1 Dimensions of the Related Work

In this section, we analyse how requirements specifications can be validated in conceptual schemas. The related work can be analysed from three perspectives (see Figure 4.1)

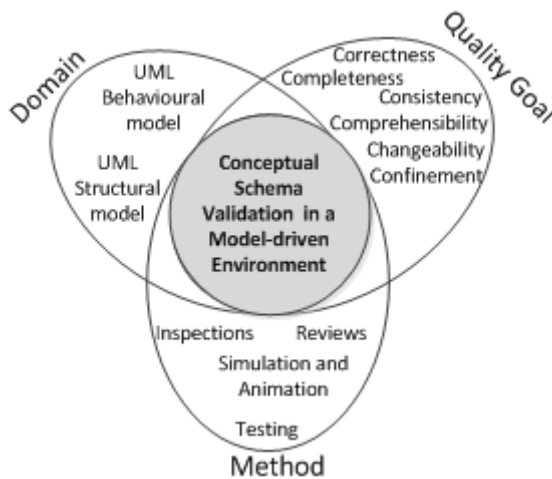


Figure 4.1. Related Work dimensions

Domain. Refers to the kind of model used to perform the validation.

Quality Goal: Refers to the quality goal of the CS to be improved with the validation.

Method: Refers to the type of method employed to perform the validation.

In the rest of this section we briefly describe these dimensions.

4.1.1 Domain

The domain dimension refers to the kind of model to be validated. In the software modelling context, the focus of the validation may be the structural model, the behavioural model or both.

Regarding the first group, only a few works (e.g. [45], [46], [47]) analyse structural models separately from behavioural models. These works are related with static methods such as review and inspections [48].

In the second group there are some research proposals devoted to the problem of validating only behavioural models. For instance, in the UML context, there are works focusing on validating only activity diagrams [49], state machine diagrams [50], [51] and state machine diagram with an activity diagram [52].

The remaining works require both type of models (structural and behavioural) to validate the requirements in the CS, e.g. class diagram including operations and OCLs [53]; class diagram, interaction diagram and activity diagrams [54]; class and sequence [55] [56], and so on.

In Section 4.3 we review in detail the most related works.

4.1.2 Quality Goal

This dimension refers to the quality goal to be improved with the validation.

Several quality goals such as consistency, completeness, comprehensibility and confinement can be assessed by means of

manual human inspections and reviews; as proposed in [57][58][47], and also by using checklists [18]. Both modelling experts and non-technical experts should be involved in inspections; especially for evaluating comprehensibility and confinement aspects. The OORT techniques (Object-Oriented Reading Techniques) are an example of systematic inspection techniques to inspect (“compare”) UML diagrams with each other for completeness and consistency (vertical and horizontal) [50].

On the other hand, testing works (e.g. [53], [54], [55] [56]) mainly aim at the completeness of a CS by validating the requirements. However, semantic correctness, confinement and changeability can be improved by analysing the elements covered and elements not covered (extraneous elements) by test cases (see Tables 3.1 and 3.3 in Sections 3.2.1 and 3.3.5 respectively).

Since information from separate diagrams (i.e. structural and behavioural) should be combined for the purpose of testing, the consistency between these diagrams should be addressed previous to the testing process. Thus, if the testing process is supported by a tool for CS execution, then the incorrect defects are detected by the parser in a previous step to testing, so that the syntactic correctness goal is also improved.

CS comprehensibility by both humans and tools is addressed when the completeness, consistency and correctness of a CS is improved (see Tables 3.1 and 3.3 in Sections 3.2.1 and 3.3.5, respectively).

4.1.3 Method

The method dimension refers to the type of method employed to perform the validation. As we explained in Chapter 3, a variety of methods can be used to analyse a model. They can be classified into static/dynamic and formal/non-formal.

In the following, we review requirements validation techniques that may be applicable to conceptual schemas. Reviews and

Inspections are general techniques that can be applied to other kinds of requirements specifications. Others have been specifically proposed to validate conceptual schemas, such as testing, simulation and animation. Additionally, 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 3.2.2.

Inspections and Reviews

Similar techniques in this respect are the inspections and reviews of requirements specifications [59], [44]. Inspections do not require formal requirements specifications. However, when semi-formal or formal specifications are the requirements artefacts under inspection, the process may be more clear, structured and traceable.

Conceptual schemas specify functional requirements and they can also be inspected and reviewed [50], [60], [46], [48]. However, as requirements validation is hard to judge only by inspecting the models, a model with executable properties is needed to evaluate them and to detect potential misconceptions expressed in the model.

Simulation and Animation

Techniques that execute CSs through simulation and animation [61], [52], [62][63], present facilities for the users to uncover inconsistencies and execute operations of the specification with parameters supplied by users, thereby calculating the value of the output parameters and the new system state. The idea of animating conceptual schemas for validation purposes dates back to the mid-80s. Dignum et al. [64] describe 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 Information Base state, perform consistency checks and ask questions about the contents of the Information Base. A similar approach was taken in [58] and [65], with the PPP and TROLL light language and environment, respectively. [49], [55]

Several tools support editions, simulations and animation of UML models. For instance, the USE tool by Gogolla et al. [55] receives a UML class diagram and a set of declarative operations and is able to validate the structure/behaviour according to the modeller/designer expectations (such as the consistency of UML models and the independence of OCL constraints) through animation. In this context, the authors Dotan and Kirshin [52]; and Teilans et al. [63] present tools providing graphical visualization of simulations on activity diagrams highlighting active states and fireable transitions, coupled with means to visualize and record execution traces.

Testing

In the context of MDD, testing techniques have also been applied for testing models. In these approaches the artefact under test is a model instead of a source code. Some examples are: [56] an approach for testing UML design models to uncover inconsistencies; [61] an Eclipse plug-in for animating and testing UML models; and [66] a method which applies the principles of TDD (Test-Driven Development) to conceptual modelling.

It is important to point out that an important initiative for building executable UML models is the fUML[67], promoted by the OMG (Object Management Group). Research on a model execution framework based on fUML is presented by Mijatov et al. [49]. This framework will enable efficient testing and validating of UML activity diagrams by providing debugging capabilities, as well as a test.

Testing is part of a process of Validation & Verification, where the conceptual schema operates under controlled conditions in order to: (1) verify that it behaves as specified; (2) detect defects, and (3) validate user requirements [12]. A lot of work on automatic verification procedures have been reported in the related literature, such as [68], [69], [70], which are focused on an automatic check of desirable properties in conceptual schemas (e.g. a well-formed instantiation of the model, and consistency between models and with

constraints) and the development of automated reasoning procedures or the semi-automated control of them. As suggested in [27], static and dynamic analysis can interact. In this regard, we believe the dynamic method developed in this thesis can be integrated with static methods, for instance to verify well-formed instantiations of the CS.

Since functional testing is the validation method addressed in this thesis, in the next Section we summarize related work on the generation, selection and prioritization process of the test cases required by the testing process.

4.2 Generation, Selection, Prioritization and Execution of Test Cases

Since testing uses model (sometimes a mental model) [71] as the basis for the construction of test cases, a good set of test cases is directly related to how adequately the model captures the features of the CS under test (CSUT). Nevertheless, designing test cases manually can yield inconsistent test cases even if the model is trust-worthy. Moreover, when the model changes, test cases must be updated and this is not always feasible manually, mainly when the number of tests grow. So that manual generation and execution of tests can be costly and error prone.

In this context, the purpose of model-based testing (MBT) [72] is to use explicit models to automatize testing. Instead of a manual design, tests are generated by a tool that processes the input model and the generated tests can be automatically run against an executable software artefact (e.g. code, executable model).

4.2.1 Test Case Generation

Some methods of test case generation depend on the application, e.g. test case generation for web application, object oriented application, structured systems, UML applications, applications based on evolutionary and genetic algorithms and many others. Throughout

the years, several different methods have been proposed for generating test cases.

At the time of writing this thesis, there are two main surveys (i.e. Escalona et al. [73] and Denger and Mora [74]), that review existing approaches dealing with generating test cases from functional requirements. Escalona's survey, published at the end of 2011, cites 24 approaches; the oldest dates back to 1988 (Category-Partition Method) and the newest to 2009. Denger's, published in 2003, cites 12 approaches; the oldest from 1988 (it is the same approach used in Escalona's survey) and the newest from 2002.

From these surveys, we can see that it is very common in software testing to generate test cases from models (e.g. [75], [76], [77]). However, the artefact under test is a model (i.e. UML CD-based CS). Therefore, works that generate test cases using a strategy that takes the information for tests from another abstraction level used for the early requirements is required, e.g. communicational analysis [21] (communication-oriented business process modelling method), i* [78] (a goal-oriented modelling method) and so on.

Regarding generation strategy, we can see from these surveys that only a recent work [73] introduces a Model-driven testing (MDT) approach, transforming an extended use case pattern (i.e. activity diagram with all paths) to activity diagrams with single paths. These authors propose test cases as an activity diagram to validate a system at code level and not at conceptual schema level, as is required.

In Section 4.3 we will review in detail the generation process of the related works.

4.2.2 Test Case Selection and Prioritization

The selection and prioritization of test cases are the two major solutions to the problem of test case optimization.

Test case selection is a method of selecting a subset of test cases from a test suite to reduce the time, cost and effort of the software testing process. Leon and Podgurski [79] presented an empirical comparison of four different techniques for filtering large test suites: test suite minimization, prioritization by additional coverage, cluster filtering with one-per-cluster sampling, and failure pursuit sampling.

Test case prioritization is a method of scheduling and ranking the test cases from multiple software test suites. There are many approaches to scheduling and ranking the test cases. Each and every test case is assigned a priority, but sometimes an issue may arise when multiple test cases have the same priority or weights. Rothermel et al. [80] define nine techniques (i.e. no prioritization, random, optimal, total branch coverage, additional branch coverage, total statement coverage, additional statement coverage, total fault-exposing-potential and additional fault-exposing-potential) for test suite prioritization for rate of Fault Detection. Note that test case selection and prioritization are closely related. In fact, given a prioritization of test cases, one can filter them simply by choosing the first n tests in the order. Therefore, any test case prioritization algorithm can be used as a test case selection algorithm. However, in general the reverse is not true.

Only Pilskalns et al.'s testing method [56] (see Section 4.1.3) selects the test cases based on variable partitions that can be derived from CS information.

4.2.3 Test Case Execution

The Executable UML approach aims at defining UML models with a behavioural specification precise enough to be effectively executed. In its purest state, an Executable UML eliminates the need for programming the software system. The software models are directly used to run the system through compilation or model interpretation.

There have been model execution tools and environments for years, even before UML. However, each tool defined its own semantics for model execution, often including a proprietary action language, and

models developed in one tool could not be interchanged with or interoperate with models developed in another tool. A Jordi Cabot post [81] describes a list of Executable UML tools; for each tool Cabot provides the name and URL, whether the tool is free, commercial or whatever and if the tool supports the recent Executable UML standards or its own kind of executable UML.

In order to model executable models, whilst the UML specification is necessary, it is not sufficient. This is for two reasons:

1. UML is not specified precisely enough to be executed. Although a UML defines some execution semantics it is not expressive enough to describe each computable function.
2. Graphical modelling notations are not good enough for detailed specifications because this notation tends to be very tedious for exhaustive specifications, confusing the specification rather than enhancing it. Graphical notation is preferred when the diagram is intuitive, but if the diagram is more verbose than a textual representation, then textual is preferred.

In order to overcome these issues, the OMG has extended the UML standard to allow the models to be executable. In particular, two new standards have been recently added to the UML standard: the “Foundational Subset for Executable UML Models” (fUML) [67] and the “Action Language for fUML” (ALF) [82]. In the following we introduce both standards and give examples of their usage.

The Foundational Subset for Executable UML Models (fUML) [67], is an executable subset of the UML that allows the structural and behavioural semantics of systems to be defined in an operational style. In order to precisely specify the behaviour, fUML includes the concept of action. An action is the fundamental unit of behaviour specification. It takes a set of inputs (input pins) and converts them into a set of outputs (output pins), where a pin is a typed and multiplicity element that provides values to actions and accepts result values from them. Some of the actions modify the state of the system in which the action

is executed. However, neither UML 2.X nor fUML provide any concrete textual syntax for actions, but an abstract syntax, which is not really precise.

In order to cover this shortcoming, the OMG proposed in October 2011 the Action Language for Foundational UML (ALF) [83], the first beta version of a concrete syntax conforming to the abstract syntax of the standard fUML. Essentially, ALF is an unambiguous, concise and readable textual language (a kind of pseudocode) that allows designers to completely specify fine-grained behavioural aspects of the model (e.g. to define the behaviour of a method of a class). ALF can be attached to any place with UML behaviour. For instance, ALF sentences can be used directly to specify the behaviours of the transitions on a statechart diagram, the method of an operation or the classifier behaviour of a class.

ALF also provides an extended notation that may be used to specify structural modelling elements. Therefore, it is possible to specify a UML model entirely using ALF, though ALF syntax only directly covers the limited subset of UML structural modelling available in the fUML subset. However, in this thesis we use UML Class diagram to represent the structural part of a CS and then the CS is automatically transformed to ALF. This is because: (1) we believe a graphical notion of the structural model is more intuitive; and (2) neither the fUML subset (nor ALF) allows integrity constraints associated to the class diagram to be defined, an element that the conceptual schema used in this thesis takes into consideration.

Before the adoption of ALF, several action languages emerged such as Object Action Language (OAL) [84], Shlaer-Mellor Action Language (SMALL) [85], Action Specification Language (ASL) [86]. A Jordi Cabot post [87] summarizes Stephen Mellor's quest of more than a decade ago to standardize executable UML tools through OMG standards for precise UML model execution semantics and a UML action language. So that, although there are a number of studies addressing the

verification of UML models that include actions [88] [89] [90] [91], only some of them [92][93] [94] [95] are aligned with the ALF action language standard.

The Papyrus tool [96][97], an open-source UML tool under the Eclipse Modelling Project uses ALF to validate UML models. This tool has executable modelling capabilities including: (1) creating a complete program as a graphical UML class model, with detailed behavioural code written textually using ALF; (2) synchronizing the graphical representation of a UML class with its textual representation in ALF; (3) concurrent execution of an activity and (4) debugging an executing activity. This means a user (modeller/analyst/tester) can manually enter the tests as an activity diagram to perform the testing and debugging process. There is also a work [96] that provides feedback and lessons learned by the Papyrus team regarding the implementation and use of the fUML with ALF from the perspective of domain-specific users.

Research has also been carried out [98][99] on using fUML and ALF as the basis for specifying the semantics of domain-specific modelling languages. However, to the authors' knowledge, there is no possibility of automatically obtaining a full version of the UML model in ALF code from these tools.

This thesis describes the use of ALF for generating executable test cases as well as for translating a UML CD-based CS in an executable model. These ALF-based artefacts are then used within the CoSTest process for validation of UML-based Conceptual Schemas by executing the test cases against the executable CS in an ALF-based testing environment.

4.3 Comparison of Related Works

In this section we compare the validation works related with this thesis based on the three dimensions (i.e. domain, quality goal and method). We also compare the main features of the testing technique

used in these works such as generation, selection and prioritization of the test cases.

4.3.1 Dimension-Based Comparison

A lot of research has been devoted to the problem of V&V (verify and validate) UML models. However, those most closely related to the present work are those which focus on the behaviour defined in structural models (pre and post conditions and invariants related to operations of the classes). Although none of the works addresses exactly the same problem as our focus (i.e. generating test cases for validating CS based on UML class diagram), some research has been done to address similar problems. In this section we review related works that have at least two dimensions (kind of model and validation method) in common with our work.

Table 4.1 classifies the related works that deal with the validation of UML models and positions our work in relation to them. For each approach, we include the following information:

- Work. References of the work.
- Source CS. Indicates the kind of model.
- Language of the Under Test CS. Indicates the kind of CS under test
- Supported Constraints. Indicates whether OCL integrity constraints are considered when analysing the models.
- Technique. Indicates the technique employed during the validation.
- Analysis. Refers to the type of analysis used for validation
- Quality Goal. Enumerates the main quality goals addressed by the work. (see Sections 3.2.1. and 3.3.5)

As can be seen in Table 4.1, a few works target the validation dynamics for UML-CD based CS.

Table 4.1. Related approach comparison

Work	Domain			Method		Quality	
	Source CS	CSUT Language	Supported Constraints	Technique	Analysis	Goal	Defect Type
Dinh-Trong et al. [54]	Class diagram, Sequence diagram, activity diagram	JAL and Java	Yes	Animation and Testing	Automatic	Consistency **Correctness **Confinement	Inconsistency
Gogolla et al.[55]	Class diagram	USE	Yes	Simulation & Animation	Automatic	Consistency **Correctness **Confinement	Inconsistency
Pilskalns et al. [56]	Class diagrams, sequence diagrams	Testable Aggregate Model, USE	Yes	Simulation and Testing	Semi-Automatic	Consistency **Correctness **Confinement	Inconsistency
Tort et al. [53]	Class diagram	CSTL	Yes	Testing	Automatic	**Consistency Correctness Completeness **Comprehensibility **Confinement	Incorrect Missing

* Indirectly addressed by the method (see Table 3.3)

Dinh-Trong et al. [54] present an approach for testing UML models consisting of class diagrams, sequence diagrams, and activity diagrams by simulating the model's behaviour and validating OCL class invariants and pre- and post-conditions of operations. In this approach a test case consists of the definition of the initial objects and links of the system under test and a sequence of operation calls. For executing test cases, Java code is generated from the UML model under test.

The generated code simulates the behaviour of the defined activity diagrams which is specified with their own action language JAL. For evaluating OCL constraints during the simulation, USE is applied.

An interesting tool to validate UML/OCL conceptual is USE presented by Gogolla et al. [55]. The tool requires the classes and operations to be specified that check whether a concrete instantiation given is accepted by the schema and the OCL constraints, but does not invent new instances that could complement the given instantiation to make it valid in case it is not.

Pilskalns et al. [56] present an approach for testing UML models composed of class and sequence diagrams. OCL class invariants and pre-/post-conditions of operations are used to validate the correct behaviour of models. To execute test cases, a UML model is transformed into another format called Testable Aggregate Model (TAM) on which a symbolic execution is applied.

The OCL constraints are validated after the execution of each message defined in the sequence diagrams with USE [55]. When applying the UML evaluation approach, faults and inconsistencies can be revealed throughout the process. Inconsistencies (e.g. class, operation parameters between class diagram and sequence diagram) are revealed via static analysis by combining the behavioural, structural, and constraint information. Two different types of faults can be revealed by application of dynamic testing techniques applied to the aggregate model. The first type of fault can be classified as a path fault (this type of fault often occur because the modeller/designer did not

address all paths associated with a condition), which is found by traversing the TAM. The second type of fault, known as an OCL fault, occurs when states recorded in the execution trace or instance table violate OCL expressions.

The most recent work, and the one most closely related to our framework is the testing approach proposed by Tort and Olivé [53] with their CSTL Processor tool [100]. These artefacts have been used in a method [101] to apply the principles of TDD (Test-Driven Development) to conceptual modelling. CSTL Processor extends the USE core to testing the structural and behavioural schema elements. However, this solution is limited to the testing elements (i.e. test scenarios with test cases, test data and oracles) that are manually entered by the tester using the CSTL language. Thus, the CS should be entered into the CSTL Processor tool by using the USE language, which makes this method unsuitable for a Model-driven environment in which automation is required for these task types. Even though the results of their testing are presented in a tool, they do not provide any kind of feedback to help designers repair defects.

The above approaches have the following weaknesses:

- i. For defining the CS under test (CSUT), these approaches use their own formalisms (i.e. Java, USE, TAM and CSTL), which are different from the standard semantics (e.g. fUML) for executable UML models.
- ii. These works address validation and none addresses the syntactic correctness of the CSs used in the validation.
- iii. Most of these approaches only focus on verifying the consistency between the structural and behavioural models (i.e. OCL constraints) by using action sequences taken from the same CS or defined ad-hoc by the tester to test the CS behaviour. Only Tort et al.'s work [53] addresses the completeness and correctness of the CS. However, as can be seen in Table 3.3. (Section 3.3.5) defects of consistency,

correctness and completeness also affect other quality goals, such as confinement and comprehensibility, and we have added '*' to this information in Table 4.1.

4.3.2 Testing feature Comparison

Testing is one of the time consuming and costly phases in the software development process, so that any advances in software testing methods and tools can reduce the time and cost of software development.

Software testing consists of activities, for instance generation, selection and prioritization of test cases, execution of the test values on the software artefact being tested, and evaluating the test results. In this section we review related work on these issues. Table 4.2 classifies related works that deal with the generation, selection, prioritization and execution of test cases for UML models. For each approach, we include the following information:

- Work. References of the work.
- Test Case Source. Indicates the source information for generation of test cases.
- Test Values Source. Refers to the source for the test values.
- Test Case Generation. Indicates the process kind for generation of test cases.
- Oracle Generation. Indicates the technique employed for generation of the test oracle.
- Test Case Selection. Indicates the selection process of the test cases.
- Test Case Prioritization. Refers to the prioritization of test cases.
- Execution Environment. Indicates the execution environment for testing.
- Repairing Feedback. Indicates whether the approach returns some kind of repairing feedback beyond a simple yes/no answer.

Table 4.2. Testing features comparison

Work	Source		Test Values	Test Cases	Generation		Selection	Prioritization	Execution	
	Test Case	Test Values			Test Oracles	Coverage			Environment	Repairing Feedback?
Dinh-Trong et al. [54]	Modeller, designer or Tester	Modeller or Tester	Manual	Automatic by using OCL constraints	Based on Sequence diagram interactions	Ad-hoc	None	UMLAnt and USE tools	No	
Gogolla et al. [55]	Modeller, designer or Tester	Modeller or Tester	Manual or Semi-automatic using snapshots code	Automatic by using OCL constraints	Based on Tester criteria	Ad-hoc	None	USE tool	No	
Pilska et al. [56]	CS with OCL constraints	Based on variable partitions derived from CS	Manual	Automatic from OCL constraints	Some coverage criteria	Ad-hoc	None	AdaptUML and USE tools	No	
Tort et al. [53]	Modeller, designer or Tester	Modeller or Tester	Manual	Manual Assertions	Based on Tester criteria	Ad-hoc	none	CSTL processor integrated with USE tool	No	

As can be seen in Table 4.2 none of the works targets the generation, selection and prioritization of test cases for UML-CD based CS. Therefore, the described approaches have the following weaknesses:

- i. A significant weakness is that these works (i.e. [53], [54], [55], [56]) use their own CS or modeller/designer criteria to define the test cases, which can derive incomplete and inappropriate test cases.
- ii. Regarding the source of the test values, only Pilskalns et al. [56] generate test values based on variable partitions derived from the CS. In the other cases, the test values are provided by the modeller or Tester, which can cause values not considered in the test cases.
- iii. Regarding test case generation, a weak point of the related work is that most of them deal with the manual and unsystematic derivation of test cases. Only the method proposed by Gogolla et al. [55] generates the test cases semi-automatically using snapshots code. This means it is not possible to state the relative execution order of test cases (i.e. test scenarios) that are expected to be executed.
- iv. One of the limitations of the most of the related work (i.e. [54], [55], [56]) is that they use their OCL constraints as the test oracles, so that these constraints need to be present in the CS. Only Tort et al. [55] use assertions entered manually for modeller/designer/tester enabling the specification of arbitrary test cases that are separated from the UML model, so the assertions could be evaluated for any point in time as well as for time periods of the execution of the CS under test.
- v. Regarding the selection and prioritization of test cases, none of them describes the criteria or the process applied to select or prioritize the test cases. We therefore considered that they do an ad-hoc selection. However, when the testing phase is done using an appropriate selection of test cases, the testing effort is reduced.

- vi. Each work has defined its own environment to execute the test cases based on the USE tool, which requires specifying both the CS and the test cases in their own formalism different from the standard semantics (e.g. fUML) for executable UML models.
- vii. Finally, we would like to highlight that most of the cited methods simply provide a response or failure (showing whether the input test designed to validate a specific CS element is satisfied or not). However, none clearly identifies the source of the problems (i.e. defect type and location) nor assist the modeller/designer to repair them. For instance, when the testing is not satisfied, Tort et al. [53] return a verdict to indicate whether the constraints are satisfied (i.e. pass) or not (fail) and when the base information state is inconsistent (i.e. error). Dinh-Trong et al. [54] report test failures whenever the following situations occur: uninitialized variables in conditions, uninitialized parameters passed in operations calls, non-existent target object of an operation call, pre-and post-conditions evaluate to false. Pilskalns et al. [56] can reveal a set of faults related to inconsistency via static analysis, and two faults types when test cases are executed: path faults (i.e. a path associated with a condition is not included in the TAM) and a OCL faults (i.e. execution trace violate OCL expressions) using the USE tool. Gogolla et al. [55] reports inconsistencies when invariants are contradictory. Thus, none of these works includes goals and test oracles in their test cases to help identify and locate the detected defects, which means an additional effort is required to identify and locate them in the CS.

The state-of-the-art as reviewed in this section indicates that no definitive approach adequately solves the problem of validating conceptual, which means the following challenges must be addressed:

Challenge 1: Test Case Generation. The test cases with oracles and test goals generated using information external to the CS (i.e.

requirements), the measurement of possible automation and the need for a profitable supporting tool using a standard semantic.

Challenge 2: Test input values. The testing framework should allow (based on any coverage criteria) input data for the parameters of the operations under test to be specified or generated in order to test different execution scenarios.

Challenge 3: Allow to select and prioritize the generated test cases.

Challenge 4: Generate Executable Conceptual Schema under test (CSUT). UML CD-base Conceptual Schemas are our CSUTs. However, a transformation into standard executable semantics (e.g. fUML) are required to execute the UML models.

Challenge 5: Testing Environment. The testing framework should allow to execute the test cases against the CS under test and reporting the detected defects in an environment based on a standard executable semantic for UML models.

Challenge 6: State validation. Assertions regarding the runtime state of the tested model, consisting of objects, their feature values, and links, should be possible for any point in time as well as for time periods of the execution of the CS under test.

Challenge 7: Execution order. It should be possible to test the chronological order in which events are executed during the execution of the CS under test. Furthermore, it should be possible to state the relative execution order of test scenarios that are expected to be executed.

Challenge 8: Input / output validation. The testing framework should enable to check whether an input of a test case results in a given output using test oracles.

Challenge 9: Syntactic Correctness and Consistency Verification.

The testing framework should enable checking whether the CS is correctly represented in terms of the syntax of the modelling language, as well as if the structural part is consistent with the specified behaviour (i.e. OCL constraints). The framework should recognize well-formed and ill-formed CS, reporting defects, if any.

Challenge 10: Repairing Feedback. The testing framework should help the modeller (analyst/designer) to locate and correct the defects detected in the CS.

These points are dealt with in the next Chapter.

4.4 Summary and Conclusions

In the related work on conceptual schema validation, several existing methods claim that they perform the validation of a conceptual schema in different degrees through animation, simulation and testing of the behaviour of a Conceptual Schema. However, these methods in fact verify the consistency between the structural part and the behavioural model (i.e. OCL constraints) using tool support. For validation, these methods need to inspect the results to determine defects, because, although some approaches point out the source of the problem, they do not indicate how the modeller/designer can correct the defect. This is because the methods focus on exploring the CS in order to execute or simulate CS states (but not on finding defects).

Test case generation is among the most labour-intensive tasks in software testing and also one that has a strong impact on its effectiveness and efficiency. For these reasons, it has also been one of the most active topics in the research on software testing for several decades, resulting in many different approaches and tools (see Section 4.2). However, these techniques are focussed on the code level and our proposal aims to generate test cases for revealing requirements defects that may be detected in the CS at the conceptual modeling stages (i.e. analysis and design).

Testing-based conceptual schema validation is a research area that admits new methods and techniques, facing challenges such as generation of test cases using information external to the conceptual schemas (i.e. requirements), the measurement of possible automation, selection and prioritization of test cases and the need for a profitable supporting tool using standard semantics, opportune feedback to support the software quality assurance process and facilitate making decisions based on the analysis and interpretation of the results.

PART III.

TREATMENT **D**ESIGN

Chapter 5

VALIDATION FRAMEWORK FOR CONCEPTUAL SCHEMAS

As mentioned in the previous chapters, the aim of this thesis is to provide a set of methods to help modellers (analysts/designers) and testers to improve the quality of conceptual schemas. The methods provided as part of this thesis are organized in a validation framework.

This chapter describes the thesis' main contribution: the testing-based validation framework for Conceptual Schemas in a Model-driven environment that overcomes the challenges specified in Chapter 4 and grouped in two issues: the validation of requirements at an early phase (i.e. conceptual schemas) and the automatic generation of test cases for conceptual schemas. To meet these challenges, our methodological framework advocates the use of Model-Driven Engineering (MDE) techniques, which involve the intense use of models to support the different phases of the proposed framework. We believe that the use of MDE reduces the complexity of the proposed validation approach because it allows modellers and testers to work at a high level of abstraction and also increases automation and reuse. In our approach,

the level of abstraction is raised by allowing modellers to specify requirements using a Requirements Engineering method that provides high-level conceptual constructs.

Automation is increased by means of model transformations, which take the requirements models as input and automatically generate test case implementations which are integrated in a testing environment that executes them against executable conceptual schemas to assist testers during the validation of requirements of conceptual schemas. Reuse is increased by allowing testers to reuse parts of the test models for generating test cases for other types of conceptual schemas (i.e. OO-Method based conceptual schemas). We thus enable the rapid construction of test models and also automate the implementation of test cases via the composition of reusable test model components.

This Chapter is organized into seven sections: Section 5.1 gives an overview of the validation framework followed to generate the test cases, execute them against executable conceptual schemas and report the test results. Section 5.2 describes the Test Analysis phase and Section 5.3 summarizes the Test Design phase. Section 5.4 presents the Test Generation phase. Section 5.5 states Test Prioritization. Section 5.6 describes Test Execution. Section 5.7 details the Test Evaluation phase. Section 5.8 gives an overview of the testing process and Section 5.9 contains the summary and conclusions of this chapter.

5.1 Framework Overview

In this section, before detailing our methodological framework, we provide a general overview.

According to the vision of Weber et al. [102], we understand a framework as a holistic and concise description of concepts and methods relating to a specific domain. In this thesis, as mentioned in Chapter 1, we propose a framework to help modellers, analysts/designers to improve the quality of their conceptual schemas. Our model-driven validation framework provides an execution environment for the automation of test cases (i.e. test scripts) and thus

provides the user with various benefits that help design, generate, select, execute and report the automated test scripts.

5.1.1 Phases of the Methodological Framework

Figure 5.1 graphically depicts the model-driven testing based validation framework proposed in this thesis, which is described in the next sections.

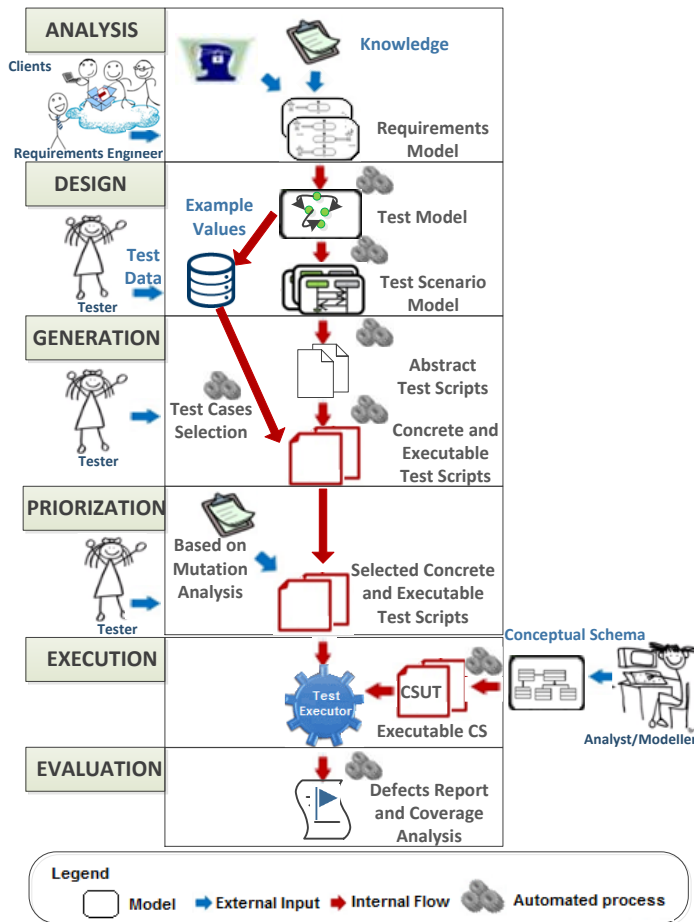


Figure 5.1. Overview of the validation Framework

5.2 Test Analysis

In this Section, we focus on the analysis phase of our validation framework. This phase requires a study of the requirements, talks to

various stake-holders to obtain communicative interactions between the information systems and its environment.

In this phase, the software requirements are understood and modelled in a requirements model using Communication Analysis [21] (see Section 3.1.1). The requirements model is an instance of the Requirements Metamodel proposed by España [103], which describes the system requirements at business level and is specified by the domain experts and system analysts.

5.2.1 Requirements Specification based on Communicational Analysis

Our first reason for using Communication Analysis is to obtain a single model to specify the functionality of an Information System (IS) and to generate the respective test cases. In this way the use of different artefacts by requirements analysts, testers and developers is avoided, thus making their work easier. As the events sequence describes the expected exchanges of messages between the actor and the system, this can be used to define the test cases. In particular, while the communicative events indicate the actions to be performed in a complete and uninterrupted way under certain constraints, the message structures for each communicative event contain references to the types involved that represent actors, or business concepts, the relationships between them and parameterized messages with data types existing in the conceptual schema of the system (the class diagram and state machines). However, this forces the requirements analyst to be precise and rigorous in the semantics given to each CA concept and thus may not be so easy to build. To reduce this complexity, we use the existing editor tool [15], which is a Domain Specific Language to create a Communicative Event Diagram (CED) and introduce a message structure for each communicative event.

Our second reason comes from the fact that requirements-based testing [3], particularly model-driven testing [16], is being increasingly

used. There is thus a need for a systematic approach to generating test cases from requirements model.

Our third reason is in the MDD context, where it is possible to obtain a test model from a requirements model by means of model transformations, so that the process can continue to generate the executable test cases. This means when a modification is made in the requirements model, not only is the test model automatically re-generated, but so are the concrete test cases.

Finally, CA has been integrated into a UML-compliant Model-Driven Development framework [6], as well as a model transformation strategy defined by España et. al [104] to derive the initial versions of conceptual schemas from Communication Analysis requirements models. This means that it includes the primitives (Event Specification Template primitives) that a model-driven method needs (fine-grained enough to be represented directly in code) to express the structure and dynamics of an IS.

5.2.2 Modelling Requirements based on Communicational Analysis

Communication Analysis offers several modelling techniques for business process modelling and requirements specification. The Communicative Event Diagram (CED) describes the business processes from a communicational perspective. Figure 5.2 shows two CEDs of the CA model for the Video Club case (i.e. management of users and movie rents, respectively). A CED consists of a structured sequence defined by precedence relationships among Communicative Events (CE) (the rounded boxes in Figure 5.2). A CE is an action related to information (acquisition, storage, processing, retrieval and / or distribution). A CE is carried out in a complete and uninterrupted way when there is an external stimulus to the system (i.e. user login into the system). A CE can be specialized by means of event variants, which are alternative events that define paths in the CED (e.g. in Figure 5.2 the Login Resolution is specialized into Login is accepted or login is rejected).

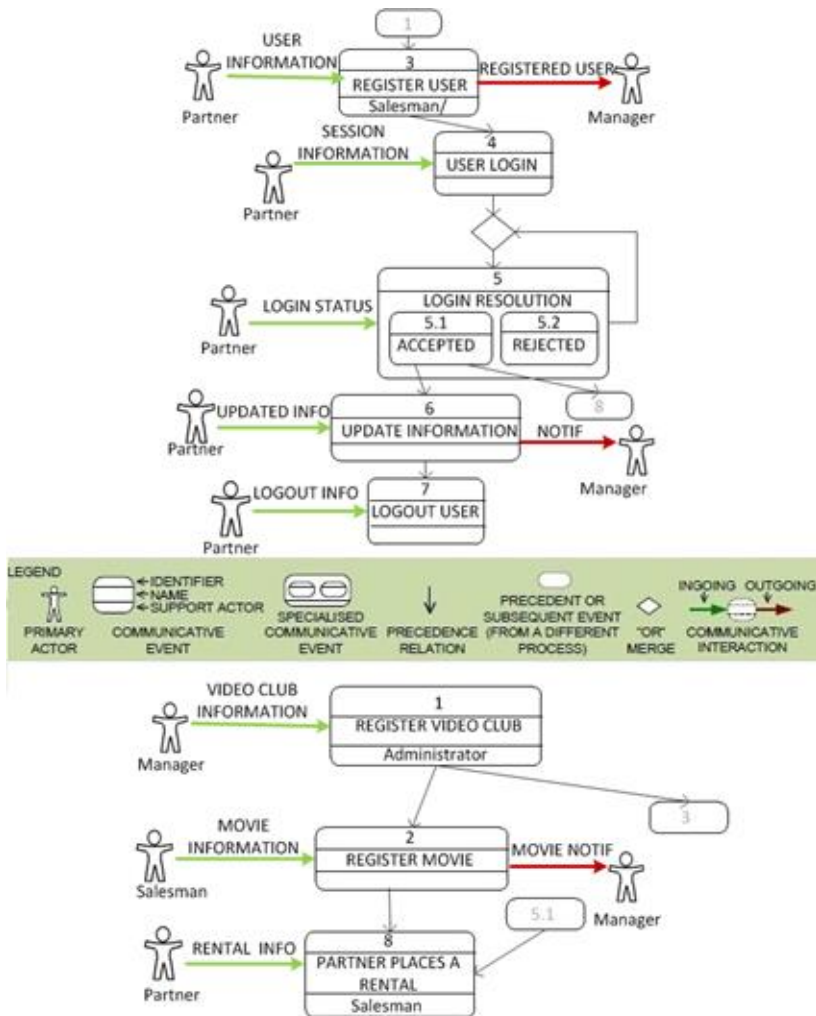


Figure 5.2. Excerpt of a CA model for the Video Club case.

A CED has relationships to specify ingoing and outgoing communicative interactions and three actor roles: i) the primary roles (i.e. primary actor) that trigger the CE and provide the input information, ii) the receiver roles (i.e. receiver actor) that need to be informed of the occurrence of an event; and iii) the interface roles (i.e. support actor) that is in charge of editing and entering input information. In the example, in the CE Register User, the partner acts as primary role, the manager as receiver role and the salesman as

interface role. To describe a CE in detail, España et al. [21] proposed to use Event Specification Templates.

The Event Specification Template structures the requirements [21] and is a textual specification technique that is used to describe both ingoing and outgoing messages transmitted to the IS in a *Communicative Event*. It uses a Message Structure to define the information that is communicated in the event.

The template is composed of a header and three categories of requirements: contact, communicational content and reaction requirements. The *header* contains information about the CE such as event identifier, name, goal, a narrative description, and so on. *Contact* (requirements related to the triggering of the event by an actor to communicate something to the information system, e.g. preconditions), *message* (specify the contents of the message being communicated to or from the IS, e.g. message fields, domain of the message fields, message constraints); and *reaction requirements* describe how the IS reacts to the communicative event occurrence (e.g. stores new knowledge, makes new knowledge and conclusions available to the corresponding actors). Therefore, this category of requirements includes business objects being registered (i.e. treatments) and outgoing communicative interactions being generated by the event (e.g. linked behaviours and linked communications), among other requirements. Our research covers the testing of the requirements related to communicative events. A simplified Event Specification Template of event is shown next in Table 5.1.

The *Message Structure* specifies the information communicated to or from the Information System [105]. Table 5.1 shows the Message Structure for the communicative event (i.e. a salesman registers a movie) in our example.

The following grammatical constructs are of interest for the purpose of Test Model derivation (see [105] for further information on this technique).

Table 5.1. Example of an Event Specification Template

7. REGISTER MOVIE INFORMATION			
<p>Goals: From the point of view of the information system, the objective of this event is to record the relevant information about the movie rents.</p> <p>Description: When a partner rent movies, both the rent date and return date should be registered in the SI. More than one movie may be included in a rental. The rental price is calculated as a derivative, adding the movie prices that make up the rental.</p>			
Contact requirements			
<p>Primary actor: Partner Communicational channel: In person Temporal restrictions: none Frequency: none</p>			
Communicational content requirements			
<p>Support actor: Salesman Communication Structure: (see the next partial view of a Message Structure)</p>			
FIELD	OP	DOMAIN	EXAMPLE VALUE
RENTAL =			
< id rental +	g	Number	7260
pick up date +	i	date	18-05-2016
return date +	i	date	20-05-2016
total +	i	money	2.5
Partner +	i	PARTNER	User100, Jorge Vidal
RENTALLINES =			100, Valencia,...
{RENTALINE =			
<rental number +	i	Number	250
price +	d	Money	this.movie.rental_price
Movie >	i	MOVIE	100, Everest,...
MOVIESTATUS =			
<status +	i	Text	Rented
Movie >>}	i	MOVIE	100, Everest,...
<p>Structural restrictions: One rent can have many movies. Contextual restrictions: Rental is identified by the id rental.</p>			
Reaction requirements			
<p>Treatments: The rent lines are recorded and they are assigned to the movie rent. Movie status is updated to "Rented". Linked behaviour: The rent is related to a partner. Linked communications: none</p>			

A *Substructure* is an element that is part of a message structure. For example, Partner, RENTALLINES, RENTALINE, Movie, MOVIESTATUS are substructures of RENTAL. The initial substructure is the first level of a message structure. In our case RENTAL = <id rental + pick up date + return date + total + Partner + RENTALLINES, MOVIESTATUS>.

There exist two classes of substructures;

- i. *Field*: Basic informational element of the message and is not composed of other elements.
 - a) *Data Field*: To represent a piece of data with a basic domain. For id rental, pick up date, return date, total, rental number, price and status.
 - b) *Reference Field*: Field whose domain is a type of business object. For instance, Partner and VideoClub both refer to a partner and VideoClub respectively, and are already known to the IS.
- ii. *Complex substructure*: Any substructure that has an internal composition.
 - a) *Aggregation Substructure*: Specify the composition of several substructures in such a way that they remain grouped as a whole. It is represented by angle brackets < >. For instance, RENTALLINE= <rental number + price + Movie>.
 - b) *Iteration Substructure*: Specify a set or repetition of the substructures it contains. It is represented by curly brackets { }. For instance, a submission can be related to several RENTALLINES and MOVIESTATUS.
Each field is characterised by properties, some of which are described below.
It must have a significant Name (e.g. pick up date).

An acquisition operation (OP) specifies the origin of the information that the field represents.

1) Input (i): The information of the field is provided by the primary actor.

2) Generation (g); The IS can automatically generate the field information (e.g. id rental).

3) Derivation (d): The field information is already known by the IS and therefore can be derived from its memory; i.e. it was previously communicated in a preceding communicative event. This operation can have an associated derivation formula (e.g. price in RENTALLINE).

If the attribute operation is of the “derivation” type, the derivation formula indicates the formula in ALF language (e.g. `this.movie.rental_price` for price of RENTALLINE).

A Domain specifies the type of information that the field contains (e.g. text, number, Partner).

An Example Value is a value for the field, provided by the organisation (e.g.7260 for id rental).

The minimum Cardinality is a value that indicates the minimum cardinality of the data field. The maximum Cardinality is a value that indicates the maximum cardinality of the data field.

An isIdentifier is a Boolean value that indicates if a data field is an identifier field of a substructure.

For each Communicative Event in the CED a message structure is required with information needed to express its behaviour.

5.3 Test Design

In this phase, the test basis information is taken from the requirements model and is transformed into a Test Model (TM) with the test conditions/items (something that could be tested e.g. services, triggers, assertions and links) ordered by precedence relationships, which generates an ordered graph. This model conforms to the Metamodel of the Test Model (TMM). The details of metamodel and transformations are discussed in Chapter VI.

Then, the different paths are identified from Test Model to generate the Test Scenarios Model with the test items combined into abstract test cases. The test cases are abstracts in the sense that they do not contain concrete objects. The metamodel and transformation are discussed in detail in Chapter VI.

5.3.1 Test Data

For specification of test values, data is extracted from the Test Model and stored in a data base. These values are the example values passed to the test model from the requirements model. Another data source is the values directly entered into the data base by the user (modeller or tester). Finally, a web-based generation strategy of valid test strings using regular expressions provided by the user (modeller or tester) may be used to generate test values (e.g. [106]).

5.4 Test Generation

Test cases can also be generated by traversing from parent root to child node using a classic pathfinder or graph traversal algorithm [107]. When all the nodes in a path from parent to child node are traversed, then it is considered as one test scenario. All nodes should be covered to make sure all flows in an application are covered. One flow is considered as one test scenario.

Test suite for CS is a set of one or more test scenarios. Each test scenario is a story that consists of one more test cases. In this phase, abstract test scripts are generated from a test scenario model. Then, the concrete and executable test cases (scripts) are generated from a test scenario model to describe what the system is supposed to do with the inputs taken from the data base, as well as the oracle and goals of the test case. All this is done through model-to-model and model-to-text transformations following a model-driven development. The details of model-driven generation are discussed in Chapter 6. In the following sections we summarize the design decisions considered for generation of the test cases.

5.4.1 Test Case Selection

Since the generation process generates many test cases to cover the different test scenarios, we need know which test cases should be inventoried and which should be deleted. Because test scenarios may share some statements in common (common path in the test scenario model), the generation process of test cases may get a large number of

duplicate test cases. The criterion to identify duplicate test cases is by matching the Test statement. Then, we omit the generation of the duplicated test cases and keep both the type of generated test case and the duplicate test cases to report as a result of the generation process.

5.4.2 Addressed Quality Goals

The test cases mainly address the validation of two CS quality goals [9]: the correctness (covers both syntactic correctness -right syntax or well-formedness, and semantic correctness -right meaning and relations relative to the knowledge about the domain) and completeness (i.e. all the necessary information is defined in the CS). However, other quality goals are also addressed, such as Consistency, Confinement, Comprehensibility and Changeability (see Section 4.1.2).

5.4.3 Test Types

Test types define the general types of expectations that need to be specified in test cases for testing conceptual schemas. In conceptual modeling, (a fragment of) the lifetime of an information system is a sequence of CS states, which represents a snapshot of the state of the domain as an instance of the conceptual schema [53]. In our approach, we conceive test cases for testing conceptual schemas as a sequence of states of the CS (i.e. concrete user story), together with formalized expectations (i.e. test oracles and test goals) about these states. This sequence of states is expected to be successfully executed if the required knowledge is correctly and completely defined in the conceptual schema. So, these kinds of tests are as follows:

Asserting the content of an object

The objective of this test kind is checking that, in a concrete object state (explicitly created by the fixture of the test case) the value of basic and derived knowledge defined in the schema is as expected. If the assertion is true, then the conceptual schema has the correct knowledge to provide information about the object 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).

Asserting the Occurrence of an Event

The second test kind corresponds to the assertion of the occurrence of an event in a CS state reached by a test case. If the assertion is true, then the event has occurred as expected and the resultant CS state complies with its specification. Otherwise, some constraints that prevent the event from occurring are too restrictive, or the specification of the event is not correct.

Asserting the Non-Occurrence of an Event

This is the rationale for the third test kind, which corresponds to the assertion of the non-occurrence of a CS event. If the assertion is true, then the event has not been allowed to occur as expected. Otherwise, the set of constraints related to the event need to be modified (i.e. need to be more restrictive) in order to prevent its occurrence.

Then, the tester can select the type of test case:

- **Partial** (only positive test cases): This kind of test case uses assertions to test the occurrence of an event and the contents of CS objects.
- **Complete**, which adds test cases (thus of positive test cases) with some negative conditions such as values out of range based on variable partitions that can be derived from CS information, constraint violations, minimum cardinality violation, and unique value violation for class variables. In this way, we test the non-occurrence of an invalid event.

In this context, the constraints that can be validated are restricted to those that can be represented in ALF language [83].

5.4.4 Test Generation Criteria

In addition, for selection of the test cases to be generated, our framework applies a set of generation criteria adapted from Andrews' proposal [108] based on coverage elements (i.e. classes, associations and generalizations) in the structural part as well as the behavioural part (condition, all message paths) (see Table 5.2).

Table 5.2. Test generation criteria for UML CD-based Conceptual Schema

<p>Association-end multiplicity (AEM) criterion Given a test suite T and a test model TM, T must cause each representative multiplicity-pair in TM to be created.</p>
<p>Generalization (GN) criterion Given a test suite T and a test model TM, T must cause every specialization defined in a generalization relationship to be created.</p>
<p>Class attribute (CA) criterion Given a test suite T, a test model TM, and a class C, T must cause a set of representative attribute value combinations in each instance of class C to be created.</p>
<p>Condition coverage (Cond) criterion Given a test suite T and test model TM, T must cause each condition in each decision in TM to evaluate to both TRUE and FALSE.</p>
<p>Full predicate coverage (FP) criterion Given a test suite T and test model TM, T must cause each clause in every condition in TM to take the values of TRUE and FALSE while all other clauses in the predicate (condition) have values such that the value of the predicate will always be the same as the clause being tested.</p>
<p>Each message on link (EML) criterion Given a test suite T and diagram Class (DC), T must cause each message on a link connecting two objects in CD to be executed at least once.</p>
<p>All message paths (AMP) criterion Given a test suite T and test model TM, T must cause each possible message path (sequence of message numbers) in TM to be taken at least once.</p>
<p>Collection coverage (Coll) criterion Given a test suite T and test model TM, T must test each interaction with collection objects of various representative sizes at least once.</p>

5.4.5 Deriving test goals

In our framework the test generation criteria and test types can be used to derive test goals. Figure 5.3 shows some examples of test goals for Video Club CS based on the Coll criterion.

For example, (i) the Coll criterion may be associated with test goals for test case positive that require the system to be brought into a

specific configuration that has a specified number of objects in a collection appearing in a test model; (ii) the EML criterion can be used to generate a test goal for test case positive that stipulates the specific links to be exercised during tests; (iii) the CA and Cond criteria may be used to derive the attribute value in each instance of class rental to be created; (iv) the FP and Cond criteria can be used to derive test goals for test case negative that stipulate values for a specific condition; (v) the CA criterion may be used to generate a test goal for a test case negative that validate the attribute value in each instance of class videoclub to be created; alternatively, (vi) the AMP criterion can be used to define a test goal for test case positive that stipulates the specific paths to be exercised during tests.

- i. Validate the Object 'videoclub_' was created (test case positive)
- ii. Validate the link 'videoclub_movie.createLink(videoclub_movie_);' with a valid value (test case positive)
- iii. Validate the derived Attribute 'context rental inv property_total_derivation:' (test case positive)
- iv. Validate unique value: 'context VideoClub:: new_videoclub() post: VideoClub->isUnique e (e.id_videoclub)' (test case negative)
- v. Validate a value above the upper limit 'context videoclub:: new_VideoClub() pre: p_atrid_videoclub<=10000' (test case negative)
- vi. Validate the 'line 28' with valid values (test case positive)

Figure 5.3. Examples of test goals generated for Video Club CS

5.4.6 Concrete and Executable Test Cases

In our test framework, we adapt the UTP's terminology [109] and consider that a test case is a specification of one case to test the conceptual schema including what to test with, which input, result, and under which conditions. Then, the test cases generated by our proposal exhibit the following properties:

- A test case consists of a fixture and one or more statements that execute one of the tests applicable to conceptual schemas, such as

testing assertions about the occurrence or the non-occurrence of an event. The fixture is a set of statements (e.g. create an object or link, execute an object method) that create a CS state and define the values of the CS variables.

- The oracle and test goal of each test case is derived from type of test cases selected in the previous phase. The expected value (oracle) to the positive test cases is the `assertEquals` or `assertTrue` equal “true” and with negatives conditions the `assertFalse` must be true otherwise the test case failed.
- Each execution of a test case starts with the execution of the fixture.
- It is assumed that the execution of each test case starts with an empty state. With this assumption, test cases of a CS are independent of each other, and the order of their execution is therefore irrelevant.

In ALF, an executable test case is an activity that provides the specification of parameterized behaviour as the coordinated sequencing of subordinate ALF units. It is the fundamental mechanism for behavioural modelling in ALF.

Each concrete test case has a name and consists of a set of statements (see Figure 5.4).

```
private import namespace::*;
public import Library;
//Goal: ... <oracle< ... (<test type>)
activity TestCaseName () {
...
assert ...
}
```

Figure 5.4. Test Case Structure

The last statement of a concrete test case is an assertion. The formal definition of ALF Language syntax is given in [82]. In this section, we describe the syntax and semantics of the five kinds of statements related to test conceptual schemas:

- Statements that update the information of the CS objects,
- Statements that assert the occurrence of events,
- Statement that assert the non-occurrence of events,
- Statements that assert the content of the CS objects.

Updating the information of the CS object

When the execution of a test case begins, the CS information is assumed to be empty and, therefore we need to set up in a progressive way the different CS states to check a state that cannot be reached by valid events.

ALF includes statements that can be used to explicitly set up a CS state in a test case. We describe them below using examples based on the schema fragment of Video Club (see Figure 5.5).

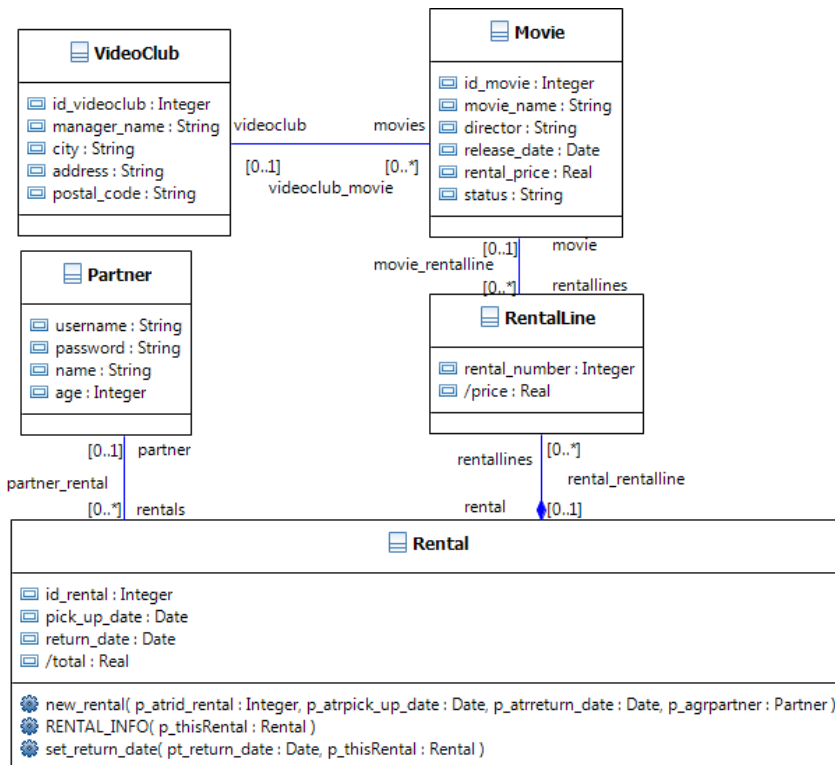


Figure 5.5. UML class diagram for Video Club CS

- We define that entityID is a new instance of the any entity type with the following statement:

```
EntityType entityID=new EntityType();
```

- 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 Inconclusive.

Often, it is convenient to state in a single statement the creation of a new entity entityID as an instance of entity type EntityType. The syntax is as follows:

```
entityID= new EntityType (parameter1=value1, ...,  
parameterN=valueN);
```

where entityID must be a new identifier and the value_i are values or expressions. For example, for creation of a videoclub instance:

```
videoclub_= new  
VideoClub(p_atrid_videoclub=100,p_atrmanager_name= "Jose  
Vicente Vidal",p_atrcity= "Valencia",p_atraddress=  
"Guardia Civil 21",p_atrpostal_code= "46020");
```

- Instances of an n-ary UML association Assoc with roles r₁, . . . , r_n are created with the following statement:

```
AssociationName.createLink(entityA, entityB);
```

Where entityA and entityB must be end members of the association. For example, to create an instance of the association videoclub_movie:

```
videoclub_movie.createLink(videoclub_,movie_);
```

- Entities can be deleted with the following statement:

```
objectID_.destroy();
```

For example, to delete the videoclub instance.

```
videoclub_.destroy();
```

The deletion of an entity implies the deletion of its attributes and the links in which it participates. However, note that (per UML Superstructure, 7.3.3 [110]) the composition annotation is on the part end of the composite association. For example:

```
assoc rental_rentaline {  
    public 'rental':Rental[1];  
    public 'rentallines': compose RentalLine;  
}
```

That is, in the above association, `Rental` is the composite while `Rentalline` is the part. Thus, when an instance of class `Rental` is destroyed, if there is a link of association `rental_rentalline` with that object at one end, then that link and the instance of `Rentalline` at the other end will also be destroyed.

Asserting the Occurrence of Events

An event is an execution of some operation (method) of the schema, which may have several kinds of defects. Among which are highlighted:

1. The pre-conditions of the event may not allow the occurrence of valid events.
2. The post-conditions may not precisely define the intended effect of events.
3. The method of an operation may produce a CS state that does not satisfy the schema invariants.

Testing the schema may be a practical means of detecting those defects. This is done by setting up for each event in the requirements model one test case with a CS state (i.e. fixture) and an instance of that event followed by an assertion of the (satisfactory) occurrence of that event.

In ALF, the event (or operation method) is a behavioural feature of a class that provides the specification for invoking an associated

method behaviour. Only classes may have operations as features. An operation is called on an instance of a class that has it as a feature using an invocation expression:

```
entityId.EventId();
```

execute the EventId associate with the entityId, whose characteristics (attribute values) can be defined as in the case of entities with the assignment of the value for its attributes att1,..., attn. The syntax is as follows:

```
entityId.EventId(att1=value1,..., attn.=valuen);
```

As an example:

```
videoclub_.movieunique();
```

Once the concrete event EventId has been executed in a test case in order to assert that it may occur in the current state of the CS, the conceptual modeller asserts that the current CS state must be consistent by writing the following statement:

```
Assert<AssertType>(("message", assertion);
```

As an example:

```
AssertTrue("MovieUnique", videoclub_.movies->isUnique e  
(e.id_movie));
```

The verdict of this assertion is determined as follows:

1. Check that the preconditions of the event are satisfied. The verdict is Inconclusive if any of the event preconditions is not satisfied.
2. Execute the method of the corresponding operation.
3. Check that the new CS state is correct (as defined in Asserting the CS state). The verdict is Inconclusive if any of the constraints is not satisfied.

4. Check that the event post-conditions are satisfied. The verdict is Inconclusive if any of the post-conditions is not satisfied; otherwise the verdict of the whole assertion is Pass.
5. Check that the current CS state is correct (as defined in Asserting the semantic correctness of a CS state). The verdict is Fail if that check fails (events may not occur in incorrect CS states). A CS state is called semantically correct if it satisfies all invariant defined in the conceptual schema.

If the verdict from step 1 is Inconclusive, then the conceptual modeller must change the CS state in order to make it valid. If the verdict from steps 1, 2 or 3 is Inconclusive, then the event has not occurred as expected by the conceptual modeller/tester. If the verdict from step 1 is Inconclusive then the following two cases are possible: (1) domain experts consider that the CS 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 modeller/tester 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 3 is Inconclusive, then the method, the event constraints or some schema constraint must be ill-specified. If the verdict from step 4 is Inconclusive then either the method of the operation or some post-condition is incorrect: the method may not produce the intended CS state, or the post-conditions may be ill-specified. If the verdict from step 5 is Fail, then the conceptual modeller/tester must change the CS state in order to make it valid.

As an example, let's assume the extension of Figure 5.5 shown in Figure 5.6, in which video clubs are restricted with pre- and post-conditions, which restrict the id of the movies to be unique and values between 1 and 10000. Consider, now, the following test case (see Figure 5.7):

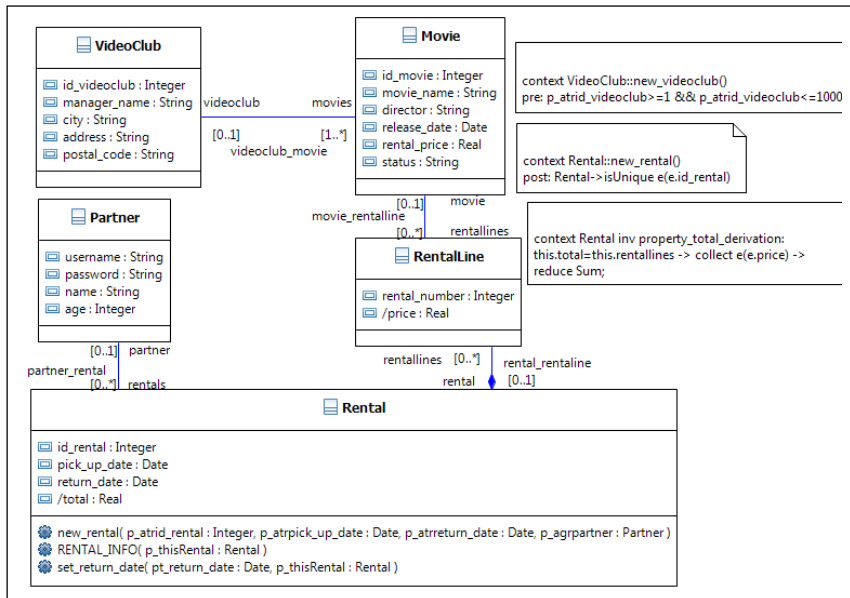


Figure 5.6. VideoClub CS with examples of pre, post-conditions and invariants

```
private import ::*;
public import Alf::Library::BasicTypes::*;
public import Alf::Library::Asserts::*;
// Conceptual Schema under Test : VideoClub
// Goal: Validate the integrity constraint 'context VideoClub inv MovieUnique:body:this.movies->isUnique
// The Script consists of 1 Test Scenarios
activity AbsTScenario_1_VideoClub () {
    videoclub_ = new VideoClub(p_atrid_videoclub=100,p_atrmanager_name="Jose Vicente Vidal",
        p_atrcity="Valencia",p_atraddress="Guardia Civil 21",p_atrpostal_code="46020");
    movie_ = new Movie(p_atrid_movie=90000,p_atrmovie_name="Jurassic World",p_atrdirector="Colin Trevorr",
        p_atrrelease_date=new Date(29,5,2015),p_atrrental_price=new Real(2,00),p_agrvideoclub=videoclub_);
    // Links
    //videoclub_movie.createLink(videoclub_,movie_);
    //Integrity Constraint 'context VideoClub inv MovieUnique: body: this.movies-> isUnique e(e.id_movie)
    videoclub_.movieunique();
    AssertTrue("MovieUnique", videoclub_.movies-> isUnique e(e.id_movie));
}
```

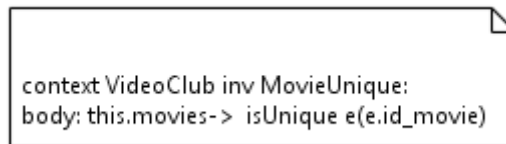
Figure 5.7. Example for validating pre-, post-conditions and invariants

The execution of the test case fails (as detected in step 5) because the occurrence of the event movieunique() is not defined in the conceptual schema. This event corresponding to the following invariant:

There are at least two possible actions that can be performed to make the test case Pass:

1. If movies may exist with duplicated id, then the `isIdentifier` has to be changed to `false` in the requirement model, and the test cases have to be regenerated.

2. If the domain experts confirm that when a `Movie` is created an invariant of the conceptual schema must ensure that it is not a duplicate code such as the invariant shown in Figure 5.8.



```
context VideoClub inv MovieUnique:  
body: this.movies-> isUnique e(e.id_movie)
```

Figure 5.8. Example of an invariant

Then, the execution of the test case pass.

A conceptual modeller 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. An event type is satisfiable if there is at least one CS state and one instance of that event type such that the event constraints are satisfied. If the conceptual modeller is able to set up a CS state and an instance of the Event for which `assert occurrence` gives the verdict `Pass`, then by definition Event is satisfiable. If the conceptual modeller is unable to set up such a CS state and event, this is not formal proof that `EventId` is unsatisfiable, but in many practical cases it provides a clue that helps to uncover a faulty event specification.

Asserting the non-occurrence of Events

A correct domain event specification must not only accept valid event executions, but also reject invalid ones. An event execution is invalid if it may not occur in the domain in the current CS state. Testing the conceptual schema may be a practical means of detecting missing events. This is done by setting up for each event one or more test cases with a CS state and an instance of that event considered may not occur

in that state, followed by an assertion of the non-occurrence of that event.

In ALF, in order to assert that the event eventId may not occur, the tool generates the following sentence:

```
assertFalse("message", assertion);
```

Consider, now, the following test case (see Figure 5.9):

The verdict of this assertion is determined as follows:

1. Check that the current CS state is semantically correct (as defined in previous Section). The verdict is Inconsistent 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.

```
private import ::*;
public import Alf::Library::BasicTypes::*;
public import Alf::Library::Asserts::*;
// Conceptual Schema under Test : VideoClub
// Goal: Validate unique value: 'context RentalLine:: new_rentalline() post: RentalLine->isUnique e (e.ref)
// The Script consists of 1 Test Scenarios
activity AbsIScenario_1_VideoClub () {
  videoclub_ = new VideoClub(p_atrid_videoclub=100,p_atrmanager_name= "Jose Vicente Vidal",
    p_atrcity= "Valencia",p_atraddress= "Guardia Civil 21",p_atrpostal_code= "46020");
  movie_ = new Movie(p_atrid_movie=90000,p_atrmovie_name= "Jurassic World",p_atrdirector= "Colin Trevorrow",
    p_atrrelease_date=new Date(29,5,2015),p_atrrental_price=new Real(2,00),p_agrvideoclub=videoclub_)
  //videoclub_movie.createLink(videoclub_,movie_);
  //Integrity Constraint 'context VideoClub inv MovieUnique: body: this.movies-> isUnique e(e.id movie)
  videoclub_.movieunique();
  partner_ = new Partner(p_atrusername= "fgranda",p_atrpassword= "123",p_atrname= "Ma Fernanda Granda",
    p_atrage=25);
  rental_ = new Rental(p_atrid_rental=200,p_atrpick_up_date=new Date(20,6,2015),p_atrreturn_date=
    new Date(20,12,2016),p_agrpartner=partner_);
  rental_.property_total_derivation();
  rentalline_ = new RentalLine(p_atrrental_number=250,p_agrmovie=movie_,p_agrrental=rental_);
  rentalline2_ = new RentalLine(p_atrrental_number=250,p_agrmovie=movie_,p_agrrental=rental_);
  AssertFalse("Not Exists",rentalline2_.instanceof RentalLine);
}
```

Figure 5.9. Example of test case for asserting the non-occurrence of events

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 modeller may prefer to change the assertion to assert occurrence. In the latter, the conceptual modeller/tester must define a new event constraint or refine an existing one in order to make it more

constraining. In the example of Figure 5.6, if we assume now that two rental lines cannot be created with the same number, then the following event constraint must be added to pass the previous test case:

```
context RentalLine::new_rentalline()  
post: RentalLine->isUnique e(e.rental_number);
```

Asserting the contents of CS objects

It is often useful to include in a test case an assertion on the current state of the CS. 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 events implies an expected result in the CS. In ALF, to assert that the current state of the CS satisfies a Boolean condition defined as a constraint, the tool generates the following statement:

```
assertEqual ("message", assertion);
```

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

Additionally, we have developed in ALF the following assertions to evaluate dates, real values, and compare data collections:

```
AssertEqualDate(in label: String, in value1: Date, in  
value2: Date)  
AssertEqualReal(in label: String, in value1: Real, in  
value2: Real)  
AssertList(in label: String, in list: any[*] sequence,  
in expected: any[*] sequence)
```

As an example, let's consider again the schema of Figure 5.6 and that the derivation rule of the derived attribute `total` of `Rental` class is defined as follows:

```
context Rental inv property_total_derivation:
  this.total=this.rentallines->collect e(e.price)->reduce
  Sum;
```

A conceptual modeler that wants to test that derivation rule may write the following test case (see Figure 5.10).

The verdict of the assertion is Fail. The conceptual modeller expects that `rentallines.prices` includes the prices of the movies, and therefore the result should be their sum.

```
private import ::*;
public import Alf::Library::BasicTypes::*;
public import Alf::Library::Asserts::*;
// Conceptual Schema under Test : VideoClub
// Goal:Validate the derived Attribute 'context rental inv property_total_derivation:(test case pos
activity AbsTScenario_1_VideoClub () {
  videoclub_ = new VideoClub(p_atrid_videoclub=100,p_atrmanager_name= "Jose Vicente Vidal",
    p_atrcity= "Valencia",p_atraddress= "Guardia Civil 21",p_atrpostal_code= "46020");
  movie_ = new Movie(p_atrid_movie=90000,p_atrmovie_name= "Jurassic World",p_atrdirector= "Colin Trev
    p_atrrelease_date=new Date(29,5,2015),p_atrrental_price=new Real(2,00),p_agrvideoclub=videoc
  //videoclub_movie.createLink(videoclub ,movie_);
  //Integrity Constraint 'context VideoClub inv MovieUnique: body: this.movies->isUnique e(e.id_movi
  videoclub_.movieunique();
  partner_ = new Partner(p_atrusername= "fgranda",p_atrpassword= "123",p_atrname= "Ma Fernanda Granda
    p_atrage=25);
  rental_ = new Rental(p_atrid_rental=200,p_atrpick_up_date=new Date(20,6,2015),p_atrreturn_date=
    new Date(20,12,2016),p_agrpartner=partner_);
  rental_.property_total_derivation();
  AssertEqualReal("property_total_derivation():money",
    rental_.total,rental_.rentallines -> collect e(e.price) -> reduce(Sum);
}
```

Figure 5.10. Example of test vase validating a derivation rule

The derivation rule does not derive the expected results because it assigns a fixed value to each movie price. The test case will pass if the derivation rule is corrected as follows:

```
context RentalLine inv property_price_derivation:
  this.price=this.movie.rental_price;
```

5.5 Test Prioritization

Since a testing process manages many test cases, we need to know how good a test case is. To do this job efficiently, we need to know the test case prioritization, which test cases should be executed? Which are critical? One problem in the design of tests to assess test case quality is that real software artefacts of appropriate size including real

faults are hard to find and hard to prepare appropriately (for instance, by preparing correct and faulty versions) [111]. Even when software artefacts with real faults are available, these faults are not usually numerous enough to allow the experimental results to achieve statistical significance [111].

In this context, mutation testing is one of the ways of assessing the quality of a test suite to prioritize efforts in those that are critical. This method injects artificial faults or changes into a CS (mutant generation) and checks whether a test suite is “good enough” to detect these artificial faults. The artificial faults can be created automatically, by using a set of mutation operators (MO) to change (i.e. mutate) some parts of the software artefact. Mutants can be classified into two types: First Order Mutants (FOM) and Higher Order Mutants (HOM) [112]. FOMs are generated by applying mutation operators only once. HOMs are generated by applying mutation operators more than once [113].

Assuming that the software artefact being mutated is syntactically correct, a mutation operator must produce a mutant that is also syntactically correct. Each faulty artefact version, or mutant, is executed against the test suite. The ratio of detected mutants over the total number of the non-equivalent mutants is known as the “mutation score” and indicates how effective the tests are in terms of fault detection. Thus, mutation test adequacy criteria can assist in optimizing the testing process [114]. It can be used for defining a test set - selecting tests from the immense test pool. The condition for test selection is detection of faults in the mutated software artefacts.

In Mutation testing the most critical activity is the adequate design of mutation operators so that they reflect the typical defects of the artefact under test. Therefore, we are required to design a set of mutation operators for Conceptual Schemas (CS) based on Unified Modelling Language (UML) Class Diagrams (CD). The main potential advantage of mutation operators is to describe precisely the mutants

that can generate and thus support a well-defined, fault-injecting process.

Figure 5.11 illustrates the definition process of mutation operators. As inputs, the metamodel of an UML Class Diagram [40], the defect types in a UML-based model [43] were provided.

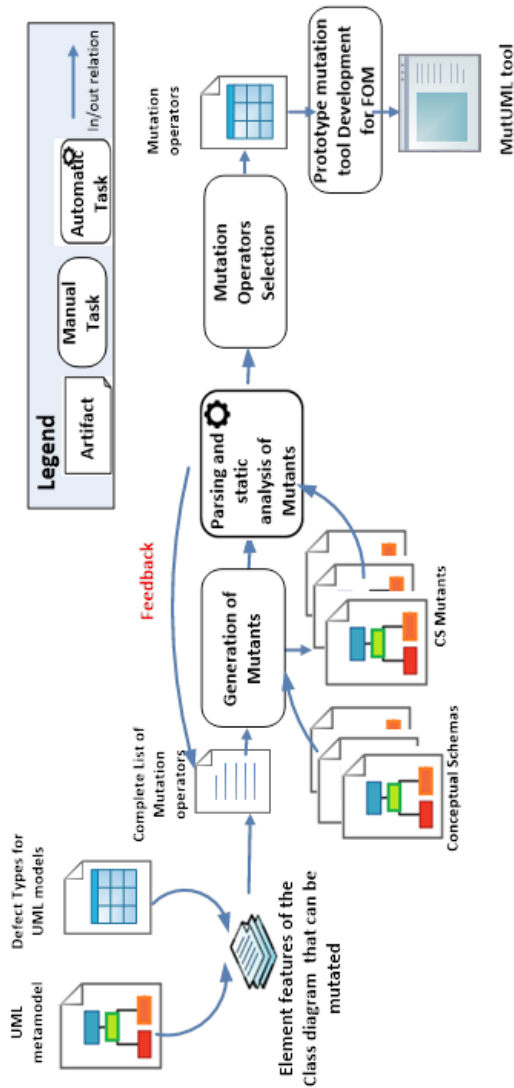


Figure 5.11. Selection process of the mutation operators

Each element of UML Class diagram was analysed based on defect types that can be injected. Then, all mutation operators were used to generate mutants. A static analysis and parsing (using ALF parser) of the mutants was performed in order to discard equivalent and non-valid mutants. Then, a selection process was performed in order to obtain a list of mutation operators (i.e. FOM and HOM) for mutation usage (see Table A.1 in Appendix A).

Finally, the researchers met for decision-making with two main objectives: 1) to focus on evaluating the usefulness of the mutation operators for FOM and 2) to automatize the mutant generation (see Section 7.7) and evaluate its feasibility. In a previous work [43], we presented a defects classification at model level and in [115] described the process of selection of the 18 mutation operators from a list of 50 for generating First Order Mutants to UML CD-based CS (see Table 5.3).

Table 5.3. Mutation operators for CS FOM taken from [116]

#	Code	Mutation Operator rule
1	UPA2	Adds an extraneous Parameter to an Operation
2	WCO1	Changes the constraint by deleting the references to a class Attribute
3	WCO3	Change the constraint by deleting the calls to specific operation.
4	WCO4	Changes an arithmetic operator for another and supports binary operators: +, -, *, /
5	WCO5	Changes the constraint by adding the conditional operator "not"
6	WCO6	Changes a conditional operator for another and supports operators: or, and
7	WCO7	Changes the constraint by deleting the conditional operator "not"
8	WCO8	Changes a relational operator for another and supports operators: <, <=, >, >=, ==, !=
9	WCO9	Changes a constraint by deleting a unary arithmetic operator (-).
10	WAS1	Interchanges the members of an Association.
11	WAS2	Changes the association type (i.e. normal, composite).
12	WAS3	Changes the multiplicity of an Association member (i.e. *-*, 0..1-0..1, *-0..1)
13	WCL1	Changes visibility kind of the Class (i.e. private)
14	WOP2	Changes the visibility kind of an operation.
15	WPA	Changes the Parameter data type (i.e. String, Integer, Boolean, Date, Real).
16	MCO	Deletes a constraint (i.e. pre-condition, post-condition constraint, body constraint)
17	MAS	Deletes an Association.
18	MPA	Deletes a Parameter from an Operation.

As opposed to code-based mutation, our mutation operators are based on the element characteristics of a UML CD-based CS and although some of the proposed operators perform syntactic changes at the constraints level, they are mainly focused (i.e. 41 of 50 operators) on the semantic changes of the high-level CD constructs. Our mutation operators are classified according to the element affected by the operator, injected defect type, and the action required by the mutation operator to generate valid mutants (syntactically correct). Since our purpose is to select mutation operators to be used to evaluate testing approaches, the selection process of mutation operators was divided into two iterations.

In the first iteration, some operators were excluded because they generated only equivalent mutants (e.g. UCO2, UAS3, UAS4) and non-valid mutants, (e.g. WCL4, UCO1, UAS1), which require a static technique (without CS execution) for detecting (e.g. syntax analysis or structural coverage analysis), and so are not useful for mutation testing. In the second iteration, we aimed to analyse the dependencies between different operators and to reduce the cost of applying mutation testing by selecting 18 mutation operators that generate only first order mutants.

These 18 mutation operators were implemented in our tool support called CoSTest (see Chapter 7) and validated on three conceptual schemas (see Sections 8.3.1 and Section 8.3.2). Based on the results obtained by applying the mutation testing, 56% (10/18) of our mutant operators generated a high number of killed mutants (score mutation=100 %). These results suggest that these operators generated mutants that are relatively easy to detect by the provided test suites. In the other case 44% (8/18) of the operators related to characteristics of associations (i.e. multiplicity and aggregation type) and constraints generated hard to detect mutants and their application would stimulate selection of high quality tests. However, the behaviour of the mutation operators may depend on the characteristics of the CS

they are applied to, such as the number, element type and complexity of constraints.

Therefore, the test cases that validate multiplicity and constraints have to be prioritized in a test suite as well as the test cases that cover complete test scenarios. However, the aggregation types require a static analysis for their validation.

5.6 Test Execution

Since test scripts (test case instructions) have to be executed against the conceptual schema under test, we require an executable CS as input to the testing environment.

5.6.1 Executable Conceptual Schema based on UML Class Diagram

A class diagram (see Figure 5.12) is the UML’s main building block that shows elements of the system at an abstract level (e.g. Class, association class), their properties (ownedAttribute), relationships (e.g. association and generalization) and operations. In UML an operation is specified by defining constraints. Figure 5.12 shows an excerpt of the UML structure [40] for a class diagram and highlights eight elements of interest for this work.

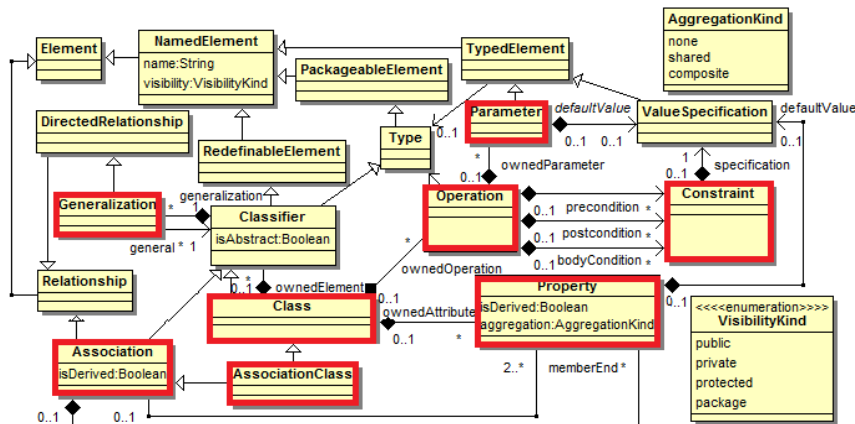


Figure 5.12. Excerpt of the Metamodel of an UML Class Diagram [40]

An executable model is at the next higher layer of abstraction, abstracting away both specific programming languages and decisions about the organization of the software (e.g. data structure and partitioning) so that a specification built in Executable UML can be deployed in various software environments without change [37]. A key ingredient of any Executable UML variant is the use of an Action language (kind of a pseudocode) that allows designers to completely specify fine-grained behavioural aspects of the model (e.g. to define the behaviour of a method of a class).

But, why do we need ALF? Programming languages such as Java, C++ or another programming language are not designed to manipulate the elements of a CS. They do not provide the facilities that we need to be able to express the actions in a CS in a clear and precise, yet abstract, manner. However, programming languages allow the developer to manipulate all sorts of implementation-specific features that are wholly inappropriate in a PIM. For instance, it is commonplace in modelling to want to navigate across an association (i.e. finding the associated object/s at the other end of an association). With a programming language we would need to know how the association is going to be implemented, for instance with any data structure therefore navigate the association using the operations related with this data structure. This immediately makes the model implementation platform specific. However, ALF allows the association to be navigated simply and concisely, without restricting the ways in which associations can be implemented. Figure 5.6 shows part of Video Club CS using an UML class diagram with constraints.

ALF is a platform independent language that works at the same semantic level as the rest of the UML-based CS. This means that actions allow direct manipulation of the elements of the PIM (no assumptions are made about middleware, implementation language or software design policy) and they are capable of being translated into different implementations for different platforms and languages. Syntactically, ALF is based on several key design principles [82]:

- ALF has a largely C-legacy (“Java like”) syntax, since that is most familiar to the community that programs detailed behaviours. Nevertheless, ALF allows UML textual syntax when it exists (e.g., colon syntax for typing, double colon syntax for name qualification, etc.).

- ALF does not require graphical models to change in order to accommodate the use of the action language (e.g., special characters are allowed in names, arbitrary names are allowed for constructors, etc.). Further, while ALF maps to the fUML subset in order to provide its execution semantics, it is usable in the context of models not limited to the fUML subset.

- ALF uses an implicit type system that allows but does not require the explicit declaration of typing within an activity, always providing for static type checking, based at least on typing declared in the structural model elements.

- ALF has the expressivity of OCL in the use and manipulation of sequences of values. These sequence expressions are fully executable in terms of fUML expansion regions, allowing the simple and natural specification of highly concurrent computations.

- ALF provides a naming system that is based on UML namespaces for referencing elements outside of an activity but also provides for the consistent use of local names to reference flows of values within an activity. ALF adds the concept of a *unit* to the basic UML concepts of namespaces and packages. A unit is a namespace defined using ALF notation that is not itself textually contained in any other ALF namespace definition. Units are lexically independent (though semantically related) segments of ALF text. Figure 5.13 shows the ALF unit definition for this example. In this definition, we can see the classes and associations that are formed the VideoClub package.

A unit may also have subunits that define namespaces that are owned (directly or indirectly) by the unit but whose ALF definition is given by a unit that is textually separate from the base unit. Inclusion in

the base unit is indicated using a stub declaration in the base unit and a namespace declaration in the definition of the subunit.

```

package VideoClub {
    public class RentalLine; // Unit definition
    public class VideoClub; // stub declaration
    public class Rental;
    public class Partner;
    public class Movie;

    public assoc partner_rental;
    public assoc movie_rentalline;
    public assoc rental_rentalline;
    public assoc videoclub_movie;
}

```

Figure 5.13. Textual definition for the package VideoClub by using ALF language

Therefore, we generate the CS under test using the structural part (class diagram with pre, post-conditions and invariants) and transforming CSUT into ALF units and transforming the pre, post-conditions and invariants into behavioural information (i.e. methods) to be used in CSUT execution (testing purposes) (see Figure 5.14). Further information is detailed in Section 6.3.6.

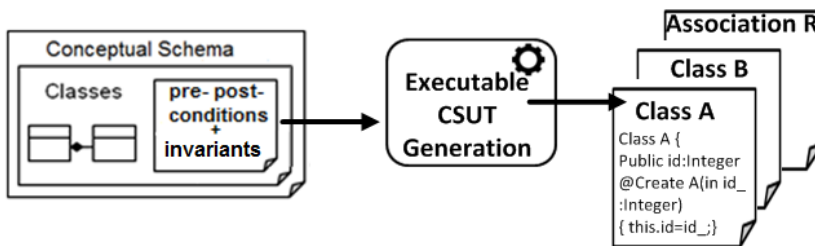


Figure 5.14 Overview to generate an executable CSUT

We decided to use the Reference Implementation⁵ as an fUML engine because (1) it is based on the reference implementation and (2) it provides an execution log. Thanks to (1) we have confidence in its conformity to the fUML specification. And (2) means that systematic

⁵ <http://modeldriven.github.io/fUML-Reference-Implementation/>

testing (i.e. reviewing hundreds of logs) is simpler than with the Moka⁶ implementation, which is more suitable for interactive testing.

The translation of UML CD-based CS into ALF is performed in two steps:

1. **Mode-to-text transformation** translates the UML CD-based CS into ALF units. This transformation is written in ATL code. It takes as inputs an UML CD-based CS, and gives as output an ALF-based CS. The resulting ALF-based CS contains the elements generated from the translation of all CS elements given as input.
2. **ALF unit parsing**. Semantically, ALF maps the CS to the Foundational UML (fUML [67]) subset. The resulting ALF-base CS is semantically equivalent to the original one. Then fUML provides the virtual machine for the execution of the ALF units.

An ALF-based CS can be executed from the command line using the ALF shell script (for Unix) or the `alf.bat` batch file (for Windows/DOS). The ALF-based CS is compiled in an in-memory representation and executed using the fUML Reference Implementation. Further details can be found in the ALF Reference Implementation [117].

The current version of our ALF transformation supports most UML CD constructs with the following notable exceptions: (1) features required to specify abstractions could be added with relatively little work; (2) transformation of OCL constraints. Currently, the UML CD-based CSs used in our approach use directly the ALF language to specify the constraints. But, there is an approach enabling OCL and fUML Integration by transformation that could be used to address this issue [118].

⁶ <https://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution>

5.6.2 Architecture and Testing Environment

In order to perform conceptual schema testing, our validation framework is based on the architecture shown in the scheme in Figure 5.15.

In our validation framework for conceptual schemas, conceptual modellers define an explicit specification of the conceptual schema of the information system under development. Then, a collection of automated tests is generated to test the schema by our testing framework. 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 ALF languages (the corresponding concepts and notation are explained in detail in [82]).

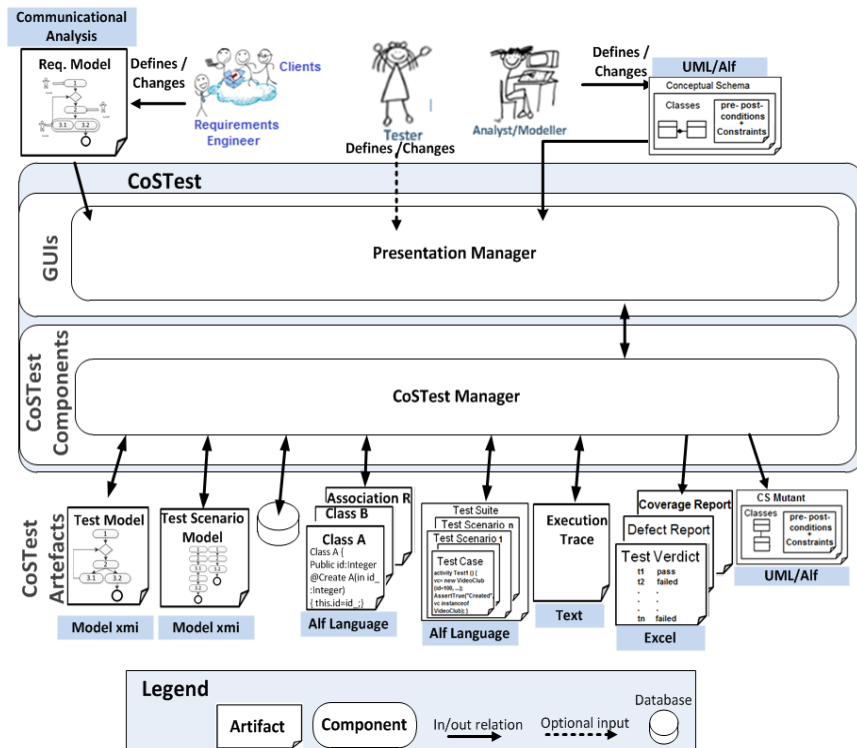


Figure 5.15. Testing environment to test Conceptual Schemas

Semantically, ALF maps the CS to the Foundational UML (fUML [67]) subset, after which fUML provides the virtual machine for the execution of the ALF language. An ALF-based CSUT can be executed from the command line using the ALF shell script (for Unix) or the `alf.bat` batch file (for Windows/DOS).

The CSUT is compiled to an in-memory representation and executed using the fUML Reference Implementation. Usage is:

```
alf [options] unitName
```

where `unitName` is the fully qualified name of a unit (e.g. test case) to be executed. The allowable options are

`-d level`: Sets the debug level for trace output from the fUML execution engine. Useful levels are:

- OFF turns off trace output.
- ERROR reports only serious errors (such as when a primitive behaviour implementation cannot be found during execution).
- INFO outputs basic trace information on the execution of activities and actions.
- DEBUG outputs detailed trace information on activity execution.

The default is as configured in the `log4j.properties` file in the installation directory.

`-f`: Treat the `unitName` as a file name, rather than as a qualified name. The named file is expected to be found directly in the model directory and the unit must have the same name as the file name (with any `‘.alf’` extension removed).

`-l path`: Sets the library directory location to `path`. If this option is not given and the `ALF_LIB` environment variable is set, then the value of `ALF_LIB` is used as the library directory location. Otherwise, the default of `Libraries` is used.

-m path: Sets the model directory location to path. Qualified name resolution to unit file paths is relative to the root of the model directory. If this option is not given, the default of Models is used.

-p: Parse and constraint check the identified unit, but do not execute it. This is useful for syntactic and static semantic validation of units that are not executable by themselves.

-P: Parse and constraint check, as for the -p option, and then print out the resulting abstract syntax tree. Note that the printout will occur even if there are constraint violations.

-v: Sets verbose mode, in which status messages are printed about parsing and other processing steps leading up to execution. If this option is used alone without specifying a unit name (i.e., `alf -v`), then just version information is printed.

More details can be found in the ALF Reference Implementation Wiki [119].

ALF is also such a language, but one that is an OMG standard that can be consistently implemented across a number of tools, promoting the same sort of interoperability for textual behavioural specification that the UML standard already does for graphical modelling.

This is the reason why in this thesis we focus on ALF language as our testing environment. However, the ideas presented in this document could be adapted to any of the above action languages.

5.6.3 Execution Trace

Execution traces resulting from execution of test cases are configured to report faults and syntax errors found during testing process by ALF parser.

Figure 5.16 shows an example of an execution trace for Video Club CS.


```

Constraint violations:
  behaviorInvocationExpressionReferentConstraint           in
C:\Users\Usuario\workspace\COSTest\ExecutableTestCases\UML-
ALF\VideoClub\VideoClub_TS_1_TC_36.alf at line 17, column 12
  instanceCreationExpressionDataTypeCompatibility         in
C:\Users\Usuario\workspace\COSTest\ExecutableTestCases\UML-
ALF\VideoClub\VideoClub_TS_1_TC_36.alf at line 20, column 22

  positionalTupleArguments                               in
C:\Users\Usuario\workspace\COSTest\ExecutableTestCases\UML-
ALF\VideoClub\VideoClub_TS_1_TC_36.alf at line 20, column 34

```

Figure 5.16. Example of an execution trace for Video Club CS

5.7 Test Evaluation

Since the tests are part of a validation and verification process, automated procedures (i.e. syntax and coverage analysis) were used to verify the models as a preliminary step to the test process.

5.7.1 Verifying the Syntax Correctness

All languages have a syntax, i.e. a set of rules about how elements of the language can be combined together meaningfully in that language. Then, specifications written in a specific language must comply with the syntax imposed by the language in which they are defined. This relationship between the specification and the language in which it is described is known as conformance.

We consider an executable conceptual schema is syntactically correct if all the elements satisfy the rules defined in the UML/fUML metamodel and well-formedness rules (WFR) – constraint that restrict the possible set of valid (or well-formed) models.

Consider the excerpt of the class diagram shown in Figure 5.17 and the constructor operation (in the context of class `CorporatePartner`) to create an instance of this class.

WFR: An alternative constructor invocation may only occur in an expression statement as the first statement in the definition for the method of a constructor operation

```

1 namespace VideoClub;
2 public import Alf::Library::PrimitiveBehaviors::StringFunctions::*;
3 public import Alf::Library::BasicTypes::*;
4 class CorporatePartner specializes Partner{
5     public contactName: String;
6     public creditLimit: Real;
7
8     @Create CorporatePartner(
9         // Corresponding to new_CorporatePartner
10        in p_atrusername:String,
11        in p_atrpassword:String,
12        in p_atrname:String,
13        in p_atrage:Integer,
14        in p_atrcontactName:String,
15        in p_atrcreditLimit:Real
16    ) {
17        this.contactName=p_atrcontactName;
18        this.creditLimit=p_atrcreditLimit;
19        super.Partner(p_atrusername,p_atrpassword,p_atrname,p_atrage);
20    }
21 }
    
```

Feedback: Syntax error at Line 19

Figure 5.17. Excerpt of the CS with a syntactically incorrect code

The above operation is not syntactically correct because the call to an alternative constructor is not the first line in the definition for the method of a constructor operation. Then, the repaired operation is shown in Figure 5.18.

```

1 namespace VideoClub;
2 public import Alf::Library::PrimitiveBehaviors::StringFunctions::*;
3 public import Alf::Library::BasicTypes::*;
4 class CorporatePartner specializes Partner{
5     public contactName: String;
6     public creditLimit: Real;
7
8     @Create CorporatePartner(
9         // Corresponding to new_CorporatePartner
10        in p_atrusername:String,
11        in p_atrpassword:String,
12        in p_atrname:String,
13        in p_atrage:Integer,
14        in p_atrcontactName:String,
15        in p_atrcreditLimit:Real
16    ) {
17        super.Partner(p_atrusername,p_atrpassword,p_atrname,p_atrage);
18        this.contactName=p_atrcontactName;
19        this.creditLimit=p_atrcreditLimit;
20    }
21 }
    
```

Figure 5.18. Example of a CS with the corrected Alf code

5.7.2 Validating the Semantic Correctness

We consider an executable conceptual schema (i.e. a set of ALF units with action-based operations) is semantically correct if all possible changes (inserts/updates/deletes/ . . .) on all parts of the

system state can be performed through the execution of those operations. Element exists but some statement about the domain is incorrect. For example, consider the excerpt of the class diagram and a test case composed by the operation `set_status` shown in Figure 5.19.

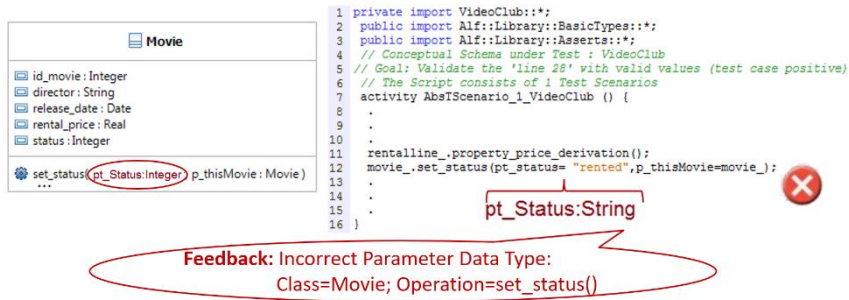


Figure 5.19. Excerpt of the VideoClub CS with a semantic incorrect defect

This conceptual schema is semantically incorrect since, for example, the operation `set_status` exists but the expected parameter type (i.e. String) is different than expected (i.e. Integer). Then, in order to correct this semantic error, the designer should change the type of the `pt_Status` parameter to an Integer. The repaired operation is shown in Figure 5.20.

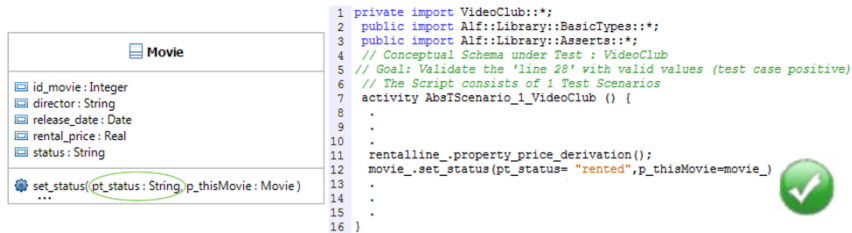


Figure 5.20. Example of the VideoClub CS with the corrected semantic defect

5.7.3 Verifying the Unnecessary Elements

In addition, unnecessary elements (i.e. redundant/repeated elements or extraneous elements) in the schema can be uncovered by analysis of coverage of the elements included in the conceptual

schema and the executed in the test cases. An example of a CS containing an extraneous association is shown in Figure 5.21.

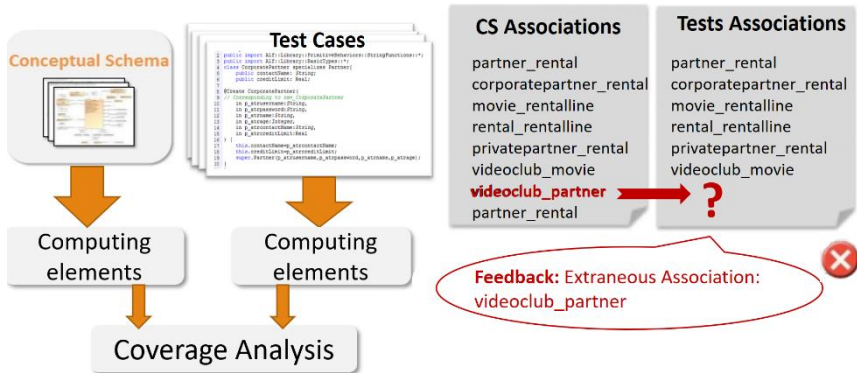


Figure 5.21. Example of a CS containing an extraneous association

5.7.4 Validating the Completeness

The method we have developed for validating the completeness property takes as input an executable model composed by a structural model (a UML class diagram) and a behavioural model (a set of Alf operations). So, a Conceptual Schema is complete if all elements exercised in the test cases exist on CS.

Then, our method returns either a positive answer, meaning that the behavioural model is complete, or a corrective feedback, consisting in a set of actions that should be included in some operation of the behavioural model in order to make it complete.

For example, consider the excerpt of the class diagram and a test case composed by the operation new Partner shown in Figure 5.22.

This conceptual schema is incomplete since, for example, the class Partner does not exist. Then, in order to correct this defect, the designer should change the CS by adding the Partner class. The repaired operation is shown in Figure 5.23.

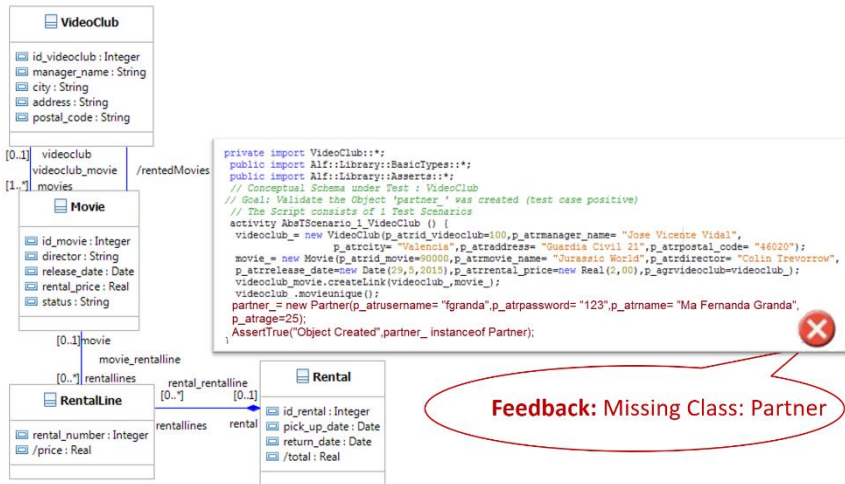


Figure 5.22. Excerpt of the CS with a missing defect

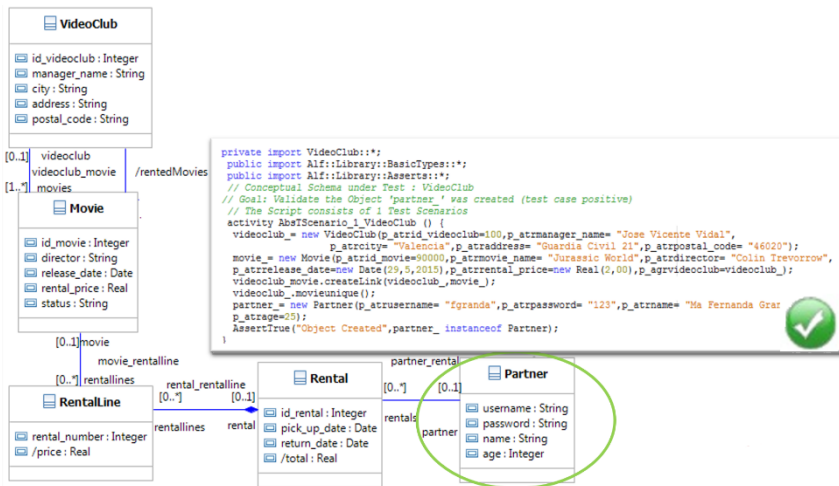


Figure 5.23. Excerpt of a corrected CS

This phase is done by using the oracles and goals included in the test cases. A test case returns the verdict Pass, Fail or Inconclusive. When the verdict is Fail, a defect list and a status of failed execution is provided. The execution of the test cases may produce an output with several defects (e.g. missing class, incorrect operation and missing operation), which are contained in the list. When the verdict is Inconclusive, this means that the execution of the test case is not

conclusive. For instance, if the fixture has caused a fault, this leads to an inconclusive status. This verdict can optionally return a defect list too. Otherwise, the status of the test case is Pass. As an example, consider again the conceptual schema of Figure 5.6. A conceptual modeller/tester that wants to test that Session entity may execute the test case shown in Figure 5.24.

```
private import VideoClub::*;
public import Alf::Library::BasicTypes::*;
public import Alf::Library::Asserts::*;
// Conceptual Schema under Test : PA_login
// Goal: Validate the Object 'session' was created (test case positive)
// The Script consists of 1 Test Scenarios
activity AbtScenario_1_PA_login () {
  partner_ = new Partner(p_atrusername= "username",p_atrpassword= "clave2016",p_atrname= "Usuario Example",
    p_atrage= 19);
  session_ = new Session(p_atrid_session=1,p_atrlogin_date=new Date(10,5,2016),p_atrlogin_time= "15:04",
    p_atrusername= "mfgranda");
  AssertTrue("Object Created",session_ instanceof Session);
}
```

Figure 5.24. Example of the test case

After test execution a generated error log is as follows (see Figure 5.25):

```
-----Test Case: 2-----
Constraint violations:
  instanceCreationExpressionConstructor                               in
C:\Users\Usuario\workspace\COSTest\ExecutableTestCases\UML-
ALF\VideoClub\PA_login_TS_1_TC_2.alf at line 10, column 19
  positionalTupleArguments                                           in
C:\Users\Usuario\workspace\COSTest\ExecutableTestCases\UML-
ALF\VideoClub\PA_login_TS_1_TC_2.alf at line 10, column 31
  classificationExpressionTypeName                                   in
C:\Users\Usuario\workspace\COSTest\ExecutableTestCases\UML-
ALF\VideoClub\PA_login_TS_1_TC_2.alf at line 12, column 37
```

Figure 5.25. Example of execution trace

The verdict of the assertion is *Fail*. Then, the execution trace is analysed by using the information shown in Table 5.4. Then, the defect missing class (or private) is reported. The test case will pass if the schema is corrected as Figure 5.26 shows.

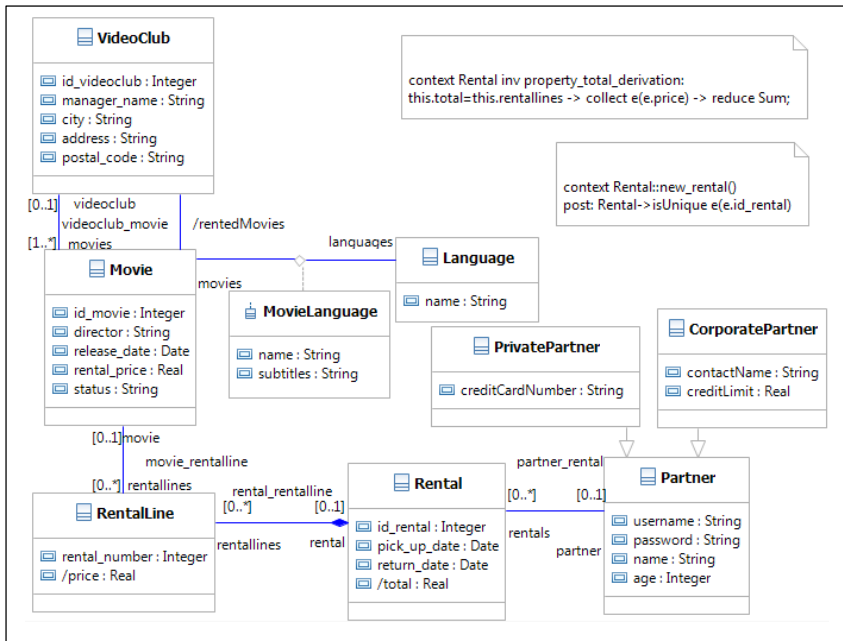


Figure 5.26. Extended UML class diagram for Video Club CS

Table 5.4. Relationship between fault and reported defect

Fault reported by	Defect Reported
propertyAccessExpressionFeatureResolution	Missing or private Association
instanceCreationExpressionConstructor	Missing Class (or private)
behaviorInvocationExpressionReferentConstraint	Missing Operation (or private)
propertyAccessExpressionFeatureResolution	Incorrect Association
linkOperationExpressionArgumentCompatibility	Incorrect Association Ends
instanceCreationExpressionConstructorlessLegality	Incorrect Constructor
assignmentExpressionSimpleAssignmentTypeConformance	Incorrect Parameter Data Type
tupleNullInput in a createlink statement	Incorrect null Value in Association Parameter
tupleNullInput in an operation statement	Incorrect null Value in Parameter
instanceCreationExpressionDataTypeCompatibility	Incorrect Operation Signature
behaviorInvocationExpressionArgumentCompatibility	Incorrect Parameter Data Type
superInvocationExpressionOperation	Incorrect Super Class

Finally, the test evaluation generates a report with test cases verdicts, detected faults, times report and coverage of test cases.

5.8 Overview of the CoSTest Testing Process

Testing methods for UML conceptual schemas are likely to differ depending on the testing criteria used [108]. To illustrate the testing process and highlight some of the issues that needed to be solved during the development of this PhD thesis, Figure 5.27 summarizes the testing process, which is divided into three phases.

- i. Test Suite Generation
 1. Transform the Requirements Model (based on Communication Analysis) into Test Model.
 2. Transform the Test Model into Test Scenario Model (sequences of events from test model).

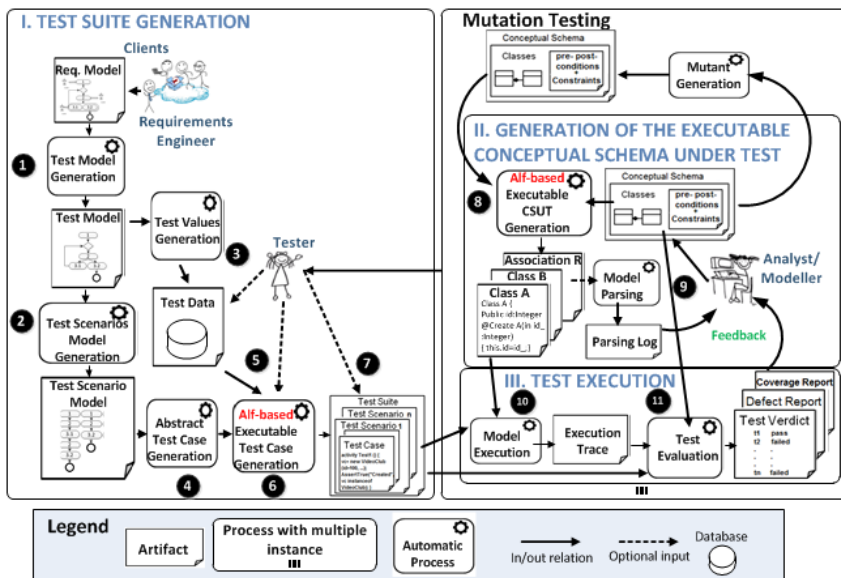


Figure 5.27. Overview of the testing process

- 3. Generate the test values for test cases from Test Model (variables concretization). Tester (optionally) can enter new test values.

4. Transform each test scenario into Test cases scripts (ALF script), which contains the abstract test cases.
5. Select the type of test cases (e.g. only positive test cases or including negative test cases)
6. Generate concrete and executable test cases
7. Prioritize and select the test cases for execution based on mutation analysis.
- ii. Generation of the Executable Conceptual Schema under test (CSUT)
 8. Generate the CS under test using both structural (class diagram) and behavioral information (pre, post-conditions and invariants) and transforming CSUT into ALF units to be used in model execution (testing purposes).
 9. Parse the CS before starting the execution of CS (testing process).
- iii. Test Execution
 10. Execute the test cases (scripts) against the CS under test.
 11. Generate the testing report and coverage analysis.

Since our proposal for generation of test cases complies with the principles of Model-Driven Testing, in the next chapter we describe in detail the Model-Driven process applied in our Testing framework. The tool support is described in detail in Chapter 7.

5.9 Summary and Conclusions

In this thesis, we propose a testing-based validation of conceptual schemas, mainly in order to enhance the validation of completeness (missing elements). However, confinement and changeability can be improved by analysing the elements covered and elements do not cover (extraneous elements) by test cases. Since the testing process has to transform the CSUT into an executable format, then redundant and incorrect elements are detected by the parser as a previous step to testing, so that the correctness goal is also improved. The CS comprehensibility by humans and tools is addressed when the

completeness, consistency and correctness of the CS is improved (see Tables 3.1 and 3.3 in Sections 3.2.1 and 3.3.5 respectively).

As the quality of a conceptual schema should not be considered as an after-thought, we aim to validate each step of the conceptual modelling process. Our proposal therefore allows both types of complete and incomplete models to be validated according to the evolution of the requirements.

Tool-supported rigorous analysis of design models can enhance the ability of developers to identify potentially costly design problems earlier and correcting design problems early also reduces the effort wasted on implementing faulty designs.

The testing process is based on test scenarios to execute high-level conceptual schemas regardless of the platform by using the standard Action Language by the OMG, ALF [82]. The validation framework follows a top-down approach to generate the test cases, where the test model is the master that generates the test scenarios and the test cases. In order to automate the test suite generation, we selected a model-driven architecture to address the analysis, design and implementation phases. The test design is therefore independent of the adaptation layer or test execution system and the test artefacts are independent of the implementation domain. This reduces costs and efforts in test system maintenance and supports communication between conceptual schema development (modellers) and the test department (testers).

In this chapter, we show how a model-driven generation for test cases written in the ALF language can be used to support testing of conceptual schemas. In the approach, a conceptual schema based on UML class diagram is transformed into an executable conceptual schema (ALF scripts), and a requirements model based on Communicational Analysis is transformed into a test model that characterizes valid sequences of test cases. A test case is an object configuration that describes a system state. A sequence of object

interactions is called a Test Scenario. These test scenarios are transformed into ALF scripts. Then, a conceptual schema based on a UML class diagram is transformed into an executable conceptual schema in order to execute against the test cases.

The methods provided as part of this thesis are organized in a *testing-based validation framework*.

Testing is part of a process of Validation & Verification, therefore, we used testing in conjunction with automated procedures (i.e. syntax and coverage analysis) aimed at verifying models.

Testing is one of the most critically important phases of the software development life cycle and consumes significant resources in terms of effort, time and cost. In this thesis, we share the criteria and try to reduce the number of test cases, while maintaining quality and customer satisfaction when faced with the challenge of testing complex applications with limited resources.

In the next chapters, we study in depth each of the proposed model-driven transformations as well as the validation method we have developed to validate them.

Chapter 6

TRANSFORMATION RULES

Model manipulation is a central activity in Model Driven Engineering (MDE) activities. Models are *merged* and *aligned* (e.g. to create a model of the system from different views), *refactored* (i.e. to improve their internal structure without changing their observable semantics), *refined* (i.e. to detail high-level models), and *translated* to other languages/representations, e.g. as part of code-generation, validation, verification or simulation processes. All these operations on models are implemented as model transformations, which automate the translation of models between a source and a target language using a model transformation language.

The objective of this chapter is to describe in detail the Model-Driven Testing (MDT) process applied in our Testing framework described in Chapter 5.

This chapter is divided into four sections: Section 6.1 describes the Model-Driven Testing (MDT) process, Section 6.2 analyses metamodels; Section 6.3 defines the different transformations types required by our MDT process; and Section 6.4 summarizes and concludes the chapter.

6.1 An Overview of the MDT Process

This Section offers a global view of test suite generation process from a CA-based requirements model, by means of metamodelling and transformations. Figure 6.1 shows the different types of models, metamodelling and transformations of our proposal at various levels of abstraction, where each model is an instance of its metamodel.

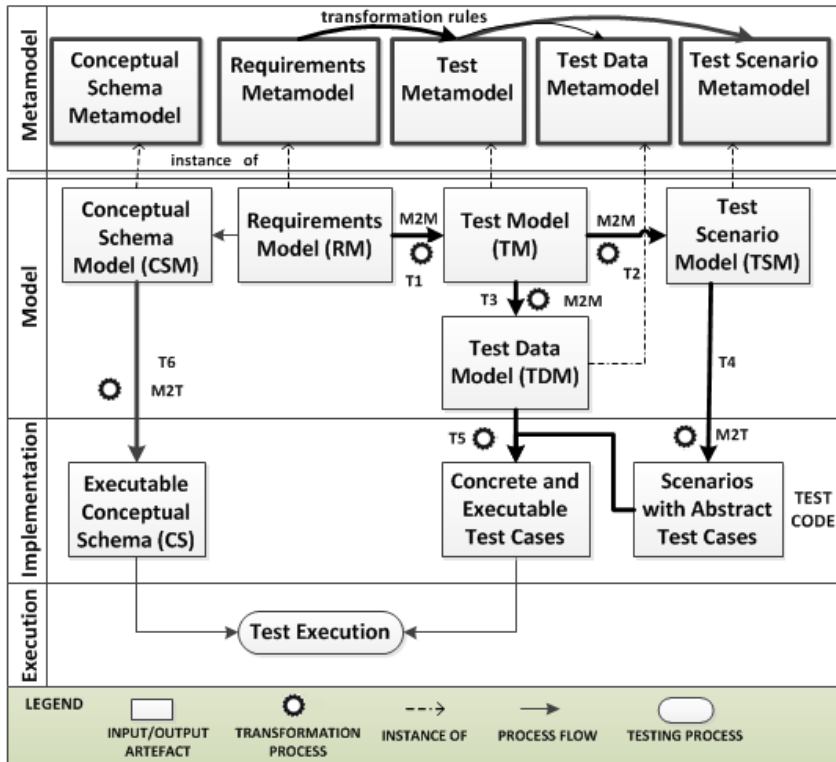


Figure 6.1. An overview of our MDT approach

Five metamodels and six transformations are required in the MDT process (see artefacts highlighted with a thick line in Figure 6.1). Figure 6.2 shows an overview of the metamodel elements and the transformation sequence with the results generated in the MDT process.

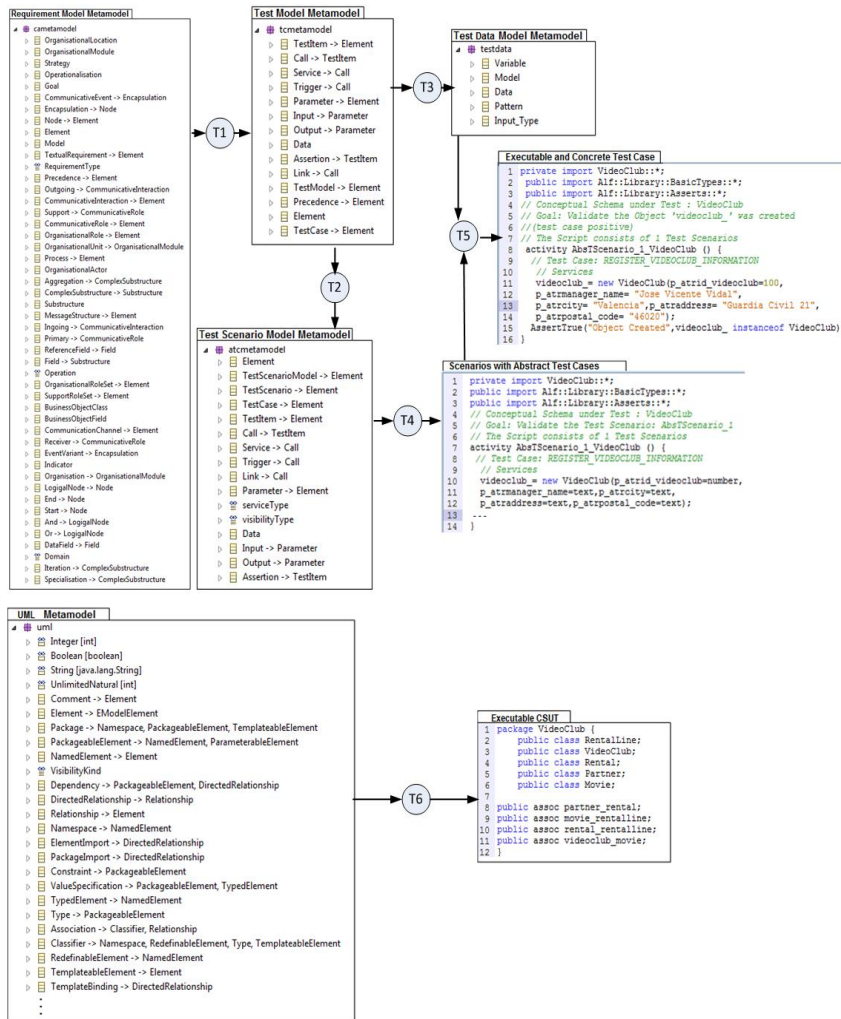


Figure 6.2. Overview of the sequence of proposed transformations

6.2 Metamodels

As shown in Figure 6.1 five metamodels are required in our approach, from which three are designed to define the information managed by the test suite generation process.

The first metamodel enumerates the elements to model the UML model, therefore more details about it can be found in UML documentation [40]. The second metamodel defines the elements to

model the requirements model (RM). The metamodels for CA based requirement models are described in [103][120]. Note that a full explanation of RM metamodel is outside the scope of this thesis, but several examples and experiences of this specification can be obtained at the project's web site (<https://staq.dsic.upv.es/webstaq/costest.html>). The third metamodel enumerates the elements to model the test model. The fourth one lists the elements to model test scenarios. The fifth one details the elements for the test data model and the fifth metamodel gives the UML structure [40] for a class diagram.

Previous work describes the *Test* metamodel and *Test Scenario* metamodel as the *Abstract Test Cases* metamodel [121]. We adapted them by (1) including some elements such as the traceability elements (i.e. `location` and `trule`) in the class `Element` of the first metamodel (2) changing the class name `TestComponent` by `TestCase` in both metamodels with the purpose of clarifying the element purpose, and so on.

The three metamodels defined for our proposal are described below.

6.2.1 Test Metamodel

The goal of this metamodel (`tcmetamodel`) is to include the relevant information for generating the Test Model (TM) with the test items and their order of precedence from a CA Requirements Model. The elements of this metamodel are represented in Figure 6.3 and described below.

The `TestModel` element represents a container for test case sequences. The key concepts of this metamodel are the `TestCase` and `Precedence` elements.

A `TestCase` element is a container for test items to be tested. A `Precedence` element is a relationship that models the test cases sequences.

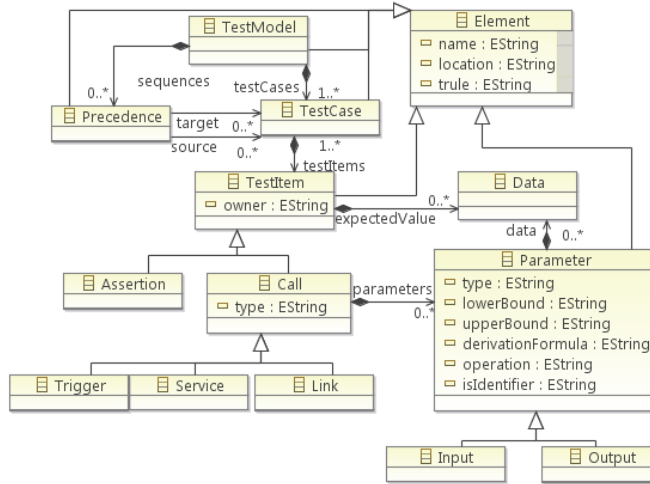


Figure 6.3. Metamodels for the first transformation adapted from [121]

`TestItem` element is a supertype that contains the CS elements to be tested. In a test model there is at least one test case that formalizes the user-system interaction sequences. This is related to the `Data` Class, which contains the expected value. In a test model there is at least one test case that formalize the user-system interaction sequences.

A `Link` element models a structural relationship between classes.

A `Service` element models an action performed by an external element of the system under test such as user input or a server response.

A `Trigger` element models an action that will be carried out by the system and that may be verified in order to evaluate the test correction, e.g. updates in the system data or other system or user outputs.

A `Call` element is a super type for the test item in a test case. Instances of `Call` element must be either a `Link`, `Service` or `Trigger` element. A `Call` element is a container of the `Parameter` elements, which contains information about the parameters required

in the test items (i.e. service, trigger and link). The `Parameter` may be an `Input` or `Output` parameter and is related to the `Data Class`, which contains its value for the concretization process.

The `Assertion` is an element that indicates `Constraint` statements. These statements are used to designate preconditions, post-conditions and the derivation condition of the class attribute.

The `Element` class is a supertype that contains the element name and traceability information of each one of them (i.e. location and transformation rule that generated it).

Figure 6.4 shows the corresponding OCL constraints for the TM metamodel. The constraints are: names must be unique within their respective contexts, classes must have a name and the multiplicity constraints for relations.

```
context TestModel inv TC_mult1: self.testCases->size()>0;
context Element inv Element_name: self.name<>null;
context Element inv Element_location: self.Location<>null;
context Element inv Element_true1: self.true<>null;
context Precedence inv Precedence_Name: self->forAll(e1,e2 | e1.name = e2.name implies e1=e2);
context Precedence inv Preced_TC: self.target<>null or self.source<>null;
context TestCase inv TestCase_Name: self->forAll(e1,e2 | e1.name = e2.name implies e1=e2);
context TestCase inv TI_mult1: self.testItems->size()>0;
context TestItem inv TestItem_owner: self-> forAll(e|e.ocIsKindOf(Link) = false implies e.owner<>null);
context Service inv Service_type: self.type<>null;
context Trigger inv Trigger_type: self.type<>null;
context Assertion inv Null_constraint: self.constraint<>null;
context Parameter inv Parameter_type: self.type<>null;
context Parameter inv Parameter_operation: self.operation<>null;
context Parameter inv Parameter_lowerBound: self.lowerBound<>null;
context Parameter inv Parameter_upperBound: self.upperBound<>null;
context Input inv Parameter_derivation: self->forAll(e|e.operation='derivation' implies e.derivationFormula<>null);
```

Figure 6.4. OCL Constraints for Test Metamodel

6.2.2 Test Scenario Metamodel

For the second transformation, we defined another metamodel (see Figure 6.5), which is the PST for our MDT proposal. The goal of this metamodel (`atcmetamodel`) is to define the information obtained after applying the algorithm for path analysis in the test model. A test scenario is a possible test case for testing a concrete test scenario. The elements of this metamodel are represented in Figure 6.5 and described below.

The `TestScenarioModel` element represents a container for test scenarios.

The key concept of this metamodel is the `TestScenario` element. A test scenario represents a user-system interaction sequence (user story). Going back to the preceding metamodel, a test scenario is a concrete path across TM. The steps performed during the test scenario execution are classified in terms of the concepts defined below.

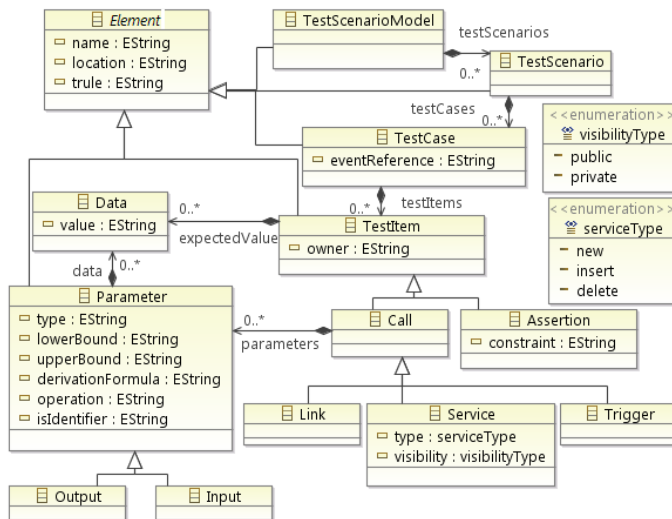


Figure 6.5. Test Scenario Metamodel adapted from [121]

A `TestCase` element is a supertype that contains the `TestItems` to be tested. In a test scenario there is at least one test case that formalizes the user-system interaction sequences.

The `TestItem` instances and the elements have already been introduced in the previous section.

6.2.3 Test Data Metamodel

The goal of this metamodel is to formalize the information required to concretize the values by applying the Category-Partition Method to functional requirements.

Figure 6.6 describes this metamodel, which is further explained below.

The `Model` element represents a container for operational variables, patterns and input types related to the test model.

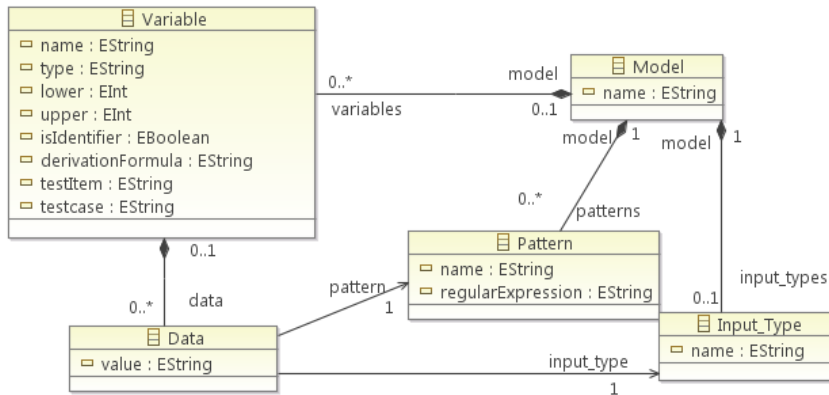


Figure 6.6. Test Data Metamodel

The key concept of this metamodel is the `Variable` metaclass, which contains information about the operational variables of the test model such as name, data type, lower and upper values (these are used as boundary values of the variable range), `isIdentifier` (i.e. true when the variable is an identifier of the class and otherwise it is false), `derivationFormula` (i.e. its contains the derivation formula when the variable is derived), test item and test cases where the variable is located. The variable class is a container of the `Data` values. During the test cases concretization, a variable may take a value from one of its sets of data.

The `Data` element models values of the variable. Each value is related a pattern and input type. An `Input_Type` element models the input type used to concretize the `Data` value for each variable.

A `Pattern` element models a regular expression required to validate the value assigned to the variable when the concretization process is done manually or automatically by using a web-based

generation. Examples of possible patterns may be an email pattern ("^[\\w-]+(\\.\\[\\w-]+)*@[A-Za-z0-9]+(\\.\\[A-Za-z]{2,})\$"), and the Spain post code ("^[1-9]{2}|[0-9][1-9]|[1-9][0-9])[0-9]{3}\$").

6.3 Transformations

The metamodels described above depict the information on the proposed test artefacts. The goal of this section is to define a process to obtain instances of the test metamodel from instances of the requirements metamodel, as well as instances of the test scenario metamodel from instances of the test metamodel. As mentioned at the beginning of this chapter, this process is modelled by means of transformations, which are relations oriented from a source toward a target metamodel, codified in ATL transformations. Figure 6.2 represents a global view of the transformations. All transformations are specified in ATL and implemented in a supporting tool through Java language. The transformations rules are introduced in the next sections and include specific metrics to ensure the quality of these transformations. ATL specification has many low-level details and a high-level representation of the transformation process structure is explained in the following sections.

6.3.1 Transformation from Requirements Model to Test Model

The goal of this transformation (T1 from Figure 6.2) is to obtain a test model conformed to the test metamodel described in the previous section.

The main task of this transformation is to invoke the 25 rules (mappings) depicted in Figure 6.7.

These rules are organized into eight groups: the first two generate the structure of the test model (i.e. test cases and precedence relations), R1 maps requirements RequirementModel to TestModel, R2_1 and R2_2 rules map each CommunicativeEvent or EventVariant to a TestCase.

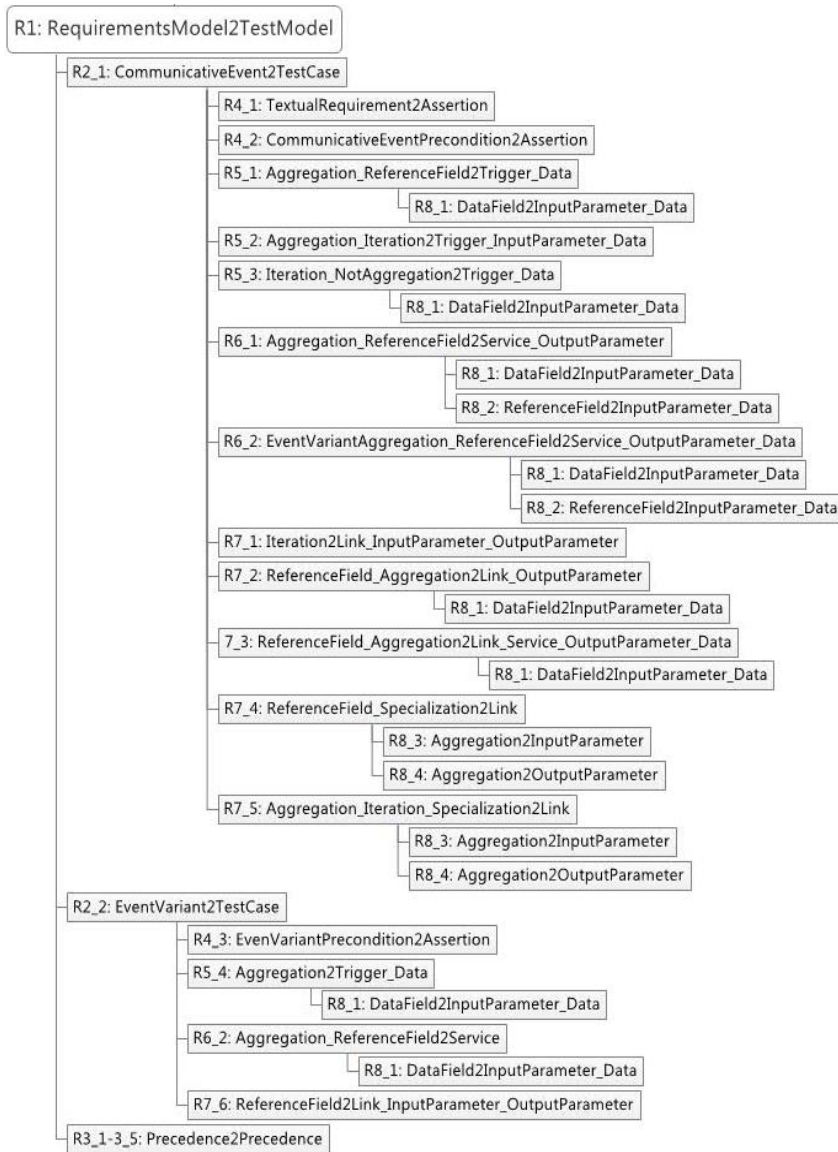


Figure 6.7. Structure of T1 Transformation

The third group of rules generates the Precedence relations, the rules R3_1 and R3_2 rule maps each Precedence between CommunicativeEvent and EventVariant to a Precedence between TestCase, and the rules R3_3, R3_4 and R3_5 map the relations with logical nodes (i.e. AND or OR) to nodes Precedence.

The fourth group of rules generates the test items Assertion in the respective TestCase. The rules R4_1 maps TextualRequirement to an Assertion. The rules R4_2 and R4_3 map a Precondition of a CommunicativeEvent or EventVariant respectively to a test item Assertion.

The following three groups of rules (R5-R7) derives the test items Trigger, Service and Link by analysing the properties of the structures Aggregation, ReferenceField, Specialisation and Iteration. The structures DataField, ReferenceField and Aggregation are mapped to instances of Parameter of the test items (i.e. Trigger, Service and Link). For generating test items we use the methodological core of the OO-Method proposed by Pastor [122], which is implemented into an object-oriented model-driven development framework with automatic code generation capabilities [6].

Tables 6.1-6.8 show the transformation rules between RM and TM. We construct them in such a way (base on steps) that they can be evaluated easily on model instances.

These transformation rules together specify the complete structural correspondence for an RM and its equivalent TM, which was validated by using the conceptual schemas derived from the Communicational Analysis requirements models by using the strategy proposed in España et al. [104].

Table 6.1. Transformation rules for generation of the Test Model

Group # 1. Generation of the Test Model		
Preconditions: none		
Steps		ATL rule
1	Create a TestModel with the requirements Model name.	R1

Table 6.2. Transformation rules for generation of the Test Cases

Group # 2. Generation of the Test Cases		
Preconditions: Class TestModel has already been generated.		
Steps		ATL rule
1	For each CommunicativeEvent and EventVariant in the CE diagram, draw a TestCase (TC) in the TestModel.	R2_1, R2_2
2	Do not draw the Start, End and Logical Nodes (i.e. Or, And) (if any).	

Table 6.3. Transformation rules for generation of the Precedence relations

Group # 3. Generation of the Precedence relations		
Preconditions: Classes TestCase have already been generated		
Steps		ATL rule
1	For each Precedence relation in the CE diagram, draw a Precedence in the Test Model. In case of a Communicative event with Event Variants, draw a precedence for each EventVariant and from each event variant, so that the test cases relate independently.	R3_1 R3_2
2	If there is a Precedence that starts from a Logical Node (i.e. Or, And) draw the Precedence from the previous CommunicativeEvent or EventVariant to the logical node until the next CommunicativeEvent or EventVariant.	R3_3 R3_4 R3_5

Table 6.4. Transformation rules for generation of the test items Assertions

Group # 4. Generation of the Assertions		
Preconditions: Classes TestCase have already been generated.		
Steps		ATL rule
1	For each TextualRequirement add an Assertion as a precondition in the respective TC.	R4_1
2	For each Precondition in a CommunicativeEvent and EventVariant add an Assertion as a precondition in the respective TC.	R4_2, R4_3

Table 6.5. Transformation rules for generation of the test items Triggers

Group # 5. Generation of the Triggers		
Preconditions: Classes TestCase have already been generated.		
Steps		ATL rule
1	For each Aggregation class that has a ReferenceField that extends a business object, add a Trigger in the respective TestCase and label it with the Aggregation name. If the substructure corresponds to a specialised CommunicativeEvent, a Trigger is derivate in each EventVariant. The Trigger type is 'set' and the owner is the domain name of the ReferenceField. Continue with Step 5.	R5_1
2	For each Aggregation class that has Aggregation children and no a ReferenceField a Trigger in the respective TestCase is derivate. The Trigger name has to correspond to the Aggregation parent name. The owner is the Aggregation parent name and the Trigger type is 'register'. An Input Parameter instance is created. The Parameter name is 'p_this' plus the name of the Aggregation parent. The Parameter type is the domain name of the ReferenceField. Continue with Step 5.	R5_2
3	For each Iteration substructure that has no Aggregation child, but has a ReferenceField that extends a business object a Trigger in the respective TestCase is derivate.	R5_3

	The Trigger name has to correspond to the Iteration name. The Trigger type is 'set' and the owner corresponds to the domain name of the ReferenceField. Continue with Step 5.	
4	For each EventVariant that has both a related Aggregation and a ReferenceField that extends a business object, a Trigger in the respective TC is derived. The Trigger name has to correspond to the name of the last parameter of the Aggregation related to EventVariant. The Trigger type is 'set' and the owner corresponds to the domain name of the ReferenceField. Continue with Step 5.	R5_4
5	After (1) and (2) an input Parameter instance is created. The Parameter name is 'p_this' plus the domain name of the ReferenceField. The Parameter type is the domain name of the ReferenceField.	
6	After (1), (2), (3) and (4). For each DataField contained in the substructure, Rule 8_1 is called.	

Table 6.6. Transformation rules for generation of the test items Services

Group # 6. Generation of the Services		
Preconditions: Classes TestCase have already been generated.		
Steps		ATL rule
1	For each Aggregation related with a CommunicativeEvent or EventVariant without ReferenceField, a 'new' Service has to be generated in the TC. The Service name has to correspond to the Aggregation substructure name. For each DataField instance in the Aggregation substructure, an Input Parameter instance has to be created with the domain value as its type. Therefore, the rule R8_1 (CommunicativeEvent) or R8_2 (EventVariant) is called. An Output Parameter has to be created with the Aggregation substructure name in lowercase.	R6_1

Table 6.7. Transformation rules for generation of the test items Links

Group # 7. Generation of the Links		
Preconditions: Classes TestCase have already been generated.		
Steps		ATL rule
1	For each Iteration substructure whose parent is an Aggregation substructure and its child an Aggregation substructure, a Link is generated between the parent Aggregation name (Input parameter) and child Aggregation substructure (Output parameter).	R7_1
2	For each ReferenceField within an Aggregation a Link is generated according to:	
2.a	If the ReferenceField does not extend a Business Object and there is no ReferenceField in the same substructure that extends a business object. The ReferenceField belong an Aggregation substructure related with a CommunicativeEvent or EventVariant.	R7_2

	The Link is between the domain name of the ReferenceField (Input parameter) and parent Aggregation name (Output parameter).	
2.b	If the ReferenceField extends a Business Object and there is another ReferenceField in the same substructure. The Link is between the domain name of the other ReferenceField (Input parameter) and the domain name of the ReferenceField (Output parameter).	R7_3
2.c	If the ReferenceField does not extend a Business Object and there is a parent Specialization substructure with child Aggregation. The Link is between the domain name of the ReferenceField (Input parameter) and Aggregation name (Output parameter). The Aggregation where the ReferenceField is excluded.	R7_4
3	For each Aggregation substructure whose parent is a Specialisation substructure and this parent has an Iteration substructure with Aggregation substructures, a Link is generated between the parent Aggregation (Input parameter) and Iteration child Aggregation (Output parameter).	R7_5

Table 6.8. Transformation rules for generation of the test items Parameters

Group # 8. Generation of the Parameters		
Preconditions: Classes Service, Trigger or Link have already been generated.		
Steps		ATL rule
1	A DataField generates an Input Parameter instance with the domain value as its type.	R8_1
2	A ReferenceField generates an Input Parameter instance with the domain value as its type. The name is formed by 'p_agr' plus domain value in lowercase.	R8_2
3	An Aggregation generates an Input Parameter instance with the substructure name as its type and name in lowercase.	R8_3
4	An Aggregation generates an Output Parameter instance with the substructure name as its type and name in lowercase.	R8_4

Table 6.9 shows a list of required and non-required RM metamodel constructs for test model generation. Each row describes a pair of constructs that match and their correspondence. Some metaclasses such as NODE (i.e. END, START) and LOGICAL_NODE (AND, OR) are informational resources. In the other hand, some metaclasses (e.g. ORGANISATIONAL_ROLE, ORGANIZATIONAL_ACTOR, and GOAL) required in the CA to model the requirements levels (i.e. L1, L4, and L5 see Section 3.1.1) but they are not used for our proposal and so are not mapped to the test metamodel instances.

Table 6.9. Requirements Metamodel constructs used in this transformation

Communication Analysis (CA) mapping	Test Model (TM) mapping	CA-TM correspondence mapping
Model	Test Model	1:1
Precedence	Precedence	1:n
Communicative Event	Test Case	1:1
	Assertion	1:1
Textual Requirement		1:1
Event Variant		1:1
	Test Case	1:1
Aggregation	Trigger	1:1
	Link	1:n
	Parameter(Output)	1:n
	Parameter(Input)	1:n
Reference Field	Link	1:n
	Parameter (input)	1:n
	Trigger	1:n
Data Field	Parameter (input)	1:1
Iteration	Link	1:n
Specialisation	Link	1:n
Node (End, Start)	-	Informational
Logical node (And, Or)	-	Informational
Communicative Interaction (ingoing, outgoing)	-	Informational
Organisational actor	-	Not used
Organizational role	-	Not used
Organisational goal	-	Not used
Organisational Location	-	Not used
Organisational Module	-	Not used
Strategy	-	Not used
Operationalisation	-	Not used
Goal	-	Not used
Communicative Role	-	Not used
Communicational Channel	-	Not used
Support role set	-	Not used
Organisational role set	-	Not used
Organisational Unit	-	Not used
Process	-	Not used
Indicator	-	Not used
Business object field	-	Not used
Business object class	-	Not used

The second part of this transformation modifies the Test Model by adjusting the Test Model precedence relationships. For the sake of readability, we use concrete syntax to describe instances of Requirements Model (RM), Test Model (TM) and Test Scenario Model (TSM) for Sudoku CS (see Figure 6.8). Figure 6.8 depicts the graphical concrete syntax of the models RM (see Figure 6.8a), TM (see Figure 6.8b) and modified TM (see Figure 6.8c).

Since a communicative event in RM can have more than one precedence relationship (see the communicative event 3 in Figure 6.8a), we modified the TM (see Figure 6.8b and Figure 6.8c) so that each node only has an input and output relationship except to the start and end nodes (i.e. only input or output relationship, but not both) as well as the predecessor node to a decision node (i.e. test case 4 in TM) or successor node to a logical node (i.e. test case 4 in TM).

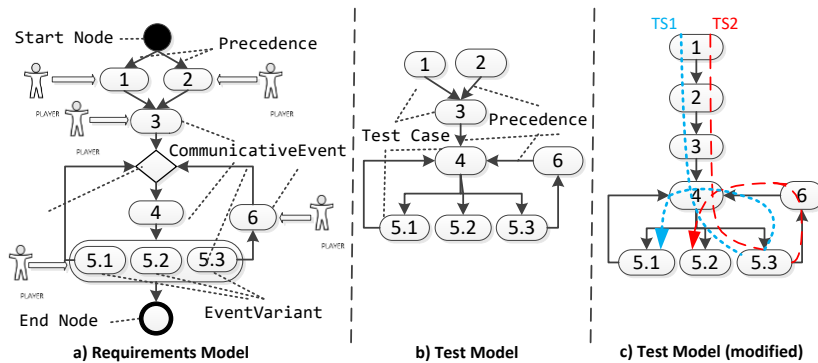


Figure 6.8. Examples using graphical concrete syntax of (a) RM, (b) TM, and (c) modified TM

Then, this transformation has been formalized in 25 ATL rules (see column ATL in Tables 1-8) and included in our CoStest tool (see Chapter 8). Figure 6.9 shows part of the related ATL code.

```

module ca2tm; create OUT: tcmetamodel from IN: cametamodel;
rule R1_RequirementsModel2TestModel{
from cametamodel : cametamodel!Model
to tcmodel: tcmetamodel!TestModel (
name<- thisModule.underscore(cametamodel.name),
location<-cametamodel.name + '(Req.Model)', trule<- 'R1',
testCases<- Sequence{cametamodel!Element->allInstances()-
>select(e|e.ocIsKindOf(cametamodel!CommunicativeEvent) or
e.ocIsKindOf(cametamodel!EventVariant))},
sequences<- Sequence {cametamodel!Element->allInstances()-
>select(e|e.ocIsKindOf(cametamodel!Precedence))} ) }

```

Figure 6.9. Example of the first rule of the ATL transformation CA2TM

6.3.2 Transformation from Test Model to Test Scenario Model

This second transformation consists of processing the `TestModel` obtained in the previous transformation by using 8 transformation rules grouped into two groups (9 and 10, see Tables 6.10-6.11) in order to generate the `TestScenarioModel`.

Table 6.10. Transformation rules for generation of the Test Scenario Model

Group # 9. Generation of the Test Scenario Model		
Preconditions: none		
Steps		Rule
1	Create the <code>TestScenarioModel</code> with the <code>TestModel</code> name.	R9

Table 6.11. Transformation rules for generation of the Test Scenario

Group # 10. Generation of the Test Scenario		
Preconditions: <code>TestScenarioModel</code> has already been generated.		
Steps		Rule
1	For each path in the <code>TestModel</code> a <code>TestScenario</code> is generated by grouping the respective <code>TestCase</code> . The test suite name is set to <code>'AbsTestScenario_'</code> + sequential number.	R10
2	For each <code>TestCase</code> in TM a <code>TestCase</code> is generated in TSM	R2'
3	For each <code>Assertion</code> in TM an <code>Assertion</code> is generated in TSM	R4'
4	For each <code>Trigger</code> in TM a <code>Trigger</code> is generated in TSM	R5'
5	For each <code>Service</code> in TM a <code>Service</code> is generated in TSM	R6'
6	For each <code>Link</code> in TM a <code>Link</code> is generated in TSM	R7'
7	For each <code>Parameter</code> in TM a <code>Parameter</code> is generated in TSM	R8'

This transformation aims to find all the possible scenarios from the test model. Our transformation is an implementation of a classic pathfinder or graph traversal algorithm using recursive functions in Java Language [107] to generate the Test Scenario Model. A `TestScenarioModel` consists of a set of `TestScenario`. Each `TestScenario` (i.e. model path) groups the corresponding `TestCase` with the respective `TestItem` (i.e. `Assertion`, `Trigger`, `Service` and `Link`). Figure 6.10 offers an overview of this transformation.

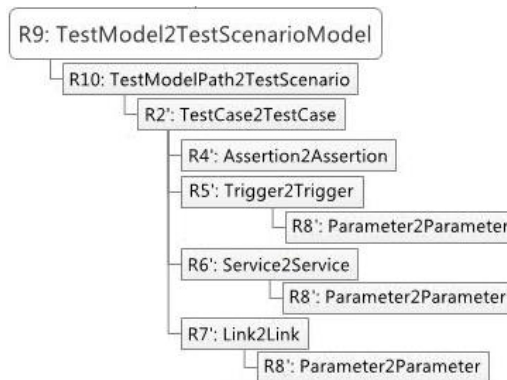


Figure 6.10. Structure of T2 Transformation

6.3.3 Transformation from Test Model to Test Data Model

The goal of this third transformation is to obtain a data model from the test model. The transformation entry point only aims to call the mapping shown in Figure 6.11.

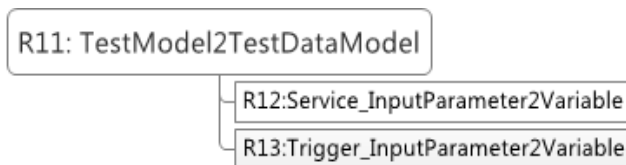


Figure 6.11. Structure of T3 transformation

The first direct mapping generates the test data model from the test model directly. The second mapping generates variables from input parameters related to Test Items of Service or Trigger type located in the different TM test cases.

6.3.4 Transformation from Test Scenario Model to Test Scenario Model with Abstract Test Cases

The goal of this fourth transformation is to obtain the test scenarios with abstract test cases from the test scenario model. The transformation is specified in Aceleo (see a partial view in Figure 6.12).

```

[comment encoding = UTF-8 /]
[module generateTScenarios ('http://atcmetamodel/1.0')]
[template public generateTScenarios
(aTestScenarioModel:TestScenarioModel)]
[comment @main/]
[for (tsc:TestScenario|aTestScenarioModel.testScenarios)]
[file (tsc.name+'_'+aTestScenarioModel.name+'.alf',false,'UTF-8')]
private import [aTestScenarioModel.name/]:*;
public import Alf::Library::BasicTypes::*;
public import Alf::Library::Asserts::*;
// Conceptual Schema under Test: [aTestScenarioModel.name/]
// Goal: Verify and Validate the Test Scenario: [tsc.name/]
// The Script consists of [aTestScenarioModel.testScenarios->size()]
Test Scenarios
activity [tsc.name+'_'+aTestScenarioModel.name/] () {
  [for (tcase:TestCase|tsc.testCases)]
    // Test Case: [tcase.name/]
    . . .
    [if tcase.testItems->selectByKind (Link)->size(>0)]
      // Links
      [if]
      [for (tl:Link|tcase.testItems->selectByKind (Link))]
      [tl.name/].createLink ([tl.parameters->selectByKind (Input).
name.toLower ()/]_, [tl.parameters->selectByKind (Output).name.toLower
()/]_);
      [ifor]
    . . .
  [ifor]
}
[/file]
[/for]

```

Figure 6.12. Partial Aceleo code of transformation

The transformation in Figure 6.12 invokes the mapping for every test scenario and creates a file (.alf) to contain the test items related to the test scenario, keeping the classification between test cases and test items (i.e. precondition assertions, services, triggers, links, postcondition assertions and invariants).

This transformation is a repetitive operation that traverses all the test scenarios from the test scenario model, creating a set of ALF scripts with abstract test cases (see Figure 6.13).

The test cases are abstracts in the sense that they do not contain concrete objects.

```

Scenarios with Abstract Test Cases
1 private import VideoClub::*;
2 public import Alf::Library::BasicTypes::*;
3 public import Alf::Library::Asserts::*;
4 // Conceptual Schema under Test : VideoClub
5 // Goal: Validate the Test Scenario: AbsTScenario_1
6 // The Script consists of 1 Test Scenarios
7 activity AbsTScenario_1_VideoClub () {
8   // Test Case: REGISTER_VIDEOCLUB_INFORMATION
9   // Services
10  videoclub_ = new VideoClub(p_atrid_videoclub=number,
11    p_atrmanager_name=text,p_atrcity=text,
12    p_atraddress=text,p_atrpostal_code=text);
13  ---
14 }

```

Figure 6.13. Test Scenario with abstract test cases

6.3.5 Transformation from Test Data Model and Abstract Test Cases to Executable and Concrete Test Cases

The goal of this fifth transformation is to obtain the executable and concrete test cases by merging the elements of the two prior transformations, relating test data model and abstract test cases to concretize the variable of the test cases.

Hence, this transformation takes both artefacts, a test data model and a scenario with abstract test cases test as inputs, and generates executable and concrete test cases as output by merging the information of the input artefacts.

Then, the mapping associates each variable of the test case statements with a concrete value from the test data model, if any. In addition, the assertions are added according to the type of test case (see Section 5.4.3). The results of this transformation are concrete and executable test cases.

Figure 6.14 shows an example of a concrete and executable test case for the Videoclub conceptual schema.


```

Executable and Concrete Test Case
1 private import VideoClub::*;
2 public import Alf::Library::BasicTypes::*;
3 public import Alf::Library::Asserts::*;
4 // Conceptual Schema under Test : VideoClub
5 // Goal: Validate the Object 'videoclub_' was created
6 //(test case positive)
7 // The Script consists of 1 Test Scenarios
8 activity AbsTScenario_1_VideoClub () {
9   // Test Case: REGISTER_VIDEOCLUB_INFORMATION
10  // Services
11  videoclub_ = new VideoClub(p_atrid videoclub=100,
12    p_atrmanager_name= "Jose Vicente Vidal",
13    p_atrcity= "Valencia", p_atraddress= "Guardia Civil 21",
14    p_atrpostal_code= "46020");
15  AssertTrue("Object Created",videoclub_ instanceof VideoClub);
16 }

```

Figure 6.14. Example of a concrete and executable test case for VideoClub CS

6.3.6 Transformation from UML CD-based CS to Executable CS under test

We use the ALF language as a notation for representing UML CD-based CS and for reasoning about this model. To obtain the result outlined in the previous section we defined a model-to-text transformation of UML to ALF, which we describe in this section. The mapping is specified as an ATL transformation included in the CoSTest tool and we outline here its points of interest.

Packages

Figure 6.15 shows the Aceleo transformation for a UML package such as the Video Club example depicted in Figure 5.6.

```

[file (p.name+'.alf', false, 'UTF-8')]
package [p.name/] {
  [for (aClass:Class|p.packagedElement->filter(Class))]
  [aClass.visibility/] class [aClass.name/][if aClass.superClass->size()>0]
  specializes [for(ac:Class|allParents())][if i>1], [if][ac.name/][for][if]
[/for]
  [for (assoc:Association |p.packagedElement->filter(Association))]
  [if assoc.isDerived=false and assoc.oclIsKindOf(AssociationClass)=false]
  [assoc.visibility/] assoc [assoc.name/];
  [/if]
  [if assoc.oclIsKindOf(AssociationClass)]
  [for (end:Property|assoc.memberEnd)]
  [assoc.visibility/] assoc [end.type.name/]_[assoc.name/];
  [/for]
  [/if]
[/for]
}
[/file]

```

Figure 6.15. Aceleo transformation rule for UML package

Classes

Figure 6.16 shows the partial ALF subunit translated for the class `VideoClub` of our example, where we can see the definition of the class attributes and part of the class constructor (i.e. `@Create`).

```
namespace VideoClub;
public import Alf::Library::PrimitiveBehaviors::StringFunctions::*;
public import Alf::Library::BasicTypes::*;
class Rental{
  public id_rental: Integer;
  public pick_up_date: Date;
  public return_date: Date;
  //derived attribute
  public total: Real;

  @Create Rental(
    // Corresponding to new_rental
    in p_atrid_rental:Integer,
    in p_atrpick_up_date:Date,
    in p_atrreturn_date:Date,
    in p_agrpartner:Partner
  ) {
    ...
  }
}
```

Figure 6.16. Partial definition for the class `VideoClub` by using ALF language

Associations

Figure 6.17 shows two examples of the ALF-based textual definition for associations. The first one (a) is the association between `Partner` and `Rental` classes and the second association (b) is the aggregation between `Rental` and `RentalLine` classes, which is transformed in a statement with a `compose` clause.

<pre>namespace VideoClub; assoc partner_rental{ public 'partner':Partner[1]; public 'rentals':Rental[*]; } </pre> <p>a) Translation of an association</p>	<pre>namespace VideoClub; assoc rental_rentaline{ public 'rental':Rental[1]; public 'rentallines': compose RentalLine; } </pre> <p>b) Translation of an aggregation</p>
--	--

Figure 6.17. Association and Aggregation of Order example using ALF language

Inheritance

Inheritance poses a particular problem in translating UML to ALF, since a subclass is dependent on its superclass, and this is an operation dependence, since creation of a subclass instance requires invocation of its superclass constructor. The inheritance relations are translated into ALF by using the `specializes` clause.

Figure 6.18 shows an example of inheritance relations translated for the `PrivatePartner` class.

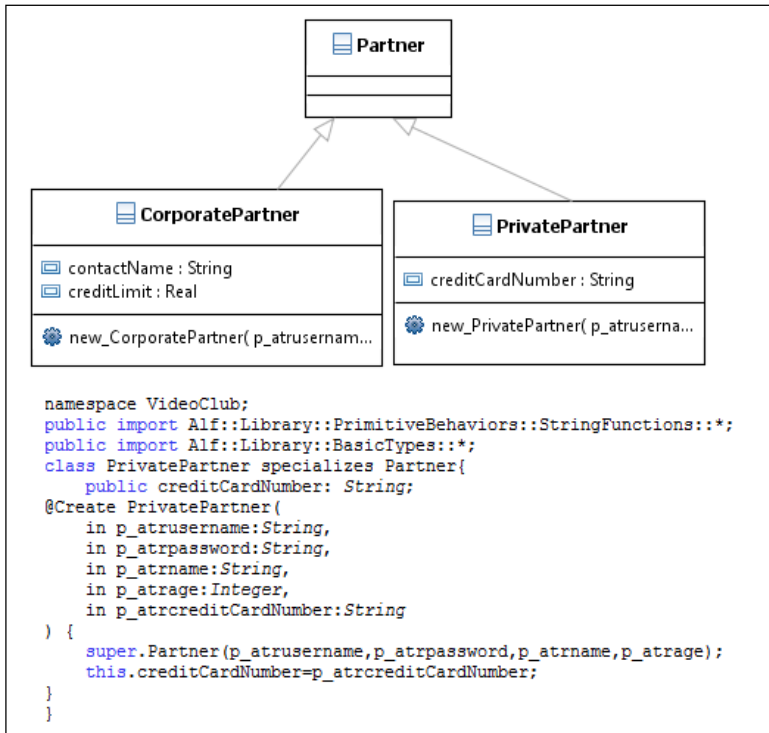


Figure 6.18. Partial view of the ALF unit including an inheritance relation

Constraints

Constraints are included in the UML models using mechanisms such as body, pre and post conditions. These mechanisms need to be translated into ALF elements to be executable. Depending on the role of the constraint, we generate a different scaffolding:

- body: If the corresponding operation is missing from the class model, we create a new operation and associated method.
- pre, post, inv: For each constraint we generate a new conditional associated with a side-effect free operation that returns an Error message when the constraint is violated. Bodies of other operations in

the model are changed in operations that check pre- and post-conditions of the operation and invariants of the class.

- **derive:** We create a getter operation (e.g., `property_<FeatureName>_derivation`). We attach the operation generated from the constraint expression to the getter and add a call for this operation in the class constructor. See derived Association in the next subsection.

- **def:** We create a new operation and associated method.

- **init:** We set the value of the property to the result of the compilation of the constraint expression in the class constructor.

Figure 6.19 shows a constraint attached to the class `Rental` of the VideoClub CS with the corresponding ALF code, which is translated to an operation of the `Rental` class.

```

namespace VideoClub;
public import Alf::Library::PrimitiveBehaviors::StringFunctions::*;
public import Alf::Library::BasicTypes::*;
class Rental{
    public id_rental: Integer;
    public pick_up_date: Date;
    public return_date: Date;
    //derived attribute
    public total: Real;
    . . .
    // derived attribute
    public property_total_derivation(){
        this.total=this.rentallines -> collect e(e.price) -> reduce Sum;
    }
}

```

} Alf code

Figure 6.19. Example of a constraint translated to ALF code

Derived Associations

For derived associations, we add an attribute to the class (e.g. `sequence`) and create a getter operation (e.g., `association_<DerivedAssociationName>_derivation`). We then attach the operation generated from the constraint expression to the getter. Figure 6.20 shows the attribute and method generated for the derived association `rentedMovies` of the VideoClub example (see Figure 5.6).

```

namespace VideoClub;
public import Alf::Library::PrimitiveBehaviors::StringFunctions::*;
public import Alf::Library::BasicTypes::*;
class VideoClub{
  public id_videoclub: Integer;
  public manager_name: String;
  public city: String;
  public address: String;
  public postal_code: String;
  public rentedMovies: Movie [] sequence;

  @Create VideoClub(
    // Corresponding to new videoclub
    in p_atrid_videoclub:Integer,
    in p_atrmanager_name:String,
    in p_atrcity:String,
    in p_atraddress:String,
    in p_atrpostal_code:String
  ) {
    ...
    this.association_rentedMovies_derivation();
    ...
  }
  ...
  public association_rentedMovies_derivation(){
    this.rentedMovies=this.movies->select e(e.status=="rented");
  }
}

```

Figure 6.20. Example of a derived association using ALF code

Association classes

The association class effect can be equivalently modelled with a class with two associations as shown in Figure 6.21. Therefore, we used this equivalence to transform an association class into ALF units.

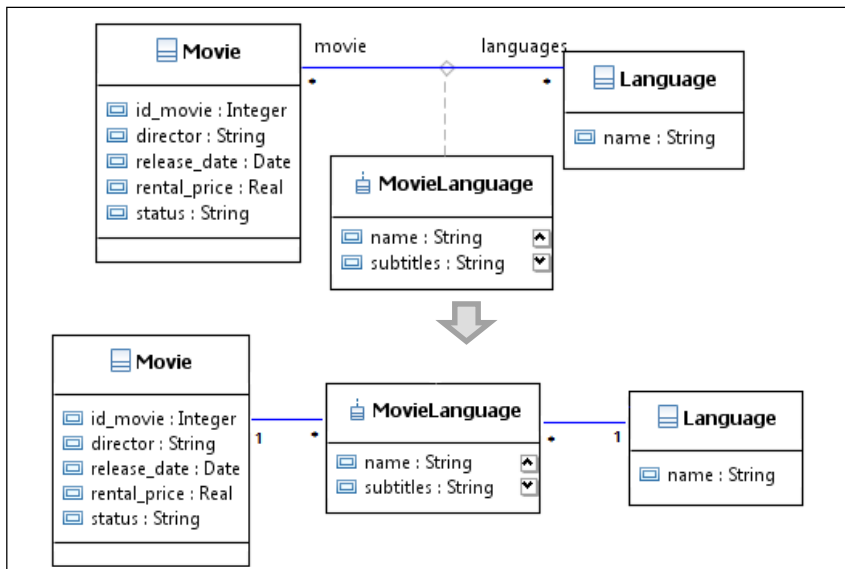


Figure 6.21. An example of class association

6.4 Summary and Conclusions

Developing model transformation definitions is expected to become a common task in model driven software development. Software engineers should be supported in performing this task by mature MDE tools and techniques in the same way as they are presently supported by classical IDEs, compilers, and debuggers in their everyday programming work.

In this chapter we have detailed the three metamodels and six transformations that we defined to implement our model-driven testing framework using the Eclipse Modelling Framework (<http://www.eclipse.org/modeling/emf>). For implementing the transformations, we used Java, ATL and Acceleo languages integrated into the Eclipse platform, one of the most popular development platforms in the software development community. These artefacts are required to implement the tool support described in Chapter 8.

In Chapter 8 the validation of the two main M2M transformations will be described and discussed.

PART IV.

TREATMENT

VALIDATION

Chapter 7

CoSTEST TOOL IMPLEMENTATION

Testing software would be extremely difficult without a reliable, fast and automated tool that runs the artefact software against a test suite, reporting the detected faults.

In Chapter 5, we proposed a validation framework for testing executable conceptual schemas. In this chapter, we summarise the prototype tool that we built to support the proposed validation framework.

CoSTest supports the generation, management and execution of automated tests against the executable conceptual schemas and makes the proposed testing framework feasible in practice.

This chapter is organized as follows: Section 7.1 explains the architecture and functionality of the CoSTest tool support. Sections 7.2-7.8 summarize individual tool functionalities. Section 7.9 contains a summary and conclusions of the chapter.

7.1 General Overview and Architecture

The main purpose of the tool is to support our testing-based framework described in Chapter 5 for validating CSs according to

stakeholders' requirements. CoSTest is a software tool that supports the generation, management and execution of test suites.

CoSTest works as standalone desktop application for Windows platforms and is available for downloading from the project website (<https://staq.dsic.upv.es/webstaq/costest.html>). Video tutorial with examples of its use may also be found on the project website, together with additional information and resources such as source files of requirements and conceptual schemas and complementary documentation.

CoSTest has been developed in the context of Design Science, which is the general framework of the present research work (Chapter 2). 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 laboratory experiments and case studies (Chapter 8).

Our tool may be used by testers/modellers/analysts in any development phase of a CS based on UML class diagrams. For example, as part of the test-last validation (i.e. correctness and completeness are checked by testing after the CS definition) or test-first development of conceptual schemas, in which the elicitation and definition is driven on a set of test cases.

The implemented release of the tool deals with schemas defined in UML class diagrams. Additionally, CoSTest is also able to deal with a representative set of constraints that involve two successive states of the modelled system (i.e. pre and post conditions), and on creation-time constraints (i.e. invariants and derived values). The definition of these additional features and their implementation are explained in Section 5.4.3.

We chose Eclipse (<http://www.eclipse.org>) as the technological platform and used the Eclipse Modelling Framework (<http://www.eclipse.org/modeling/emf>) to implement the

metamodels. Atlas Transformation Language
(<http://www.eclipse.org/at/>) and Aceleo Language
(<http://www.aceleo.org>) to implement the model transformations.

CoSTest's main features are as follows:

1. The generation of a **Test Model** from a Requirements Model.
2. The generation of a **Test Scenario Model** containing the Abstract Test Cases.
3. The generation and management of values for **Data Concretization**.
4. The generation and management of executable **Conceptual Schemas under Test**.
5. The generation and management of **Executable Test Cases** (scripts).
6. The execution of the test cases and the automated computation of **Testing Results**, which include verdicts, reports of defects and failing information as well as the automatic analysis of testing coverage according to a basic set of testing adequacy criteria.
7. The **Mutant Generation** of first order mutants for conceptual schemas, which are required to prioritize and validate the quality of CoSTest's test suite.
8. The **Batch Testing** allows the execution of the test suite and the automated computation of testing results for a set of selected CS.

The user interface of the CoSTest tool is implemented in Java Swing [123], assisted by a specialized tool to design graphical interfaces in Java, called JFormDesigner. The user interface of the tool is composed of seven tabs (see Figure 7.1) with one tab for each of the above features.

Figure 7.2 shows the main components of the CoSTest tool architecture. In the following sections, we describe the responsibilities and the implementation of each component.



Figure 7.1. Screenshot of the CoStest tool support

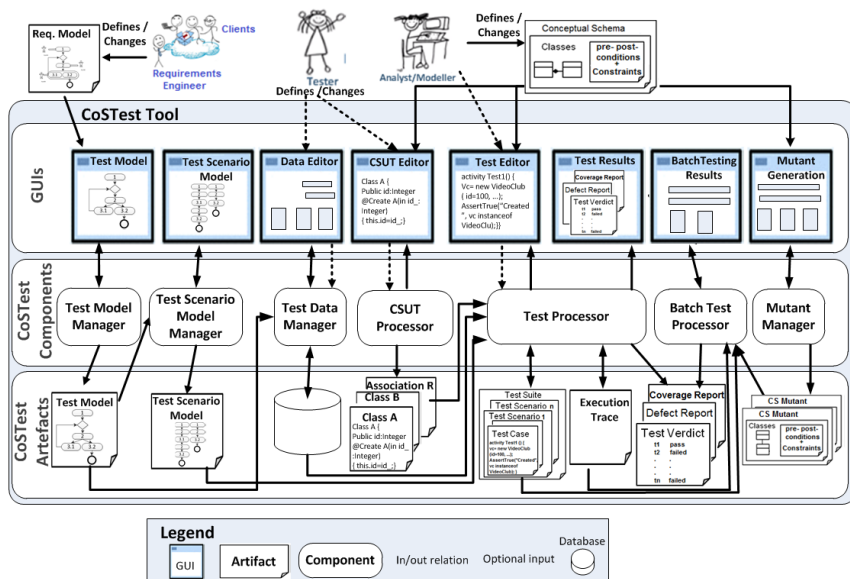


Figure 7.2. The CoStest tool architecture

7.2 The Test Model Manager

The Test Model Manager provides functionalities for generating and viewing the test model. Figure 7.3 shows the main components with a 3-layer architecture of the Test Model Manager, which consists

of the *Presentation Manager*, the *Test Model Generator*, Graph and Tree Builder, and the *Element Report Generator*.

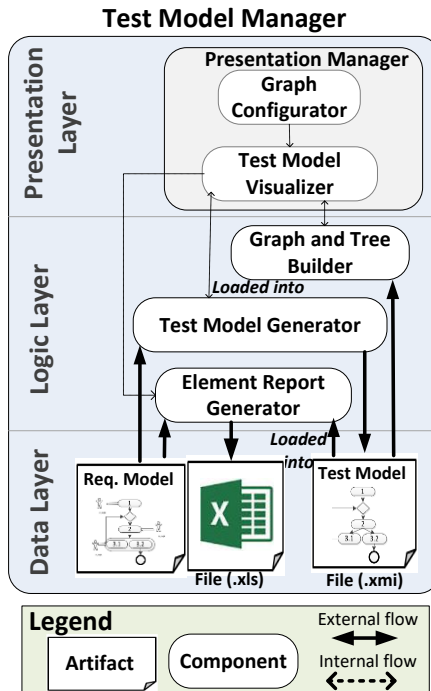


Figure 7.3. Test Model Manager design

7.2.1 Presentation Manager

The *Presentation Manager* implements two user interfaces: one related to the visualization configuration and another related to test model generation. The user interface for generating test models selects source and target files, which are saved as persistent files in a specified directory of the files system. The interface also provides functionalities to open existing test models, then the test model is presented as a graph and a visual tree. The graph view is provided by the JGraph and ListenableGraph libraries and the visual tree is implemented by XMLTreePanel library. The interface also includes a comboBox to select the CSUT type. Two options are available: (1) OO-Method conceptual model and (2) UML, depending on the derivation strategy of the CS elements. The interface also includes the following buttons:

- *Generate Test Model* to request the generation of the test model by the *Test Model Generator*.
- *Report Elements* to request a report of the generated elements as well the requirements model elements used in the transformation. This report helps to calculate the metrics presented in Chapter 8.
- *Graph Configuration* to call the interface to configure the visualization.

Figure 7.4 shows the user interface to configure the visualization, which can adjust the graph properties such as visualize the grid and the route tree edges, personalize the scale, change the distances between node levels, nodes as well as the node width.

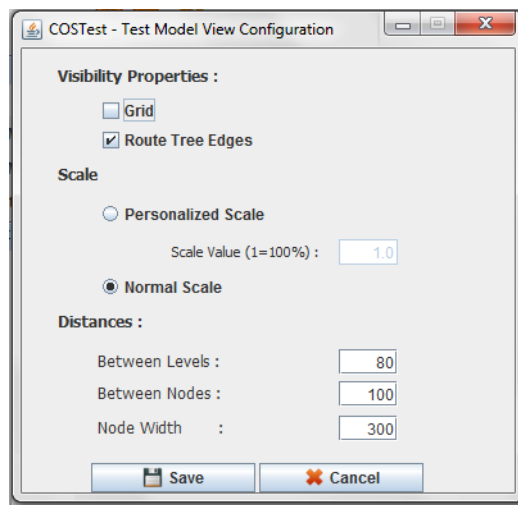


Figure 7.4. Screenshot with a test configuration example of the CoSTest tool

Figure 7.1 and Figure 7.5 show a CoSTest screenshot with the test model for the Video Club system using a visual tree.

7.2.2 Test Model Generator

Every time the user requests the generation of the test model, the *Presentation Manager* communicates with the *Test Model Generator* which executes the ATL model transformation (i.e. `ca2tc.asm`) in order

to generate the test model. Then, the test model is used by the Presentation Manager in order to show the result of the model transformation.

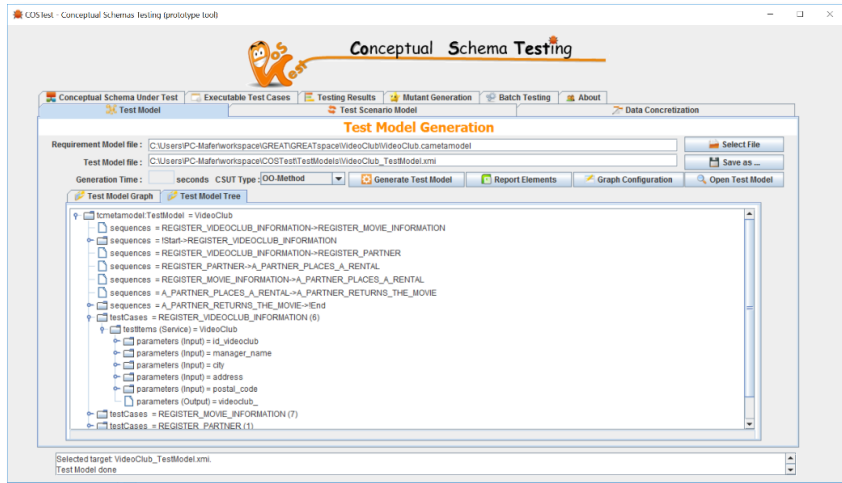


Figure 7.5. Screenshot with a test model example of the CoStest tool

7.2.3 Graph and Tree Builder

When the user requests the generation or the opening of the test model, the *Presentation Manager* communicates with the *Graph and Tree Builder*, which loads the test model in order to generate the respective views (i.e. graph and tree). Then, the test model is used by the Presentation Manager to show the result of the generation.

7.2.4 Element Report Generator

Every time the user requests the generation of the elements report, the *Presentation Manager* communicates with the Element Report Generator, which executes the query in both source files (the requirement model and test model) in order to generate the Excel report. Then, the report is saved as an Excel file by using the jxl library.

7.3 The Test Scenario Model Manager

The Test Scenario Model Manager implements the generation and the visualization of the test scenario model. Figure 7.6 shows the main

components with a 3-layer architecture of the Test Scenario Model Manager, which consists of the *Presentation Manager*, the *Test Scenario Model Generator*, Tree Builder, and the *Element Report Generator*.

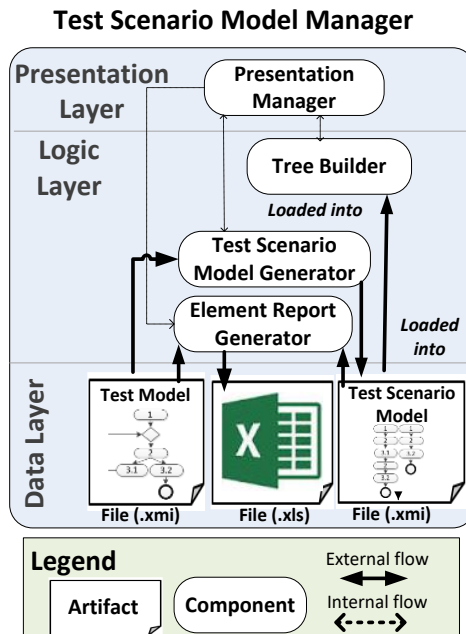


Figure 7.6. Test Scenario Model Manager design

7.3.1 Presentation Manager

The *Presentation Manager* implements only one user interface related to test model generation. The user interface can select source and target files, which are saved as persistent files in a specified directory of the files system. The interface also provides functionalities to open an existing test scenario model, then the test scenario model is presented as a visual tree. Additionally, the interface includes the following buttons:

- *Test Scenario Model Generation* to request the generation of the test scenario model by the *Test Scenario Model Generator*.

- *Report Elements* to request a report of the generated elements as well the requirements model elements used in the transformation. This report helps to calculate the metrics presented in Chapter 8.

7.3.2 Test Model Generator

Every time the user requests the generation of the test model, the *Presentation Manager* communicates with the *Test Scenario Model Generator*, which executes the Java model transformation to generate the test scenario model. Then, the test scenario model is used by the *Presentation Manager* to show the result of this transformation.

7.3.3 Tree Builder

When the user requests the generation or the opening of the test scenario model, the *Presentation Manager* communicates with the *Tree Builder* which loads the test model to generate the respective tree view. Then, the test scenario model is used by the *Presentation Manager* to show the result of the generation.

7.3.4 Element Report Generator

Every time the user requests the generation of the elements report, the *Presentation Manager* communicates with to the *Element Report Generator* which executes the query in both source files (the test model and test scenario model) to generate the Excel report. Then, the report is saved as an Excel file. Figure 7.7 shows a CoSTest screenshot with the test scenario model for the Video Club system using a visual tree.

7.4 The Test-Data Manager

The Test-Data Manager is able to setup a data base by creating, reading, updating and deleting test data values to concretize the test cases. A variable may be concretized with values by using (i) the requirements model, (ii) a manual entry, or (iii) a web-based generation.

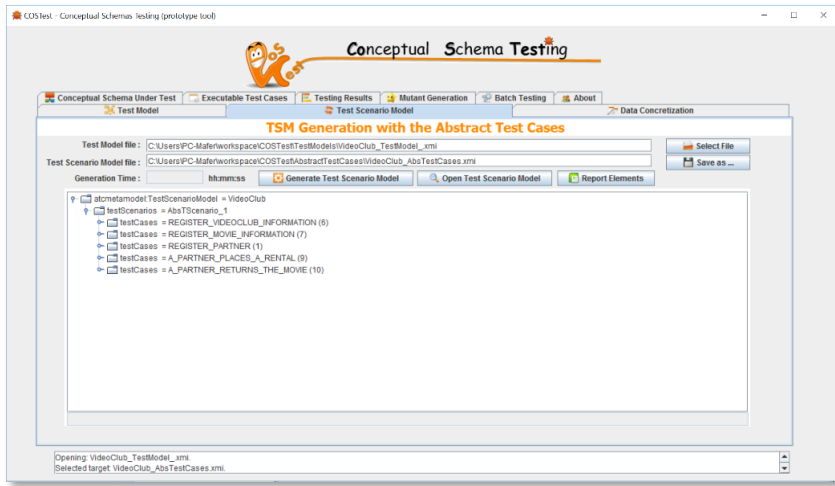


Figure 7.7. Screenshot of a test scenario model example in the CoStest tool

Figure 7.8 shows the main components with a 3-layer architecture of the Test-Data Manager, which consists of the (1) *Presentation Manager*, (2) the *Web-based Generator*, (3) the *Requirements-based Generator*, and (4) the *Database Manager*.

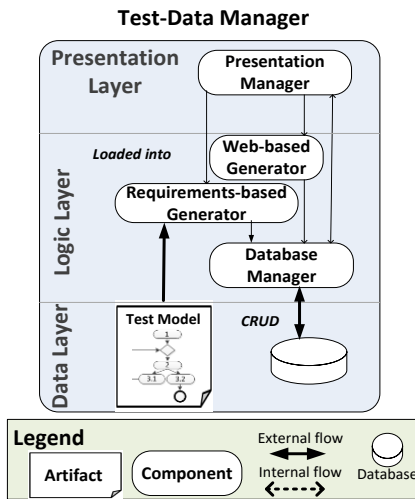


Figure 7.8. Test Data Manager design

7.4.1 Presentation Manager

Since the *Presentation Manager* implements an interface to support the CRUD functionalities (i.e. create, read, update and delete) on the test data, the test model filename is required as input. Then, the test data is presented as a list of variables with their properties (i.e. type, upper limit, lower limit, test item, test case, data source, related pattern data source type and concrete values).

When the user requires to concretize a variable with the values included in the requirements model, the user must click on the button “Generate from Model”. Then, the *Presentation Manager* communicates this request to the *Requirement-based Generator*.

When the user requires to concretize a variable with a manual entry, the user must (1) select the variable from list of variables, (2) select the “Manual Entry” option from the data source list, (3) select a pattern previously defined from the patterns list, and (4) click on “+” button located below the concrete values list. Then, the *Presentation Manager* enables the input controls to edit a concrete value for the selected variable.

Finally, when the user clicks on the save button, the *Presentation Manager* communicates the value entered, the pattern, the source type and the variable to the *Database Manager* to save the data.

The user interface also provides functionalities to support the CRUD functionalities (i.e. create, read, update and delete) on regular expressions (i.e. sequence of characters) that forms a search pattern for searching data on the web.

When the user requires to concretize a variable with values found in the Web, the user must (1) select the variable from the list of variables, (2) select the “Web-based Generation” option from the data source list, (3) select a pattern previously defined from the patterns list, and (4) click on “+” button located below the concrete values list, the *Presentation Manager* communicates the request to the *Web-*

based Generator in order to search for values to concretize on the Web.

7.4.2 Web-based Generator

When the *Presentation Manager* communicates a request to the *Web-based Generator*, it executes the Java module to search for the data on the Web by using the related pattern and the `org.jsoup.Jsoup` library. Then, the list of values found is passed to the *Database Manager* to save the result of the search.

7.4.3 Requirement-based Generator

Every time the user requests the generation of the values from the Model, the *Presentation Manager* communicates with the *Requirement-based Generator*, which uploads the test model file to retrieve the values related with each variable and then passes them to the Database Manager to save the loaded values.

7.4.4 Database Manager

This component is responsible for executing the commands or queries on the database in order to support the CRUD operations. Then, this information (i.e. value, pattern, variable with its properties) is returned to the *Presentation Manager* in order to refresh the information displayed on the interface.

The Database Manager is supported by Hibernate (<http://hibernate.org>, an Object/Relational Mapping (ORM) framework).

Figure 7.9 shows a CoSTest screenshot of the data concretization of the Video Club system.

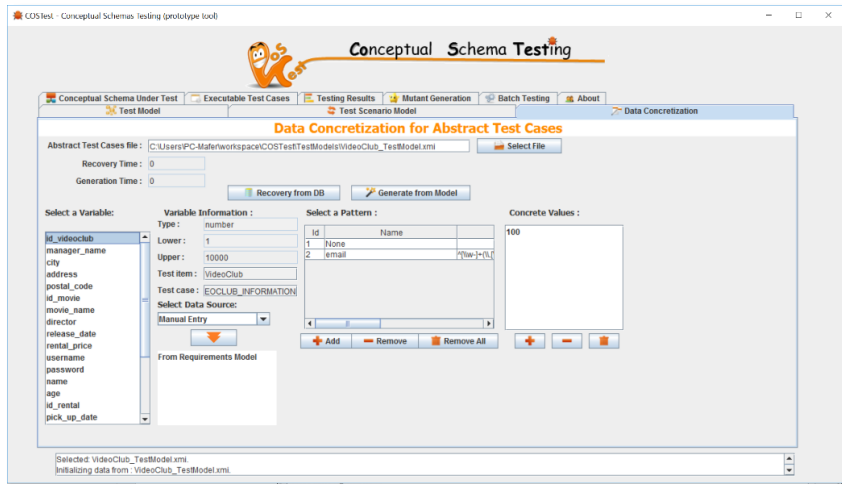


Figure 7.9. Screenshot for the data concretization in the CoSTest tool

7.5 The CSUT Processor

The *CSUT Processor* has the responsibility of transforming the UML-based Conceptual Schema into an executable format by using ALF language. UML relationships, constraints and classes with attributes and operations are transformed into ALF scripts. Details of the ALF Language and its grammar can be found in [82]. Figure 7.10 shows the main components with a 3-layer architecture of the CSUT Manager, which consists of the *Presentation Manager* and the *CSUT Manager*.

7.5.1 Presentation Manager

The *Presentation Manager* implements three user interface parts: the first is *Script Editor* tab, which is related to the management of the CSUT scripts (see Figure 7.11), the second the *Log Visualizer* tab, which provides the errors found by the parser in the syntax validation of the ALF scripts (see Figure 7.12), and the last one is the *CSUT Elements* tab, which reports the different elements identified in the conceptual schema (see Figure 7.13).

The user interface for managing ALF scripts can create (i.e. result of the generation), parse, edit and save the generated scripts.

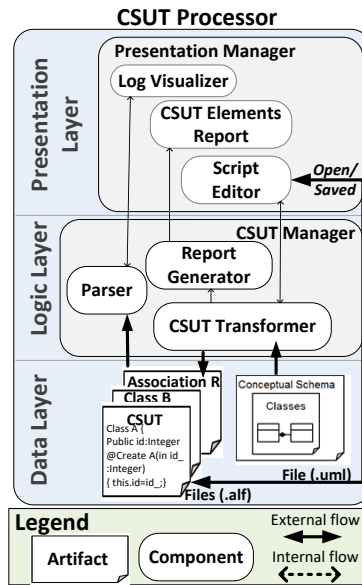


Figure 7.10. CSUT Processor design

The user interface can select source and target files, which are saved as persistent files in a specified directory of the files system. The interface also provides functionalities to open existing ALF scripts, then the file is presented as a text file in the *Script Editor*.

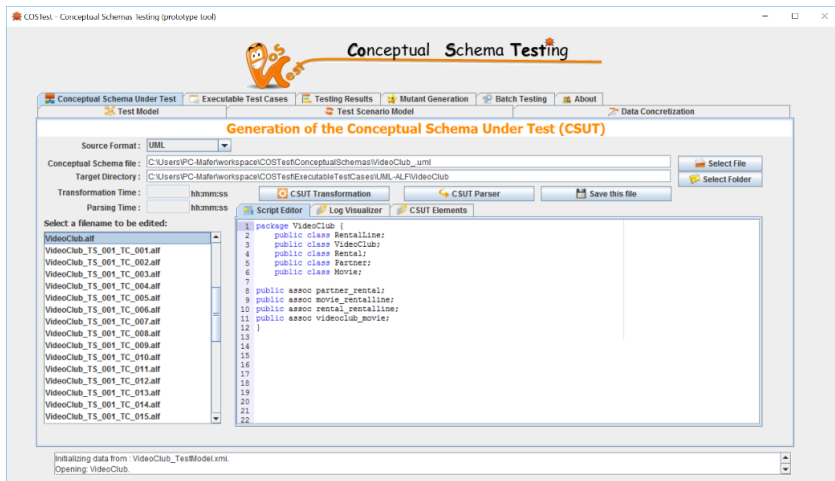


Figure 7.11. Screenshot for editing an executable CSUT in the CoStest tool

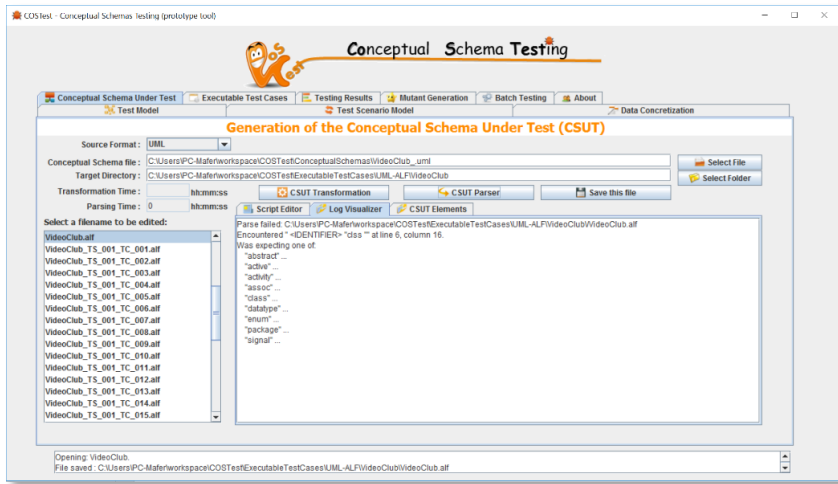


Figure 7.12. Screenshot for showing the parser results in the CoStest tool

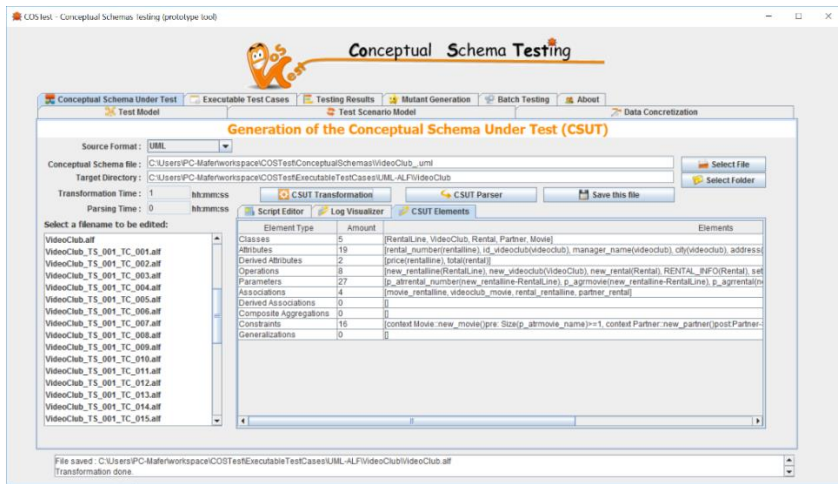


Figure 7.13. Screenshot for showing the CSUT elements in the CoStest tool

The *Script Editor* is implemented by using the JSyntaxPane library. JSyntaxPane provides resources to handle basic syntax highlighting and editing of various languages within the Java Swing application. Since JSyntaxPane does not include syntax highlighting for the ALF Language, we extended it to allow this. The interface also includes the following buttons:

- *CSUT Transformation* button to request the CSUT transformation from UML format (.uml) to executable format into an ALF script (.alf) by using the *CSUT Transformer*.
- *CSUT Parser* button to request the syntax validation of CSUT Scripts, which are transferred to the *Parser Executor*.

7.5.2 CSUT Manager

The *CSUT manager* has three main roles:

- 1) Read and transform the CSUT written in UML format (.uml) into an executable CSUT format (ALF script .alf). For this, the *Presentation Manager* communicates with to the *CSUT Transformer*, which executes the Java model transformation to generate the ALF scripts. Then, the ALF scripts are used by the *Presentation Manager* to show the result of this transformation in the files list shown in the interface as well as in the *Script Editor* Tab. For transformation, we use libraries such as `org.eclipse.uml2.uml`, `org.eclipse.emf`, `java.io.File`, and `java.io.FileWriter`.
- 2) Perform the execution of the *Report Generator* to list elements identified in the conceptual schema during the transformation.
- 3) Call the *Parser* in order to check the syntax of the CSUT. Then, the log generated by the parser is passed to the *Log Visualizer* of the *Presentation Manager* to show the result of the generation. If the log is empty no errors were found, otherwise the log reports the errors using the ALF report. Details of the ALF Language and its grammar can be found in [82].

7.6 The Test Processor

The *Test Processor* implements the generation, management and the execution of the test cases. Figure 7.14 shows the main components of the *Test Processor*, which consists of the presentation manager and the test manager, as described below.

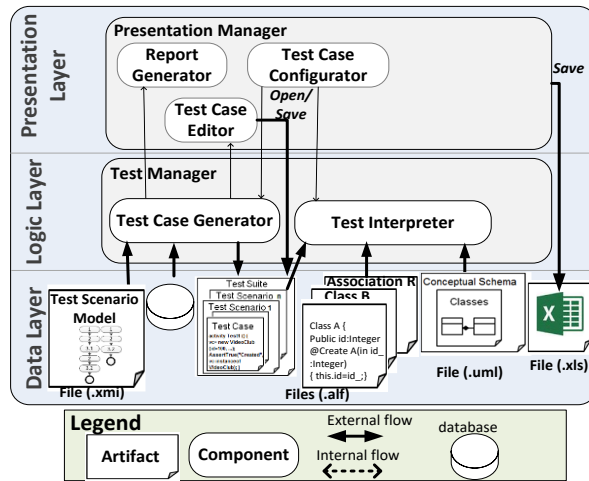


Figure 7.14. Test Processor design

7.6.1 Presentation Manager

The Presentation Manager implements three user interface parts: the one related to the configuration of test cases and testing process (Figure 7.15), the second related to the generation and management of the test suite (Figure 7.16), and the last one related to the presentation of the testing results (Figure 7.17).

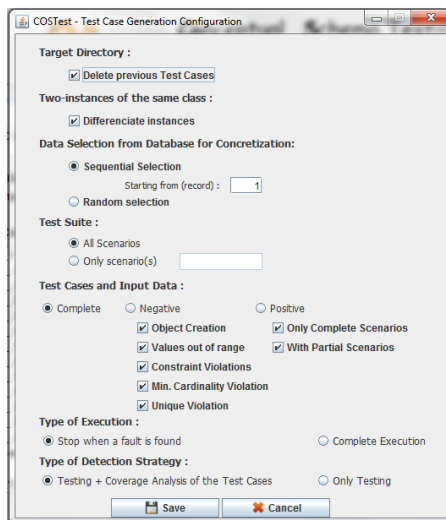


Figure 7.15. Screenshot of the test configuration in the CoStest tool

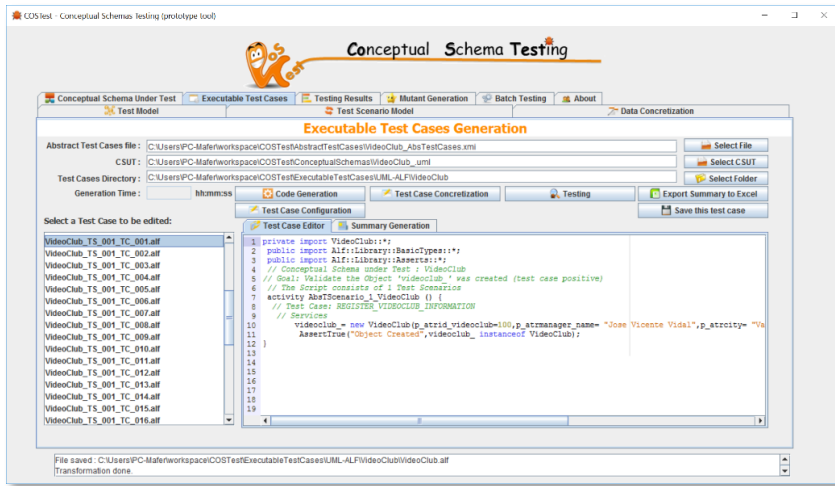


Figure 7.16. Screenshot of a test suite management example in the CoStest tool

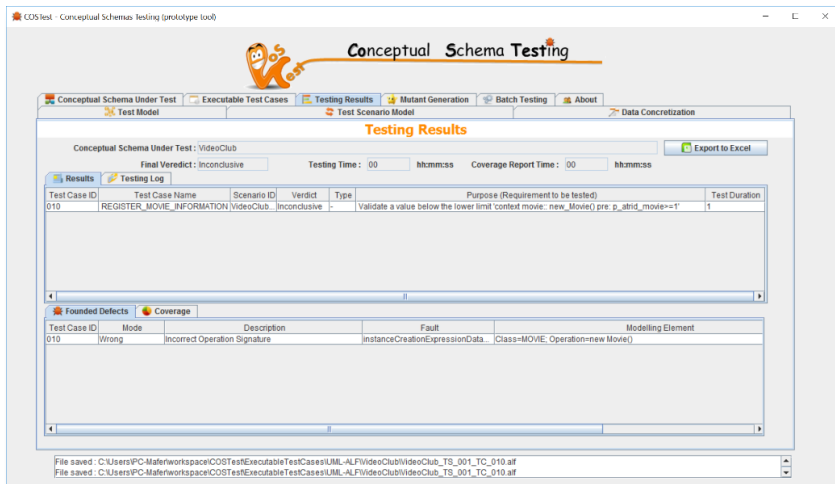


Figure 7.17. Screenshot of a test execution report in the CoStest tool

The user interface for managing test cases can generate, edit, save and report all information related to the generated test cases, which are saved as persistent files in a specified directory of the files system. For this, the interface implements buttons such as Select File, Select CSUT, Select Folder, Code Generation, Test Case Concretization, Save this test case and Export Summary to Excel.

The presentation manager includes the *Testing* button to request the execution of test cases selected from the test case list. After the execution, this module shows the faults (if any) in the *Testing Results* Tab (see Figure 7.17), the global verdict of the whole test suite, and the “Results” Tab that contains the detail of the verdict of all test cases in a tabular form. This information includes test case identifier, test case name, scenario ID, verdict, test case type, test case purpose, test case duration. All this information is collected, organized and transmitted to this component by the Test Manager. Figure 7.17 shows the result of the execution of a CoSTest test suite example on the Testing Results Tab, in which one test case has had problems in its execution, so that the global verdict is Inconclusive.

When the user clicks on the *Export Summary to Excel* button the *Presentation Manager* saves the report shown on the Summary Generation tab (see Figure 7.18) in an Excel file. This report contains details of test case types generated in each ALF script.

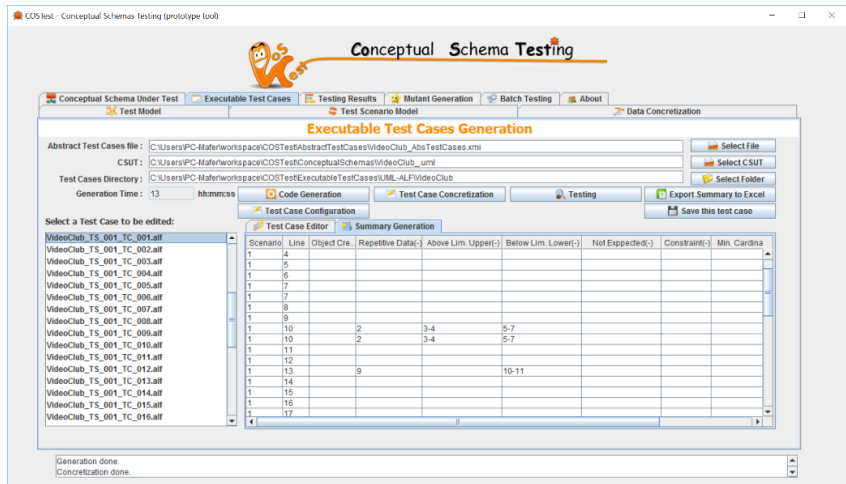


Figure 7.18. Screenshot of the Summary Generation tab of the CoSTest tool

Note that the table on the “Found Defects” tab (see Figure 7.17) indicates information about of the test case that fails or ends as inconclusive, such as test case identifier, the defect mode, the defect

description in natural language, the fault and the number of the lines where the fault has been revealed, and the modelling element. This information assists the modeller to point out the errors and faults.

Figure 7.19 also shows information about an execution log generated by the execution engine on the “Testing Log” Tab as well as the coverage report on the “Coverage” Tab. When the user clicks on *Export to Excel* the *Presentation Manager* generates an Excel file with details of the testing process (i.e. time report, results testing, found defects, coverage report, log report and CSUT element report) by taking the information from the different controls and tables on interfaces.

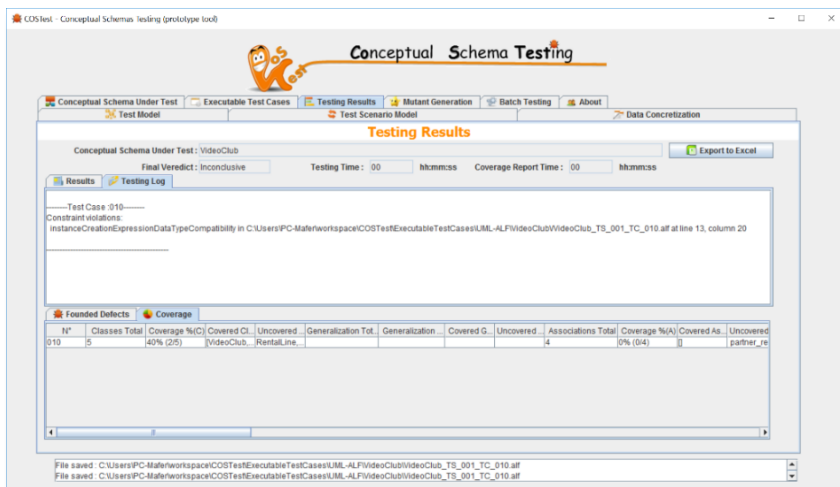


Figure 7.19. Screenshot of a log and coverage report in the CoSTest tool

7.6.2 Test Manager

The Test Manager has three main roles in the process of executing test cases: (1) the generation of the test cases (i.e. *Test Generator*), (2) the test cases execution, and (2) the collection and organization of the results to be shown by the Presentation Manager.

The generation of test cases consists of executing the model-to-text transformation written in Aceleo (see Section 5.4), then these

abstract test cases are concretized with the values taken from the data base. Details about the generation are collected, organized and transmitted to the Presentation Manager in order to generate a report by calling the *Report Generator*. Next the Test Manager collects all the test cases that are selected from the test suite (i.e. List), and requests its execution by the test interpreter. The individual results provided by the Test Interpreter are collected, organized and transmitted to the Presentation Manager.

The Test Interpreter has two main roles: (1) parse the test cases written in ALF language, and (2) perform the execution of the test cases specified in ALF scripts, as requested by the Test Manager.

CoSTest test cases can be executed from the command line using the *alf.bat* batch file (for Windows). The CSUT and test cases are compiled to an in-memory representation and executed using the fUML Reference Implementation. The result is an execution trace reporting faults (such as when a primitive behaviour implementation cannot be found during execution). Each assertion defined in the test case is evaluated by analysing the execution trace. The test verdict comprises the information on which assertions succeeded and which failed. When the test verdict is failed, the root causes are analysed in order to report the associated fault.

After the execution of all test cases, the report generator queries the coverage in order to obtain the sets of covered and uncovered elements of both CSUT and test suite respectively and computes information about coverage results.

7.7 The Mutant Generator

The *Mutant Generator* implements the computing, generation, and parsing of mutants for UML class diagrams. Figure 7.20 shows its main components, which consist of the *Presentation Manager*, and Mutant Manager.

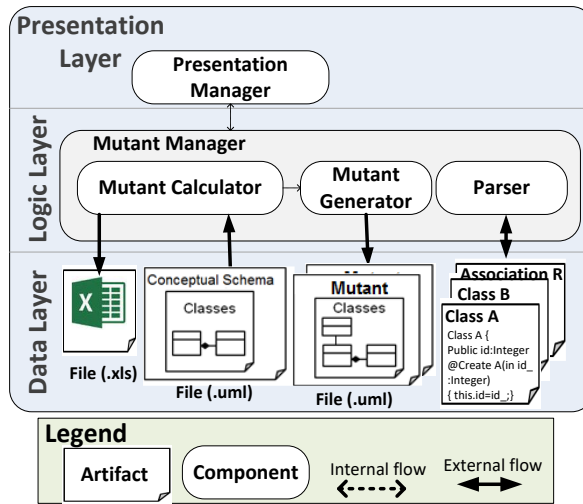


Figure 7.20. The Mutation UML tool architecture

In the following sections, we describe the responsibilities and the implementation of each component

7.7.1 Presentation Manager

The *Presentation Manager* implements the user interface related for the configuration and generation of mutants.

The interface can select a source CS file for the generation process, which are saved as uml files in a specified directory of the files system. The interface also includes two checkboxes for selecting between two options: (1) apply all mutation operators or select them individually and (2) generate all calculated mutant or select them individually.

The interface also includes the following buttons:

- *Calculate Mutants* to request the computation of the valid first order mutants from CS source by the *Mutant Calculator*.
- *Generate Mutants* to request the generation of the mutants selected from the previously calculated mutant list (by default all mutants are selected) by using the *Mutant Generator* component.

- *Parse Mutants* to request the parsing of mutants selected from the list by using the *Parser* component.

Export Results to Excel to request a report from the *Mutant Calculator* of the calculated valid and non-valid mutants.

7.7.2 Mutant Manager

Every time the user requests the calculation of the first order mutants, the Presentation Manager communicates with the Mutant Manager, which executes the Mutant Calculator to read the CS source and calculate the valid mutants that can be generated by applying the mutation operators selected by the user. This mutant list is used by the Presentation Manager to show it on the “Mutant Description Table” and can be exported as a report by pressing the “Export Report to Excel” button.

When the user requests the mutant generation, the Mutant Generator executes the Java code to generate the CS mutants (.uml) from the CS source file (.uml).

When the user requests mutant parsing, the Presentation Manager communicates with to the Mutant Manager, which executes the Parser to transform each selected mutant into an executable format by using ALF language. The ALF parser then produces an output with the analysis results of each mutant, which can be classified into valid and non-valid mutants. The working of the *Mutant Generator* can be seen in the partial view of a CS in Figure 7.21.

Five mutation operators have been applied to the CS. Four operators generate valid FOM (i.e. b) UPA2, c) WAS3, d) WCO3, e) MCO). However, applying the MAS operator to the WhiteCells association generates a non-valid FOM because there is a constraint (i.e. MovieUnique) that is related with the association.

Simply deleting the association would result in a Dangling constraint, which evidently is not desirable. Therefore, we need to add

more steps to the operator (going from FOM to HOM). The HOM should delete the association together with the respective constraint. This way, the mutant will not be detected by the parser and can generate a valid mutant for testing.

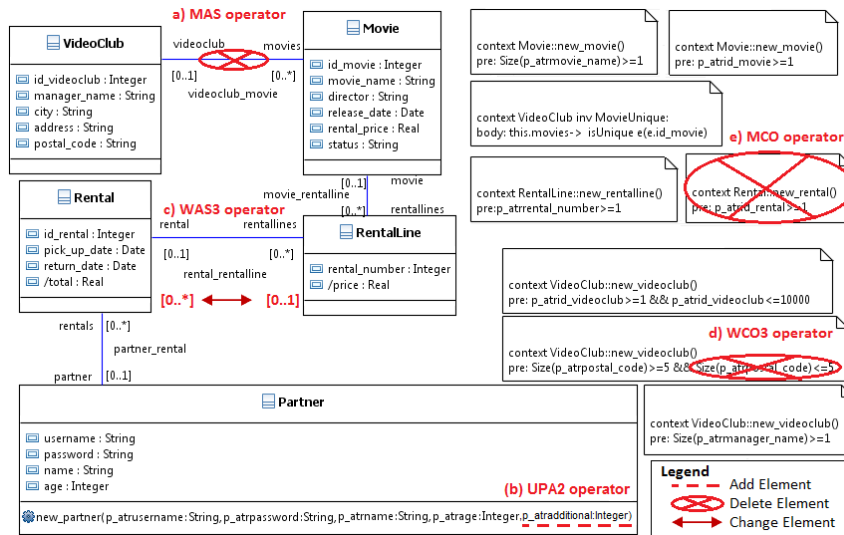


Figure 7.21. Application of five mutation operators for our CS example

The Mutant Generator had been presented as the MutUML tool [124] before being integrated into CoStest. This integration will allow us to conduct studies evaluating the effectiveness of CoStest test cases and facilitate making decisions (i.e. prioritize and select the test cases) based on analysis and interpretation of the results (see Section 5.5).

7.8 The Batch Testing Processor

The Batch Testing Processor implements the execution of the test cases for a group of mutants.

Figure 7.22 shows the main components of the Test Processor, which consists of the presentation manager and the test manager described below.

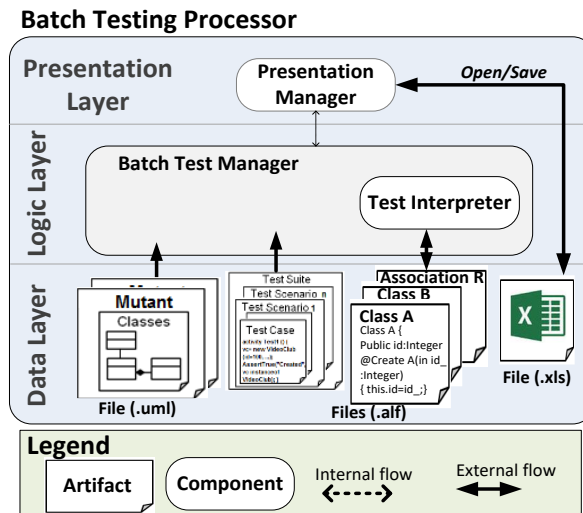


Figure 7.22. Batch Testing Processor design

7.8.1 Presentation Manager

The *Presentation Manager* implements the user interface (see Figure 7.23) for test mutants and enables both mutant directories and test cases to be selected, which are saved as UML and ALF files, respectively, in a specified directory of the files system.

The interface also includes a checkbox for parsing mutants or not (if not checked) previous to the testing process.

Additionally, the interface includes the following buttons:

- *Mutant Testing* to request the execution of the test cases against the mutants, which is passed to the *Batch Test Manager*.
- *Results Summarization* to request the report with the results of the testing process, which generates an Excel file summarizing the defects found in all mutants. For this purpose, the *Presentation Manager* reads the Excel file generated for each mutant and recovers the found defects only (if any).

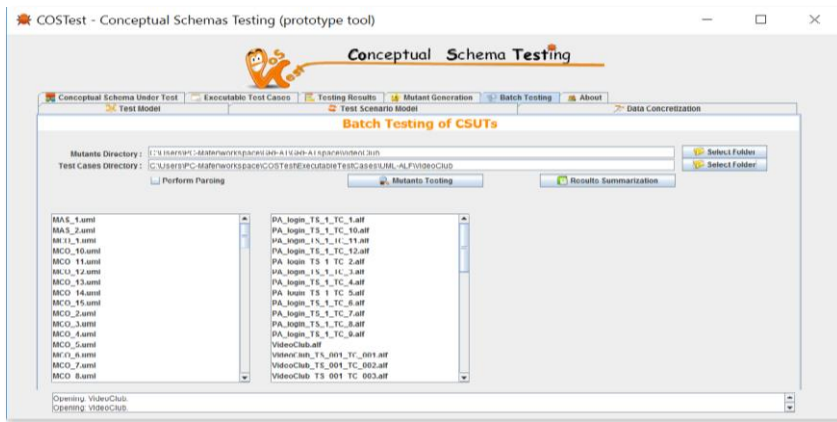


Figure 7.23. Screenshot for Batch Testing of the CoSTest tool

7.8.2 Batch Test Manager

When the user clicks on *Mutants Testing* the *Batch Test Manager* calls the *CSUT Manager* (see Section 7.5) to transform each mutant into an executable format (the parser is executed only if the checkbox is active). Then, the *Batch Test Manager* executes the test cases specified in ALF script by executing the *Test Interpreter* (see Section 7.6). In this option the Excel report is generated for each tested mutant automatically.

7.9 Summary and Conclusions

In Model-driven development it is very important to provide tools that support and promote the application of model-driven solutions.

In this chapter we have explained the fundamentals of the prototype tools that we developed to prove the feasibility of our approach and to support its validation. We implemented these prototypes as a standalone desktop application by using Java, ATL and Aceleo languages integrated into the Eclipse platform, which is one of the most popular development platforms in the software development community.

CoSTest focuses on the implementation of the model-driven testing framework presented in Chapter 5 to validate the correctness and

completeness of requirements of conceptual schemas. For this purpose, we implemented the transformation rules detailed in Chapter 6. Users (e.g. conceptual modelling researchers, modellers, testers, students and practitioners) considering or planning to conduct Conceptual Schema validation using a tool, as well as those interested in taking a systematic sound snapshot of the conceptual schema validation practice are the expected users of our tool.

Additionally, the tool implements a component in Java to generate first order mutants of Conceptual Schemas. This functionality helps to validate the effectiveness and adequacy of CoSTest test cases (see Chapter 8).

Further development will extend CoSTest to create a multiplatform support as the first step towards a tool supporting model-driven testing at the conceptual schema level.

In the next chapter, different validation and evaluation processes of the tool support will be presented and discussed.

Chapter 8

VALIDATION AND EVALUATION OF CoSTEST

According to the Design-Science Research (DSR) paradigm proposed by Wieringa [15], the validation of the designed artefacts produced as a result of the research process is crucial. So, the next step in our design cycle to develop our research project is the design validation.

The evaluation of the designed artefacts may rely on several methodologies available in the knowledge base such as observation (case studies, field studies, etc.), analysis (static analysis, architecture analysis, optimization, etc.), experimentation (controlled experiments, simulation, etc.), testing (functional black box, structural white box, etc.).

In [125], Shull et al. provide a basis for both understanding and selecting from the variety of methods applicable to empirical software

engineering. Following the criteria suggested by these authors, we selected experimentation as the method of evaluating several features of the result of our research (i.e. our validation framework and our prototype tool). An experiment is an investigation of a testable hypothesis where one or more independent variables are manipulated to measure their effect on one or more dependent variables. This methodology has been largely used in software engineering [126][127][128].

For the validation of our framework, we have performed several evaluations and validations throughout the development of our framework as summarized below.

For the validation of the UML-to-ALF transformation of the conceptual schema, we performed an experiment for the purpose of validating the effectiveness of CoSTest CSUT processor (Section 8.1.).

For the validation of the two first model-to-model transformations, we performed a comparative experiment (manual and automatic) for the purpose of validating them with respect to their syntactic and semantic correctness (Section 8.2).

For the evaluation of some properties of the mutation operators implemented in the CoSTest tool, we used a laboratory experiment (see Section 8.3.1). These mutation operators were used to prioritize the test cases (see Section 5.5) and to validate the effectiveness of our validation framework (see Section 8.4). Another laboratory experiment was performed for evaluating the efficiency and effectiveness in terms of the percentage of valid and non-equivalent mutants generated by the tool and the time that can be saved by using it (see Section 8.3.2).

For the validation of CoSTest's effectiveness, we performed a comparative experiment [15] of CoSTest test cases for detecting defects in both first order and high order mutant types. For that we used conceptual schemas of different sizes and domains (e.g. information systems, games). Among them is a real CS case that

conceptualises the Incident Management process in everis, a Spanish consultancy company. Some other CSs are well-documented cases that were found in the literature and others were selected because they contained the relevant CS elements required to inject the faults.

For evaluating CoSTest's usefulness and ease-of-use, we ran a pilot experiment prior to contacting real practitioners, and performed an observational case study [15] using interviews on CoSTest user perceptions. As a result, the everis testers evaluated the CoSTest tool as a very useful tool for validating information systems at the conceptual schema level. They recognise the usefulness of the tool and that opportune feedback given by our validation tool can support the quality assurance process of software and facilitate in making decisions based on analysis and interpretation of the results.

We believe that the results of these studies make the CoSTest framework strong and attractive to be transferred to industry.

This chapter is structured into two sections. Section 8.1 describes the experiment to evaluate the UML-to-ALF transformation of the CSs. Section 8.2 summarizes the experiment to evaluate the transformation rules used in the model-driven generation of CoSTest test cases. Section 8.3 describes the two experiments to validate and evaluate the mutant generation process. Section 8.4 describes a laboratory experiment to validate the effectiveness of the test cases of our CoSTest framework in detecting fault types for FOM and HOM sets of mutants. Section 8.5 describes an observational case study taken from industry to evaluate user perceptions in the correction process of the defects found on UML CD-based CS. We describe the design, procedure, results, conclusions, and lessons learnt. As a result, we improved the process, and the latest version of the tool is presented in Chapter 5 and the implementation in Chapter 7. Section 8.6 summarizes the conclusions of the chapter.

8.1 Validating the Effectiveness of CoSTest CSUT Processor

Model transformations are key elements of Model-driven Engineering (MDE). They allow querying, synthesizing and transforming models into other models or into code. However, it is very difficult and expensive (time and computational complexity) to validate in full the correctness of the model transformations.

Validation clarifies the question “Is the transformation right?” by allowing modellers and designers to test if the transformation behaves as expected. Intuitively, the validation of a transformation consists of exercising the transformation to certify that it works for a selected set of input models and compare the result with the expected outcome [129], without trying to validate it for the full input space [130]. Although such a certification approach cannot fully prove correctness, it can be very useful for identifying bugs in a very cost-effective manner.

In this section we present a laboratory experiment performed as part of our research to demonstrate the effectiveness (i.e. ability to be successful and produce the intended results) of CoSTest CSUT Processor using uml-to-alf transformation rules for obtaining ALF-based CS from UML CD-based CS.

The experiment was an iterative process in which we evaluated the V0.5 transformation rules, which were evolved until achieving the stable version V1.0.

8.1.1 Experimental Design

The experiment was performed by the author of this PhD thesis in a controlled environment using different CSs in an iterative process; approximately 10 iterations in one year (from June of 2014 to June 2015) with the objective of demonstrating the effectiveness of the CoSTest CSUT Processor.

Subject CS

Most of the input CSs used in this experiment were small UML/ALF-based models in the form of ten CSs containing a variety of characteristics that can be present in UML CD-based CS, including classes, relations (i.e. association, composite aggregation, and generalization) and different types of constraints (i.e. pre-condition, post-condition and body condition). These CS were of different sizes and domains (e.g. information systems, games). One case was taken from industry (i.e. IM), while other CSs were found in the literature (i.e. [131], [132], [133], [134] and [135]). The different CSs were specified using UML2 and Papyrus⁷ tools. Table 8.1 summarizes their characteristics.

Table 8.1. Elements of the Subject Conceptual Schemas

Element	VC	MT	SG	ER	OCR	SS	PA	OC	DBLP	IM
Classes	5	6	11	7	10	9	15	20	17	6
Attributes	19	26	26	36	61	44	43	33	59	29
Derived Attributes	2	0	6	6	1	1	33	27	21	0
Operations	8	13	19	24	16	32	30	24	32	13
Parameters	27	43	48	75	77	91	82	50	80	51
Associations	4	5	6	8	10	9	19	13	10	4
Derived Associations	0	0	2	0	0	0	0	0	4	0
Composite Aggregations	0	0	3	0	0	0	0	1	4	0
Constraints	16	9	19	21	14	12	45	24	44	8
Generalizations	0	0	4	0	3	0	0	10	13	0

A brief description of each CS is as follows:

1) Video Club (VC) CS represents the functionality of a chain of video stores to manage movies, partners and movie rentals.

2) The Medical Treatment (MT) CS defines part of the CS (of a Medical Treatment business process) of the fictional Santiago Grisolia University Hospital, developed by España et al. [131].

⁷ <https://eclipse.org/papyrus/>

3) The Sudoku Game (SG) CS was developed by Tort and Olivé [133] as an object-oriented CS of the Sudoku Game system. This CS defines the functionality for managing different users, playing with their Sudokus and generating new ones.

4) The Expense Report (ER) CS defines the functionality of an information system to manage the expense report life cycle of a business. This CS deals with several entities such as departments, employees, projects and expense types.

5) The Online Conference Review (OCR) CS, which is based on the description of the CyberChair System [136], defines the functionality of an information system to deal with members (committee chair and program committee) of a conference, as well as authors that submit papers to be evaluated for inclusion in the conference proceedings.

6) The Super Stationery (SS) CS defines the information system of a company that provides stationery and office material to its clients. This CS was developed by España et al. [132].

7) The Photography Agency (PA) CS makes use of classes, associations and constraints but has no generalizations and derived associations to define the information system that manages photographers and their photographic reports for distribution to newspaper publishers.

8) The osCommerce (OC) CS specified by Tort [134] represents all the essential structural and behavioural knowledge needed to perform the main user functionalities of the osCommerce system when placing an order.

9) The Digital Bibliography & Library Project (DBLP) case contains parts of the conceptual schema of the DBLP system [135], a computer

science bibliography website⁸, which deals with persons (authors and editors) and their publications.

10) The Incident Management (IM) CS defines the functionality of an information system to solve the incoming incidents (reception, process, allocation process and resolution process). This CS is a real case taken from Everis Company⁹, a multinational firm offering business consulting, as well as development, maintenance and improvement IT.

Procedure

Figure 8.1 illustrates the i -th iteration of the experiment. Each subject CS was transformed from UML to ALF using the CoStest UML-to-ALF transformation rules version V0.5. Then, the fUML virtual machine (executed from CoStest) was used to parse the CSs.

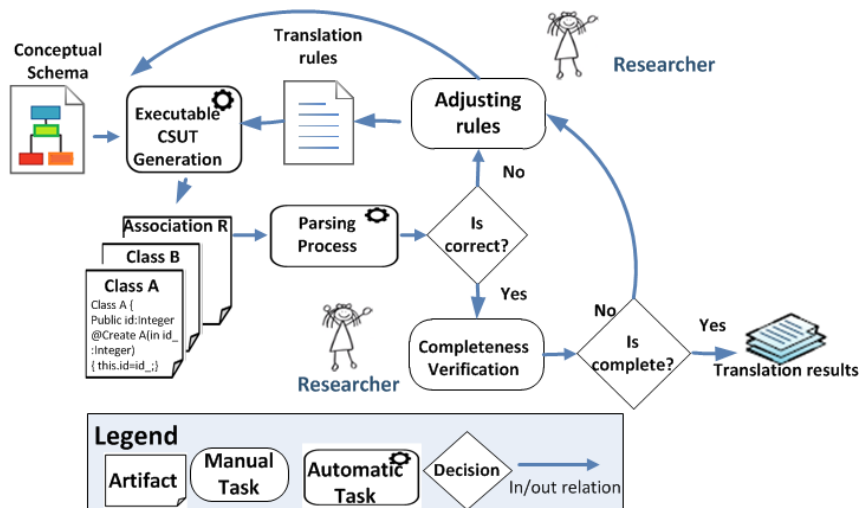


Figure 8.1. i -th iteration of the experiment applying the CoStest tool

If the result was incorrect, the transformation rules had to be adjusted and the process was then re-run. If the result was correct, the researcher reviewed the generated code for each CS, and compared

⁸ <http://www.informatik.uni-trier.de/~ley/db/>

⁹ www.everis.com

the ALF units generated with source elements to evaluate the completeness of the CS. The last iteration of the experiment was used to exemplify the CoSTest tool.

8.1.2 Conclusions and Changes on the CoSTest CSUT Processor

The experiment let us validate the UML-to-ALF transformation rules of the CoSTest CSUT Processor by verifying the syntactic correctness and evaluating the completeness of the transformed CS (100% in syntactic correctness of our generated CSUTs and 100% in completeness). These results suggest that these translation rules are effective in generating ALF-based CSUT. However, the behaviour of the translation may depend on the characteristics of the CS they are applied to, such as the CS element types (see Section 5.6.1) and syntactic correctness of the CS.

Some of the main changes applied to the transformation rules were the restrictions included in the operation and constraint names, for example, the constructor operation name should begin with “new_” and the constraint name for a derived association should be “association_<DerivedAssociationName>_derivation”. Some of the main changes applied to the CoSTest tool were: (i) include the facility for visualizing the parser log with syntax defects, (ii) structure a report containing element type, amount and translated CS elements. All reports were exemplified using the ten analysed CS.

8.2 Validating the CoSTest Transformation Rules

In this Section, we validate the model-to-model transformations by means of a comparative experiment between the generated results in five CS specifications with the expected outcomes. Measuring our model transformation entails evaluating correctness across the following two dimensions: (1) Semantic Correctness of transformations (mapping algorithms) is accomplished if for each simulation sequence of the source model we find a corresponding simulation sequence in the target model [137] i.e., the elements generated are equivalent to

the requirements model from which these elements are mapped. (2) Syntactic Correctness of the generated elements is achieved if given a well-formed source model, the target model generated by the transformation is a well-formed instance of the target metamodel [138], whether individual values of these elements are appropriate locally (e.g., for a component) as well as globally (e.g., for all dependent components). For this purpose, we proposed a set of metrics to measure the semantic and syntactic correctness of the proposed transformations as well as each one of the used transformation rules.

Once the metamodels and transformation rules had been specified (Chapter 6), and the information to trace the elements added to the metamodels, the transformation rules were evaluated on the instance models after each execution of the transformation. These rules can be evaluated by performing a classic pathfinder or graph traversal algorithm on the instance models, and checking if the transformation rules are satisfied at each transformation. This process is as follows:

1. We generate the code that traverses the instance models and reports the transformation rules used for generating the model constructors. Since the metamodels of both the source and target models are available with the transformations, and the trace information is included in the metamodels in attribute form that can be checked automatically, the model traverser code was defined from the transformation rules specification. This needs to be done only once each time the rule specification changes. This code was included in our CoSTest tool (see Chapter 7).
2. We call the model traverser code (i.e. include in the CoSTest tool) at the end of each execution of the transformation, supplying to it the source and target model instances with the trace information. In the case of the RM to TM transformation, we traverse the input requirements model and evaluate the transformation rules at each node (i.e. communicative event)

and precedence relation. For each Communicative Event, the trace information is checked to find the corresponding Test Case as well as at precedence level (see the group of rules 1-3 in Section 6.3.1). In the second transformation from TM to TSM transformation, we traverse the input test model and evaluate the transformation rules at each test scenario (i.e. path) and their test cases. For each Test Scenario, the path is checked to find the corresponding path in the test model. Then, the rest of the internal elements of each test case generated by the transformations in the two models are compared with the expected elements by applying manually the transformation rules.

3. We check if the generated elements conform to the respective metamodel (i.e. syntactic correctness), otherwise a syntactic problem is found. We also verify if the information assigned to each attribute is well-formed, otherwise the ATL transformation rule is wrong. Thus, if some element is absent that should be present, the rule is missing in the ATL implementation. On the other hand, if a TM element has been defined for an RM element, and no corresponding transformation rule is found, then this signals an unnecessary ATL rule implemented in the model transformation.

Finally, after locating the corresponding assignments, these are evaluated. If all the rules are satisfied for all the model nodes, then we can conclude that the transformation has been executed correctly. If any of the rules are not satisfied, the problem is reported by using the metrics described in the next Section.

8.2.1 Definition of Basic and Derived Metrics with Rule Scope

The basic metrics have been defined considering the elements of the two metamodels of our proposal. The basic metrics with rule scope are shown in Table 8.2.

Table 8.2. Basic metrics for Semantic and Syntactic Correctness of a Rule

Metric	Definition
N_EG_j	Total number of elements generated by the rule j
Syntactic Correctness reached by a Rule j	
N_CEG_j	Total number of Correct Elements generated by the rule j
Semantic Correctness reached by a Rule j	
N_EE_j	Number of expected elements to be generate by the rule j

The respective derived metrics are listed in Table 8.3. The values of these metrics are only at the element level, so that they do not consider the contained elements.

Table 8.3. Derived metrics for Semantic and Syntactic Correctness of a Rule

Metric	Definition	Formula
SyC_{rulej}	Syntactic Correctness reached by the rule j	N_CEG_j / N_EG_j (1)
SeC_{rulej}	Semantic Correctness reached by the rule j	N_EG_j / N_EE_j (2)

8.2.2 Definition of Basic and Derived Metrics with Transformation scope

In order to evaluate the correctness of the model transformations we adapted Yue and Ali's proposal [139], which involves an MOF-based framework for defining metrics to measure the quality of models.

As in the metrics with rule scope, we defined metrics to calculate the syntactic and semantic correctness of the whole transformation by considering that the result of the execution of a rule depends on the outcome of other nested rules (e.g. From Tables 6.5-6.7 the Rules 5, Rules 6 and Rules 7 are containers of Rules 8). To do this, we first define the following relevant concepts and variables: An **AtomicRule** is a rule that does not contain any reference to any rule including self-references; otherwise it is a **CompositeRule**, e.g. Rule 6 is an instance of CompositeRule because its result depends on the outcome of Rule 8. On the other hand, Rules 3, 4 and 8 are instances of an AtomicRule.

The **syntactic correctness** of a transformation (SyC_T#) is measured by the syntactic correctness achieved by the respective rules in the target model. An *ATrule* measures the correctness of an atomic rule, while a *Crule* measures the correctness of the composite rule, which depends on both values, the *ATrule* of its nested rules and its own *ATrule* value.

If an atomic rule generates a correct element, then its *ATRule* value is 1; otherwise its value is 0. Since TM has a hierarchical structure (see Figure 6.3) SyC_T is calculated starting from the most nested level of the structure (i.e. rules 3, 4 and 8) up to the highest level (i.e. Rule 1). The syntactic correctness value for Atomic Rules 3, 4 and 8 corresponds to their *ATrule* value. Finally, we have defined values *Wl*, which denotes the weight (i.e. value range between 0 and 1) assigned for each element *l* of the target model, so that it allows differentiating the impact of each model construct type on the correctness of the transformation. Notice that the sum of the weights should always be equal to the number of weighted model elements corresponding to the same level in the hierarchy. If the user does not assign any weights, then all weights are automatically assigned to 1. These derived metrics are as follows:

- **Syntactic Correctness for rule *i***, which generates the target model element *k* (*Cruleik*). The rule *i* is formed by the rules *j*, which generate the target model elements *l* (i.e. Parameter=Pr, Service=S, Trigger=T, Assertion=A, Link=L, Test Case=TC, Precedence=Pr, Test Model=TM, Test Scenario Model=TSM). The *j* depends on element type *l*, for instance, if *l* corresponds to the `Link` element then rule *j* can be R_7_1 - R7_5 (see Table 6.7).

$$Cruleik = \frac{\sum_{l=1}^{\#elem_in_k} Wl * [ACrulejl | ATrulejl]}{\#elements_in_k} \quad (1)$$

- **Average Composite Syntactic Correctness for rule *i***, which generates the target model element *k*. *ACrule* is equal to an *ATrule* at leaves level (i.e. parameter, assertion and precedence).

$$ACruleik = \frac{Cruleik + ATruleik}{2} \quad (2)$$

Syntactic Correctness of the # Transformation (SyC_T1) corresponds to $ACrule1-TM * 100\%$ value in the first transformation and the $SyC_T2=ACrule9-TSM * 100\%$ in the second transformation respectively.

For **Semantic Correctness (SeC_T#)** of a transformation, a similar pattern as for the metrics on Syntactic Correctness is followed. However, we only consider one value for semantic correctness of an *SCrule*, if the rule is a *CompositeRule*, we take the composite *SCrule*

value; otherwise we take the *ATRule* value. *ATRule* is 1 if the rule generates the expected element; otherwise its value is 0. The metrics that can be used in both transformations are as follows:

- **Semantic Correctness for rule *i***, which generates the target model element *k*. The rule *i* is formed by the rules *j*, which generate the target model elements *l*. The *j* depends on element type *l*. *Crule_i* is equal to *ATrule* at leaves level (i.e. parameter, assertion and precedence).

$$SCrule_{ik} = \frac{\sum_{l=1}^{\#elem_in_k} Wl * [SCrule_{jl} | ATrule_{jl}]}{\#elements_in_k} \quad (3)$$

- **Semantic Correctness of the # Transformation (SeC_T1)** corresponds to *SCrule1-TM* * 100% value in the first transformation and the *SeC_T2=Crule9-TSM* * 100% in the second transformation respectively.

Figure 8.2 shows an example of the execution order to calculate the metrics *SyC_T#* and *SeC_T#*.

From this picture we can see the bottom-up process required to calculate the metrics. For **Syntactic Correctness of the first transformation (SyC_T1)** (see Figure 8.2 left side), we started by calculating the Average Composite Syntactic Correctness for rule 8.1 of the parameter P1 (*ACrule8.1-P1*) by using the formula (2) and the values *ATrule8.1-P1* and *Crule8.1-P1*. Since Parameter P1 is missing in the transformation output, the atomic value *ATrule8.1-P1*=0. In addition, rule 8.1 is at leave level, then the Composite value for rule 8.1 (*Crule8.1-P1*) is equal *ATrule8.1-P1* and therefore the *ACrule8.1-P1* value is 0.

In the next level (i.e. Test Item level), we followed a similar process to calculate the Average Composite Syntactic Correctness for rule 5.2 corresponding to the Trigger T1 (*ACrule5.2-T1*) by using the formula (2) and the values *ATrule5.2-T1* and *Crule5.2-T1*.

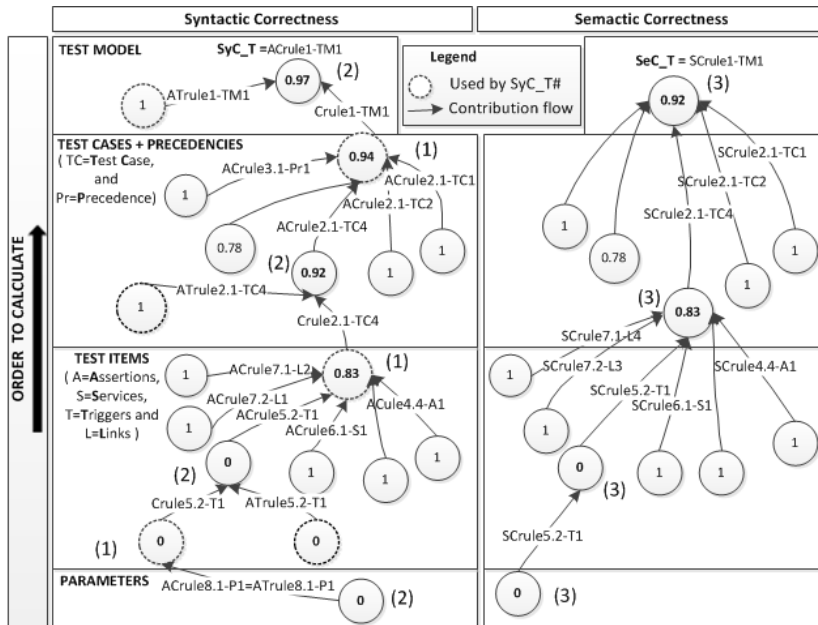


Figure 8.2. Example of the calculation of the metrics SyC_T1 and SeC_T1

Since Trigger T1 is missing in the transformation output, the values $ATrue5.2-T1=Crue5.2-T1=0$. Then, at Test Cases level the Average Composite Syntactic Correctness for rule 2.1 of the Test Case TC4 ($ACrue2.1-TC4$) was calculated by using the formula (2) and the values $ATrue2.1-TC4$ and $Crue2.1-TC4$. In this case, the information about of the test case TC4 was generated correctly, then the $ATrue2.1-TC4=1$. The $Crue2.1-TC4$ value was calculated by using the formula (1) and the values $ACrueik$ corresponding to the rules applied to the elements forming the Test Case TC4 (e.g. $ACrue7.1-L1$, $ACrue7.1-L2$, $ACrue6.1-S1$). In this paper all weights Wl required in the formula (1) are considered equal to 1. Then, the $Crue2.1-TC4=0.83$ and $ACrue2.1-TC4=0.92$.

In the top level (i.e. Test Model level), the Average Composite Syntactic Correctness for the rule 1 corresponding to the Test Model TM1 ($ACrue1-TM1$) was calculated by using the formula (2) and the values $ATrue1-TM1$ and $Crue1-TM1$. In this case, the information about of the Test Model TM1 was generated correctly, then the

$ATrule1-TM=1$. The $Crule1-TM1$ value was calculated by using the formula (1) and the values $ACrule_{ik}$ corresponding to the rules applied to the elements forming the Test Model $TM1$ (e.g. $ACrule_{3.1-Pr1}$, $ACrule_{2.1-TC1}$, and $ACrule_{2.1-TC2}$). Then, the $Crule1-TM1=0.94$ and $SyC_T1=ACrule1-TM1=0.97$ are calculated by using the formulas (1) and (2) respectively. In a similar way, the Syntactic Correctness of the second transformation (SyC_T2) is calculated by applying the formulas (1) and (2).

For **Semantic Correctness of the first transformation** (SeC_T1) (see Figure 8.2 right side), we started by calculating the Semantic Correctness for rule 5.2 of the Trigger $T1$ ($SCrule_{5.2-T1}$) by using the formula (3). Since Trigger $T1$ is missing in the transformation output, the value $SCrule_{5.2-T1}=0$. Then, at Test Case level the Semantic Correctness for the rule 2.1 of the Test Case $TC4$ ($SCrule_{2.1-TC4}$) was calculated by using the formula (3) and the values of the Semantic Correctness obtained from the rule values nested in the Test Case $TC4$ (e.g. $SCrule_{7.2-L3}$, $SCrule_{7.2-L4}$, $SCrule_{5.2-T1}$, $SCrule_{6.1-S1}$, $SCrule_{4.4-A1}$), then the $SCrule_{2.1-TC4}=0.83$. Finally, at Test Model level, the Semantic Correctness for the rule 1 of the Test Model $TM1$ ($SCrule_{1-TM1}$) was calculated by using the formula (3). Then, the $SeC_T1=SCrule_{1-TM1}=0.92$. In a similar way, the Semantic Correctness of the second transformation (SeC_T2) is calculated by applying the formula (3).

8.2.3 Experimental Design

This section describes the goal of the validation, experimental research questions, metrics used, and the subject Conceptual Schema definitions.

Goal/Question/Metric Definition

Following the line related with the Goal/Question/Metric Paradigm [140], the goal of our study is: **to analyse** the model-to-model transformations **for the purpose of** validating them **with respect to**

their syntactic and semantic correctness **from the viewpoint** of the researcher.

In order to address this goal, we defined the questions related with the respective metric to measure the syntactic and semantic correctness of the M2M transformation of our proposal (see Table 8.4).

Table 8.4. GQM for M2M transformation validation

Goal: Semantic Correctness	
Question	Derived Metric
ERQ1: What is the semantic correctness extent of our transformation rules used for generating test model from a requirements model?	SeC_rule_i. Percentage of Semantic Correctness of the rule i.
ERQ2: What is the overall semantic correctness extent of transformation rules used for generating test model from a requirements model?	SeC_Tj. Percentage of overall Semantic Correctness of the Transformation j.
Goal: Syntactic Correctness	
ERQ3: What is the syntactic correctness extent of test case elements generated by our transformation rules from a requirements model?	SyC_rule_i. Percentage of Syntactic Correctness of the elements generated by the rule i.
ERQ4: What is the overall syntactic correctness extent of test model elements generated by our transformation rules from a requirements model?	SyC_Tj. Percentage of overall Syntactic Correctness of the elements generated by the Transformation j.

Subjects: Conceptual Schemas

To assess the correctness of our proposal M2M transformation, we selected five CS from the literature, which contained a variety of characteristics that can be present in UML class diagram-based CS, including classes, relations (i.e. association, composite aggregation, and generalization) and different types of constraints (i.e. pre-condition, post-condition and body condition). These CS were of different sizes and domains (e.g. information systems, games). Table 8.5 summarizes the characteristics of these CS. A brief description of each one was introduced in Section 8.1.1. These CS specifications were first processed by hand to a requirements model based on Communication Analysis (see Section 5.2.2) using the GREAT tool modeller [141].

Table 8.5. Elements of the CSs

CS Element	MT	SS	SG	OC	DBLP
Classes	6	9	11	20	17
Attributes	26	44	26	33	59
Derived Attributes	0	1	6	27	21
Operations	13	32	19	24	32
Parameters	43	91	48	48	80
Associations	5	9	6	14	10
Derived Associations	0	0	2	0	4
Composite Aggregations	0	0	3	1	4
Constraints	9	12	19	24	44
Generalizations	0	0	4	10	13
Elements Total	102	198	144	201	271

We carefully reviewed the RM models to ensure that they were syntactically correct and that the behaviour described in the CS specification document was the intended one. Then, the two M2M transformations were executed by using the CoSTest tool (see Chapter 8) in order to generate the test scenarios model from the requirements model.

Experimental Procedure

Once the metrics, the model transformations and their transformation rules had been specified and the information needed to trace the elements had been added to the metamodels, the transformation rules were evaluated on the model instances after each execution. These rules can be evaluated by performing a simple depth-first search on the model instances, and checking whether the transformation rules have been satisfied at each transformation, as follows (see Figure 8.3).

1. **Execution of the M2M.** The first step to analyse the model-to-model transformations in our proposal is to execute the respective transformation $M2M_i$.
2. **Traversing the Models.** In our tool CoSTest, we have implemented code to traverse the model instances and reports the transformation rules used for generating the different model elements. Since the metamodels of both the source and target models are available with the transformations, and the trace

information is included in the metamodels during transformation in attribute form (i.e. location and trule of the `Element` class), this trace information needs to be analysed each time the rules specification changes.

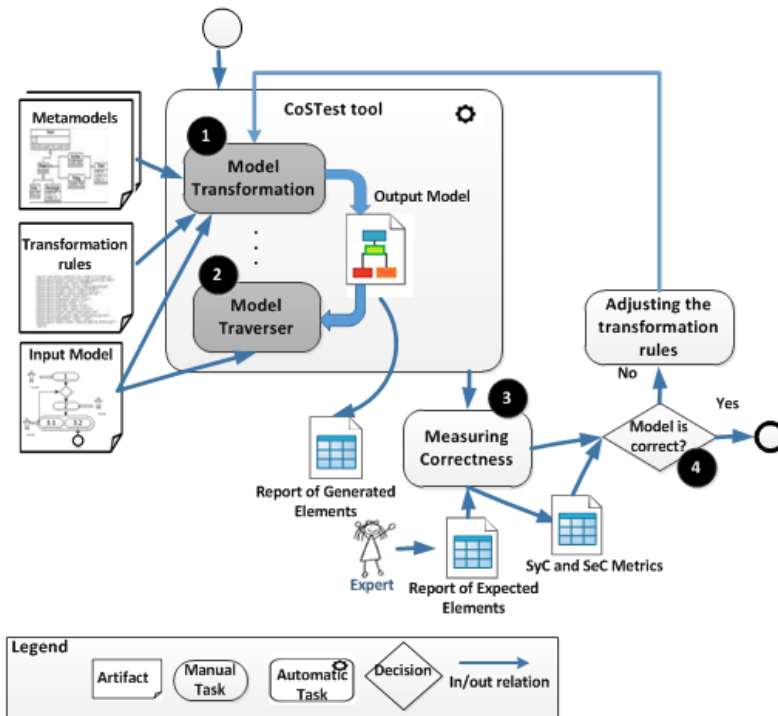


Figure 8.3. Process to evaluate a $M2M_i$ in our proposal

We call the model traverser at the end of each execution of the transformation, supplying to it the source and target model instances with the trace information.

3. **Measuring Correctness.** We check the generated elements are well-formed (i.e. syntactic correctness), which would otherwise indicate that the ATL transformation rule is syntactically incorrect. Thus, if there are differences between the obtained model and the expected one (i.e. semantic incorrectness), it could mean: a) there are unnecessary rules in the $M2M$ that generate additional elements of the expected ones, or b) there are incomplete rules

because the elements were not generated as expected. For this task, the tester uses the set of metrics proposed in the next Section and measures the correctness of the evaluated M2M.

- 4. Take a Decision.** If the output model is correct, then we can conclude that the transformation and its rules are correct. In another case, the transformation rules have to be adjusted and the evaluation process has to be executed again.

8.2.4 Results and Discussion

This section presents the results of metrics-based validation for measuring the semantic and syntactic correctness of the model transformations with their transformation rules implemented in the CoSTest tool (see Chapter 8). We collected metrics data from a heterogeneous collection of five requirements model in their two transformations for generating the test scenario models.

First Transformation

In this section, we summarize and discuss the results obtained for the selected CSs previously described (see Section 6.3.1), which were transformed from Requirements Model (RM) to Test Scenario Model (TSM) using CoSTest (see Chapter 8) to perform the proposed model transformation.

Automation increases the quality of this transformation, as errors manually implanted into transformation rules during implementation are eliminated. We used the most basic level of validation for transformations, which executes the transformation in one direction [129]: and given a source (RM) model provided by a designer or modeller, generate the corresponding target (i.e. test model).

We then checked whether the generated test model conformed to the test model metamodel and the constraints (see Section 6.3.1).

Tables 8.6-8.7 summarize the different elements of both the RM and TM models.

Table 8.7 shows the elements of both phases: column G shows the number of elements generated in the transformation and the column E to the number of elements expected after the transformation.

Table 8.6. Elements of the requirements model included in the five examples

Example \ Elements	MT	SS	SG	OC	DBLP
Start	1	1	1	1	1
End	1	1	1	1	1
Precedence	9	20	10	20	28
Communicative Event	6	11	6	11	16
Event Variant	0	2	3	4	10
And	0	1	0	1	0
Or	0	2	1	1	2
Iteration	1	4	3	4	8
Specialisation	0	0	2	3	8
Textual Requirement	0	0	8	0	18
Aggregation	7	14	13	22	28
Data Field	26	45	20	47	66
Reference Field	5	12	8	19	18

Table 8.7. Elements of the Test Model generated for the five example

Example \ Elements	MT		SS		SG		OC		DBLP	
	G	E	G	E	G	E	G	E	G	E
Test Case	6	6	12	12	8	8	13	13	21	21
Precedence	9	9	19	19	11	11	21	21	32	32
Final Precedence	7	7	14	14	10	10	15	15	29	29
Assertion	7	7	12	12	15	15	14	14	39	39
Service	6	10	21	21	15	15	20	20	26	26
Trigger	2	3	12	12	6	6	6	6	17	17
Link	5	5	9	9	9	15	24	26	19	19
Parameter	49	58	120	120	90	102	155	159	197	197

From these results we can see that the values in bold (e.g. services, triggers, parameters for MT) represent the elements that differ from those expected and therefore indicate an error in the transformation rules related to these elements. For example, for MT services we expected 10 elements (see rows related to 7.x rules in Table 8.8) but only 6 elements were generated by the respective transformation rules.

We added the “Final Precedence” row to report the number of precedence relations obtained after of adjusting these relations in the

test model, inspected the MMT result and determined its correctness by comparing the Test Model elements with the manually derived CS elements by expert.

For each question related to the correctness goal, we report the results obtained when applying the respective metrics for each measurement model (e.g. MT, SS systems). This report was supported by CoSTest (see Chapter 7).

Table 8.8 shows some of the results of the comparative effort of the Syntactic and Semantic Correctness achieved for each transformation rule in the five CS used in this study during the first transformation.

From these results, we can see that the number of rules (i.e. the number of rows with data in Table 8.8) required by DBLP is the highest (i.e. 21 rules for semantic correctness) of the five generated Test Model, while this is not the case for MT, which only requires 13 rules for semantic correctness (SeC).

The differences found in each validation phase allowed us to take corrective actions to adjust our M2M transformation, so that for the next phase the problems identified in the transformation rules were fixed.

For example, for the first phase four of six Service classes were omitted by Rule 7.3. Therefore, in this first phase rule 7.3 achieved 33% of semantic correctness and 100% of syntactic correctness for the generated elements. In this phase rule 5.2 was missing, omitting a Service class, so that the semantic correctness achieved by 5.2 is 0% and the syntactic correctness value is not required.

Finally, for this first phase (i.e. MT case), the M2M transformation achieved 100% of syntactic correctness, while the semantic correctness was 96.30%. Similarly, the values of correctness for the other phases were calculated.

Table 8.8. Results of SyC_T1 and SeC_T1 for the five cases

Phase T. Rule	Test Models									
	1° – MT (%)		2° – SS (%)		3° – SG (%)		4° – OC (%)		5° – DBLP (%)	
	SyC	SeC	SyC	SeC	SyC	SeC	SyC	SeC	SyC	SeC
SyC_T	100	96.3	98.3	100	100	98.0	100	99.6	100	100
SeC_T	100	100	100	100	100	100	100	100	100	100
1	100	100	100	100	100	100	100	100	100	100
2.1	100	6/6 90.7	100	10/10 100	100	5/5 92.5	100	9/9 98.4	100	11/11 100
2.2	-	-	100	2/2 100	100	3/3 100	100	4/4 100	100	10/10 100
3.1	100	9/9 100	100	14/14 100	100	5/5 100	100	12/12 100	100	18/18 100
3.2	-	-	100	2/2=100	100	3/3 100	100	6/6 100	100	8/8=100
3.3	-	-	-	-	-	-	-	-	-	4/4=100
3.4	-	-	100	1/1 100	100	3/3 100	100	1/1 100	100	2/2 100
3.5	-	-	50	2/2=100	-	-	100	2/2=100	-	-
4.1	-	-	-	-	100	8/8 100	100	100	100	18/18 100
4.2	-	-	100	1/1=100	-	-	-	-	-	-
4.3	100	1/1=100	-	-	-	-	-	-	-	-
4.4	100	6/6 100	100	11/11 100	100	7/7 100	100	14/14 100	100	21/21 100
5.1	100	1/1=100	100	1/1=100	100	1/1=100	-	-	100	1/1=100
5.2	-	0/1=0	100	2/2=100	100	1/1=100	100	2/2=100	100	5/5=100
5.3	-	-	100	1/1 100	-	-	-	-	-	-
5.4	-	-	100	2/2 100	100	3/3 100	100	4/4 100	100	10/10 100
6.1	100	6/6 100	100	9/9=100	100	11/11 100	100	20/20 100	100	22/22 100
7.1	100	1/1 100	100	3/3 100	100	3/3 100	100	3/3 100	100	4/4 100
7.2	100	3/3 100	100	3/3 100	100	5/5 100	100	16/16 100	100	11/11 100
7.3	100	2/6 33.3	100	18/18 100	100	6/6 100	-	-	100	6/6 100
7.4	-	-	-	-	-	0/6 0	100	4/4 100	100	3/3 100
7.5	-	-	-	-	-	-	-	0/2=0	-	-
8.1	100	27/27 100	100	51/51 100	100	35/35 100	100	61/61 100	100	101/101 100
8.2	100	3/3 100	100	3/3 100	100	11/11 100	100	19/19 100	100	10/10 100
8.3	-	-	-	-	-	0/6 0	100	4/6 66.7	100	3/3 100
8.4	-	-	-	-	-	0/6 0	100	4/6 66.7	100	3/3 100

From these results, we see that transformation rules 5.2, 7.3, 7.4, 7.5, 8.3 and 8.4 (see rows in Table 8.8 achieved less than 100% semantic correctness in some of the validation phases (see columns in Table 8.8), while the Syntactic Correctness of the rules achieved a score of 100% in most phases, except for rule 3.5 in the second phase (i.e. SS CS).

We also calculated the semantic and syntactic correctness of all first M2M transformations based on the partial values of semantic and syntactic correctness of each transformation rule.

The first row of Table 8.8 shows the values of these metrics (i.e. SeC_T1 and SyC_T1), so that the syntactic correctness was 100% in 4 out of 5 of the analysed CS. This result was as expected because syntactic correctness is easier to achieve with the tests performed while the transformation is implemented.

The semantic correctness varies in each phase (i.e. 96.3%, 100%, 98%, 99.6% and 100%) depending on the number of elements that matched with the expected elements.

Once we identified the correctness problems (see Figure 8.4) in this M2M transformation, we reviewed the transformation rules and found the following explanations:

- **Missing Rules.** For rules 5.2, 7.4, 7.5, 8.3 and 8.4 it was necessary to extend the respective rules by adding the required code.
- **Incorrect Rules.** Rules 3.5, 7.3, 8.3 and 8.4 required some adjustments, e.g. 3.5 was modified by adding “->AND->OR->” in the Precedence class name it generated. The definition of Rules 7.3, 8.3 and 8.4 was correct, however there was an unreachable code in the ATL code implemented in CoSTest. We therefore restructured the code to correct this unreachable code bug.
- **Unnecessary rules.** Metrics can also sometimes detect unnecessary rules in the transformation (e.g. alternative rules that implement code for Rules 3.5 and 7.3 are not applied in any case, as well as some helpers) and need to be deleted.

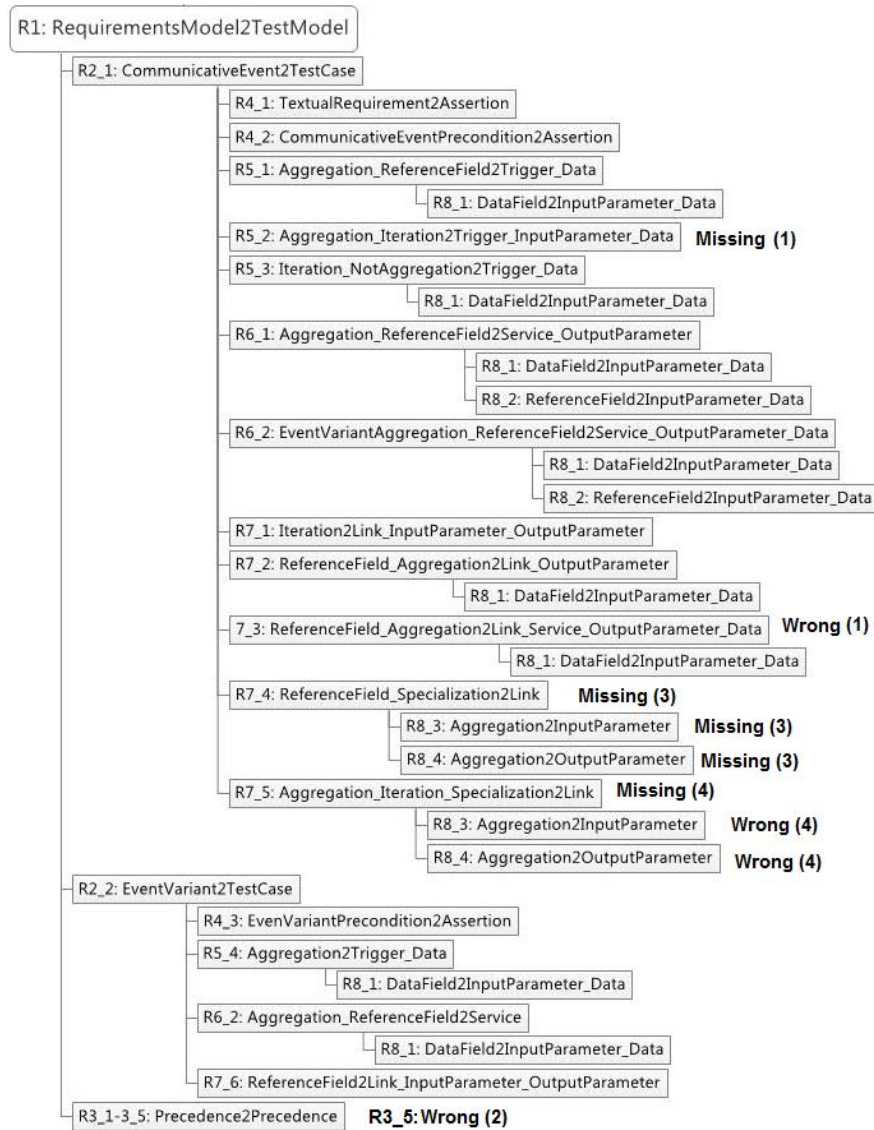


Figure 8.4. Structure of T1 Transformation with the identified problems

Finally, at the end of the evaluation (i.e. in the fifth phase), the syntactic and semantic correctness achieved by the first two MMTs and each transformation rule was 100%.

Second Transformation

The goal of this transformation is to obtain a model conformed to the test scenario metamodel presented in Section 6.2.2.

Table 8.9 shows the different elements of Test Scenario Models for the five subjects. As in the previous transformation, we included the columns G (generated elements) and E (expected elements).

Table 8.9. Elements of the Test Scenario Model generated for the five examples

Example Elements	MT		SS		SG		OC		DBLP	
	G	E	G	E	G	E	G	E	G	E
Test Scenario	1	1	2	2	2	2	3	3	7	7
Test Case	6	6	17	17	14	14	31	31	52	52
Assertion	7	7	15	15	28	28	31	31	85	85
Service	6	10	32	32	30	30	57	57	69	69
Trigger	2	3	16	16	10	10	10	10	38	38
Link	5	5	16	6	18	30	67	73	44	44
Parameter	49	58	188	188	176	200	419	431	487	487

Like the analysis done for Table 8.7, the values in bold represent the elements that differ from those expected and therefore there is an error in the transformation rule related to that element. For example, the error detected in the first phase for MT services is translate to the second phase, so that we expected 10 service elements (see rows related to 7' in Table 8.10) but only 6 elements were generated by the respective transformation rules

Table 8.10 shows the results of the comparative effort during the second transformation to measure the syntactic and semantic correctness achieved for each transformation (i.e. SyC_T and SeC_T row and SyC and SeC columns respectively) and with each rule (different rows in Table 8.10) in the five CS used in this study.

Table 8.10 shows the calculation of the metrics, and the rules that had errors are those that do not have a value of 100% in the respective metric.

Table 8.10. Results of SyC_T2 and SeC_T2 for the five cases

Phase T. Rule		Test Scenario Models									
		1° – MT (%)		2° – SS (%)		3° – SG (%)		4° – OC (%)		5° – DBLP (%)	
		SyC	SeC	SyC	SeC	SyC	SeC	SyC	SeC	SyC	SeC
SyC_T	SeC_T	100	90.7	100	100	100	94.6	100	98.6	100	100
T2	T1										
9	1	100	100	100	100	100	100	100	100	100	100
10		100	1/1 90.7	100	2/2 100	100	2/2 100	100	3/3 98.6	100	7/7 100
2'	2.1	100	6/6 90.7	100	10/10 100	100	10/10 92.5	100	9/9 100	100	11/11 100
	2.2	-	-	100	2/2 100	100	4/4 100	100	4/4 100	100	10/10 100
4'	4.1	-	-	-	-	100	8/8 100	100	100	100	18/18 100
	4.2	-	-	100	1/1=100	-	-	-	-	-	-
	4.3	100	1/1=100	-	-	-	-	-	-	-	-
	4.4	100	6/6 100	100	11/11 100	100	7/7 100	100	14/14 100	100	21/21 100
5'	5.1	100	1/1 100	100	1/1 100	100	1/1 100	-	-	100	1/1 100
	5.2	-	0/1 0	100	2/2 100	100	1/1 100	100	2/2 100	100	5/5 100
	5.3	-	-	100	1/1 100	-	-	-	-	-	-
	5.4	-	-	100	2/2 100	100	3/3 100	100	4/4 100	100	10/10 100
6'	6.1	100	6/6 100	100	9/9 100	100	11/11 100	100	20/20 100	100	22/22 100
7'	7.1	100	1/1 100	100	3/3 100	100	3/3 100	100	3/3 100	100	4/4 100
	7.2	100	3/3 100	100	3/3 100	100	5/5 100	100	16/16 100	100	11/11 100
	7.3	100	2/6 33.3	100	18/18 100	100	6/6 100	-	-	100	6/6 100
	7.4	-	-	-	-	-	0/12 0	100	4/4 100	100	3/3 100
	7.5	-	-	-	-	-	-	-	0/6=0	-	-
8'	8.1	100	27/27 100	100	51/51 100	100	35/35 100	100	61/61 100	100	101/101 100
	8.2	100	3/3 100	100	3/3 100	100	11/11 100	100	19/19 100	100	10/10 100
	8.3	-	-	-	-	-	0/12 0	100	12/18 66.66	100	3/3 100
	8.4	-	-	-	-	-	0/12 0	100	12/18 66.66	100	3/3 100

Since the second model transformation generates test scenarios, the problems found in some of the rules in the first transformation are translated into each scenario generated from these rules. Columns T1 and T2 shows the correspondence of the rules of the transformation T2

with the rules of the transformation T1. For example, R2' transforms the elements generated by R2.1 and R2.2 in T1.

For Syntactic Correctness (SyC) the rules in the first transformation achieved a score of 100% in most of the phases (4/5), except in the second phase (i.e. SS CS), with a SyC value of 98.3% (see row SyC_T1 of Table 8.8). In the second transformation the SyC value was 100% for all phases, suggesting that the impact of the syntactic correctness problems in the second transformation depends on the defect type detected in the transformation rule. For example, in the SS subject, the defective rule R3_5 (i.e. R3_5 does not include the specification “->AND->OR->” in its name) and does not affect the generation of the test scenario.

On the other hand, Semantic Correctness varied in each phase of the first (i.e. 96.3%, 100%, 98%, 99.6% and 100% in Table 8.8) and second transformation (i.e. 90.7%, 100%, 94.6%, 98.6%, 100% in Table 8.10), according to the number of elements generated by each defective rule and the number of test scenarios generated in this second transformation. For example, in the SG subject the impact on correctness is greater in the second transformation (i.e. SyC=100% and SeC=94.64%), because two scenarios were generated using all elements of the test model, so there are more elements generated with rules that have anomalies.

Since the purpose of this chapter is to validate the syntactical and semantical correctness of the M2M transformation, we exercised the transformation with a set of requirements models derived from CS specifications found in the literature and then compared the results with the expected outcomes by using a set of metrics defined to measure the semantic and syntactic correctness of the proposed M2M transformation. Both the M2M transformation and the report of the generated TM elements are supported by the tool.

The M2M transformation validation was performed in several phases. In this chapter we report the results of the comparative effort

in five phases, where the Syntactic Correctness (SyC) of the rules in the first transformation achieved a score of 100% in most phases (4/5), except in the second phase (i.e. SS CS), with a SyC value of 98.3%. In the second transformation the SyC value was 100% for all phases, suggesting that the impact of the syntactic correctness problems on the second transformation depends on the defect type detected in the transformation rule. On the other hand, the semantic correctness varied in each phase of the first (i.e. 96.3%, 100%, 98%, 99.6% and 100%) and second transformation (i.e. 90.7%, 100%, 94.6%, 98.6%, 100%), depending on the number of elements generated by each defective rule and the number of test scenarios generated in this second transformation.

Although this validation does not guarantee full correctness of the M2M transformation, it shows that it has very interesting benefits. In particular, the defined metrics were useful for identifying bugs (i.e. incorrect, missing and redundant rules) in the transformation rules in a cost-effective manner, so these M2M transformations are suitable to be integrated in our tool support (see Chapter 8). Moreover, the metrics can measure the correctness of CSs without having to transform them into any other formalism or to abstract away any of their features.

8.3 Evaluating the CoSTest Mutant Generator

The empirical assessment of test techniques plays an important role in software testing research. One common practice is to instrument faults, either manually or by using mutation operators. The latter allows the systematic, repeatable seeding of large numbers of faults, helping to clarify assumptions, support understanding, analysis, prediction, and decision-support.

In Mutation testing the most critical activity is the adequate design of mutation operators so that they reflect the typical defects of the artefact under test. We therefore designed a set of mutation operators for Conceptual Schemas (CS) based on Unified Modelling Language

(UML) Class Diagrams (CD). The main potential advantage of mutation operators is that they describe precisely the mutants they can generate and thus support a well-defined, fault-injecting process. Figure 8.5 illustrates the definition process of mutation operators.

The research group met for decision-making with two main objectives: (i) to focus on evaluating some properties of the mutation operators for FOM and (ii) to validate the effectiveness of CoSTest components to automatize mutant generation (i.e. Mutant Generator) (see Section 7.7 in Chapter 7).

Two experiments were performed using mutation in an iterative process to evaluate some properties of the mutation operators as well as the effectiveness and efficiency of the CoSTest Mutant Generator.

8.3.1 Experiment No 1: Evaluating the Mutation Operators Implemented in CoSTest

The first experiment was an iterative process to evaluate the mutation operators for FOM implemented in CoSTest.

In this experiment we used the CoSTest tool V0.5 (conception of the tools), which was evolved until achieving the stable V1.0 version. The tool generated the first order mutant (FOM), but did not include the facility for selecting (all or partially) the mutation operators before calculating and generating the mutants.

Experimental Design

The experiment was performed by the authors (researchers) of [115], which reports on the use of the tools in a controlled environment using three types of system: (i) the Super Stationery (SS) system, (ii) an Expense Report (ER) management system, and lastly, (iii) the Sudoku Game (SG) system [133], which is more variant-rich than the other two CS. The source files of the requirements models and Conceptual Schemas can be found at <https://staq.dsic.upv.es/webstaq/costest.html>

The experiment was run as an iterative process with approximately 3 iterations in three months (from July 2015 to September 2015). The objective was to evaluate the usefulness of the mutation operators for FOM. Then, we prepared a renewed version of the tools that included the emerging improvements.

The experiment was performed in a laboratory environment where the CS requirements were specified using GREAT tool [141]. Eclipse Framework tools such as UML2 or Papyrus were used for modelling conceptual schemas, and Microsoft Excel for managing and analysing the test results.

Figure 8.5 illustrates the i -th iteration of the experiment. Each CS subject was analysed based on FOM that can be generated using CoSTest version V0.5. It was then used to generate the test cases according to the requirements model and to execute them against the mutants.

Finally, the mutation score for mutant and mutation operator, contribution factor of mutation operator and impact indicator were computed in order to evaluate some properties of the mutation operators. The last iteration of the experiment was used to exemplify the CoSTest tool V1.0 and evaluate some properties of the mutation operators [93].

Conclusions and changes to the tool

The FOM mutation operators were evaluated in the experiment by means of the contribution factor, impact indicator and mutation score.

Based on the results obtained by applying mutation testing, 56% (10/18) of our mutant operators generated a high number of killed mutants (score mutation=100 %). These results suggest that these operators generated mutants that are relatively easy to detect by the provided test suites.

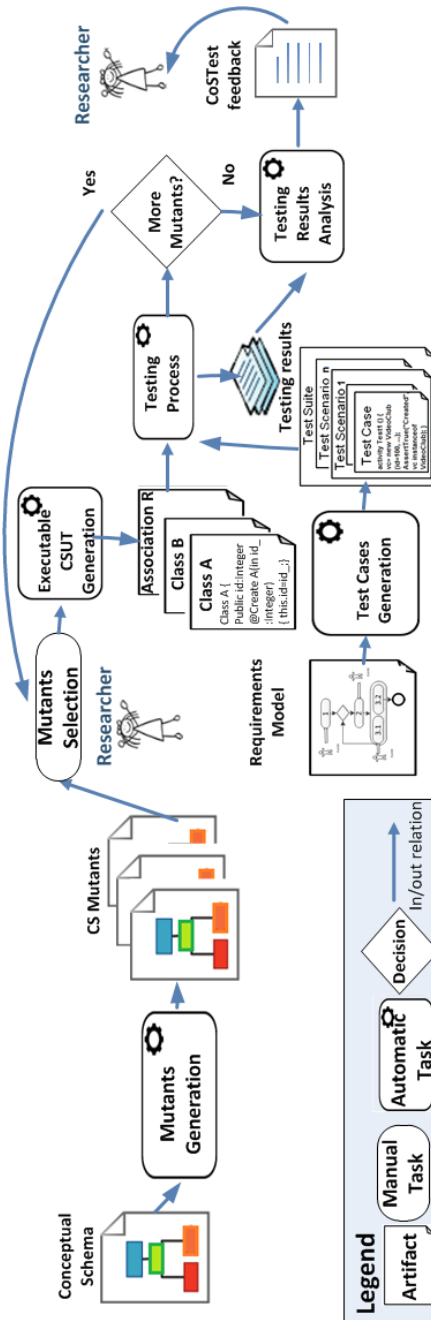


Figure 8.5. i-th iteration of the experiment applying the CoSTest tool

In the other case 44% (8/18) of the operators related to characteristics of associations (i.e. multiplicity and aggregation type) and constraints generated hard-to-detect mutants and their application would stimulate selection of high quality tests. However, the behaviour of the mutation operators may depend on the characteristics of the CS they are applied to, such as the number, element type and complexity of the constraints.

Some of the main changes applied to the mutation operator list were the restrictions included in the mutation operator rules to avoid generating non-valid mutants (see Table A.1 in Appendix A). Some of the main changes applied to the tool were: (i) include the facility for selecting the mutation operators before calculating and generating them, (ii) structure the Excel report containing CSUT elements, testing log, covered elements, found defects, test case verdicts and (i) generate a report for each mutant to help identify defects. All the reports were exemplified using the three analysed CS.

8.3.2 Experiment No 2: Validating the Effectiveness and Efficiency of Mutant Generator of CoSTest

The second experiment was an iterative process to evaluate the effectiveness and efficiency of the CoSTest mutant generator. The V1.0 version generated mutants using the mutation operators but could not facilitate the summaries of the generated mutants including the required mutation time nor could it analyse the CS information to help discard equivalent mutants. The V1.0 version used in this study can only execute the test cases on one conceptual schema at a time, so it was not possible to select several conceptual schemas (i.e. mutants) to test with the selected test cases and report the summarized results of all of them.

Experimental Design

The experiment was performed by the authors (researchers) of [124], which reports on the evaluation of the tool in a controlled

environment using six types of system: (i) the Medical treatment (MT) system, (ii) the Sudoku Game (SG) system [133], (iii) an Expense Report (ER) management system, (iv) the Online conference review (OCR) system, (v) the SuperStationery (SS) system and lastly, (vi) Photography Agency (PA) system. The source files of requirement models and Conceptual Schemas can be found at <https://staq.dsic.upv.es/webstaq/costest.html>

The experiment was run as an iterative process; approximately 6 iterations were performed in three months (from September 2015 to November 2015). The objective was to carry out an evaluation of the CoSTest tool V1.5 with respect to the effectiveness and efficiency in generating valid First Order Mutants to UML CD-based CS.

In this experiment the CS requirements were specified using GREAT tool [141], Eclipse Framework tools such as UML2 or Papyrus were used for modelling conceptual schemas and Microsoft Excel (for managing both results testing and mutation analysis). Figure 8.6 illustrates the i -th iteration of the experiment.

The version V1.0 of the CoSTest tool was provided. Each CS was used to generate the FOM that can be generated using the CoSTest *Mutant Generator*. CoSTest was then used to generate the test cases according to the requirements model and to execute them against the mutants and report the results. The researcher manually analysed the mutants that were not killed to determine whether they were equivalent (i.e. the CS mutant produces the same output as the original CS as if it had no faults) and register them.

The last iteration of the experiment was used to evaluate CoSTest V1.2 with respect to its effectiveness and efficiency in generating valid First Order Mutants to UML CD-based CS [124].

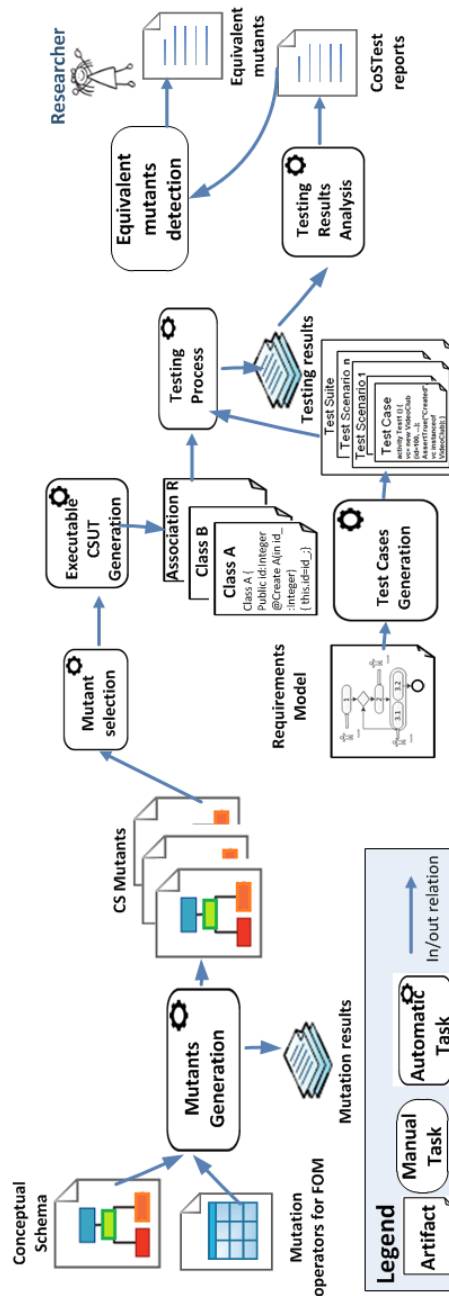


Figure 8.6. i-th iteration of the experiment applying the CoStest tool

Conclusions and changes to CoSTest

The experiment showed the effectiveness and efficiency of CoSTest in generating FOM [124] and that mutation operators can be automated avoiding the generation of a high percentage (49.1%) of non-valid mutants and generating a low percentage (7.2%) of equivalent mutants.

However, detecting these mutants is costly in terms of the time and effort of creating, executing and manually inspecting them. We therefore implemented the restrictions and rules for eliminating them by performing a static analysis of the CS. As these results show, the reduction achieved in this analysis of equivalent mutants is about 74.3%, which is equivalent to 2249.24 seconds estimated by KLM, and the cost of reducing non-valid mutant is 49.1% (48833.4 seconds estimated by KLM) by using the mutation tool in the six subject CSs involved in this study.

Therefore, the results of this study suggest that the mutation tool can help researchers and supports a well-defined, fault-injecting process to generate a potentially large number of valid and non-equivalent FOMs, increasing the statistical significance of results obtained in assessing test case quality.

However, some changes were applied to the tool to (i) include the report on generated mutants, (ii) add the restrictions to avoid equivalent mutant generation using WOP2, (iii) report the time required to generate mutants (iv) include the facility of executing a set of test cases against a set of mutants and generate an Excel report containing CSUT name, testing time, final verdict, defect id, defect mode, found defect and CSUT element for all tested mutant of a CS to help with information visualization.

8.4 Validating of the Effectiveness of CoSTest' Test Cases

The following is a description of the comparative experiment to evaluate the effectiveness of CoSTest test cases.

The experiment was motivated by the need to investigate the effectiveness of our testing framework; that is, we intended to compare the effectiveness when they were applied in both first order mutants and high order mutants to detect faults in eight CS.

The experiment was carried out in 2016 (from January to March) and was designed according to Wholin et al. [142] as reported by Jurist and Moreno [127].

8.4.1 Experiment Goal and Questions

The experimental goal according to the Goal/Question/Metric Template [143] is to analyse the resulting CoSTest test cases for the purpose of evaluation with respect to their effectiveness in detecting fault types from the point view of the researchers in the context of mutants generated for eight CS.

We are interested in determining if the test case effectiveness is the same for both types of mutants (i.e. FOM and HOM). Therefore, we pose and study the following experiment research questions (ERQ):

- **ERQ1:** How significant is the influence of the mutation type in the effectiveness of CoSTest test cases for detecting faults and fault types?

And as we are also interested in measuring whether the test case quality is depending on the type of mutant:

- **ERQ2:** How adequate are CoSTest test suites for killing both the First Order Mutants and High Order Mutants of Conceptual Schemas?

8.4.2 Variables

Independent Variables

We consider one independent variable (a.k.a. factor [127]):

- *Mutant type*. Since this study uses mutations for injecting the artificial faults into a CS, CSs can be classified into two types according to the number of mutated elements:
 - First Order Mutant (FOM) (baseline), which is generated by applying mutation operators (i.e. rules to modify the grammar used to capture the syntax of a software artefact [113]) only once.
 - Higher Order Mutant (HOM), which is generated by applying mutation operators more than once [113].

Dependent Variables

We consider the following dependent variables (a.k.a. response variables [127]), which are expected to be influenced to some extent by the independent variable.

- *Effectiveness in Detecting Fault*. To investigate our ERQ1 we need to measure the effectiveness of the CoSTest test cases in terms of the number of faults found and the type (or cause) of the faults that were found [144] as well as the mutation score, which can be used to measure the effectiveness of a test suite in terms of its ability to kill mutants because it is one outcome of the Mutation Testing process, which indicates the quality of the input test set [15].

8.4.3 Metrics

Effectiveness

For evaluating the effectiveness of our testing technique, we used three metrics:

- *Rate of Fault Detection (FDR)*. The metric FDR is the value calculated by dividing the number of faults detected by the tool by the total number of faults that are expected to be identified from the CS mutants.

$$FDR(T) = \frac{F_D(T)}{F_E}$$

- *Rate of Fault Type Detection (FTDR)*. The metric FTDR is the value calculated by dividing the number of fault types detected by the tool by the total number of fault types that are expected to be identified from the CS mutants.

$$FTDR(T) = \frac{FT_D(T)}{FT_E}$$

- *Mutation Score*. During execution each CS mutant M_i will be run against a test case suite T . If the result of running M_i is different from the result of running CS (without defects) for any test case in T , then the mutant M_i is said to be “killed”, otherwise it is said to have “survived”. A CS mutant may survive either because it is equivalent to the original model (i.e. it is semantically identical to the original model although syntactically different) or the test set is inadequate to kill the mutant.

Thus, the adequacy of a test suite T for a given set of M mutants is quantitatively evaluated with a mutation score (MS). It is measured as the ratio of the number of killed mutants $M_K(T)$ over the total number of the non-equivalent mutants M_T generated for a CS. It is calculated by:

$$MS(T) = \frac{M_K(T)}{M_T}$$

8.4.4 Hypotheses

We defined three hypotheses: Table 8.11 shows the null hypotheses (represented by a 0 in the subscript), which corresponds to the absence of an impact of the independent variables on the dependent variables.

Table 8.11. Specification of hypotheses

Null hypothesis	Statement: Mutant type does not influence ...
$H1_0$ (ERQ1)	... the effectiveness of the CoSTest test cases in detecting faults in Conceptual Schemas
$H2_0$ (ERQ1)	... the effectiveness of the CoSTest test cases in detecting fault types in Conceptual Schemas
$H3_0$ (ERQ2)	... the adequacy of the CoSTest test cases

The alternative hypotheses involve the existence of such an impact and are the expected result.

8.4.5 Experimental Material

Subject CS

Most of the input CSs used in this experiment were small UML/ALF-based models. In particular, this experiment took as input eight CSs containing a variety of characteristics that can be present in UML CD-based CS, including classes, relations (i.e. association, composite aggregation, and generalization) and different types of constraints (i.e. pre-condition, post-condition and body condition). These CS were of different sizes and domains (e.g. information systems, games). One case is taken from industrial, others CSs were found in the literature (i.e. [131], [133] and [132]).

In order to guarantee that our tool is also effectively detecting the different mutants created using the defined mutation operators (see Table A.1), we also used CSs artificially created for this purpose (i.e. ER, OCR, VC, and PA) containing the CS elements required to inject the faults.

Table 8.12 summarizes the characteristics of these CS. A brief description of each CS is given in Section 8.1.1:

Table 8.12. Elements of the Subject Conceptual Schemas

Element	VC	MT	SG	ER	OCR	SS	PA	IM
Classes	5	6	11	7	10	9	15	6
Attributes	19	26	32	42	62	45	85	29
Operations	6	13	19	24	16	32	31	13
Parameters	22	43	48	75	77	91	86	51
Associations	4	5	11	8	10	9	19	4
Constraints	17	9	19	21	14	12	37	8
Generalizations	0	0	4	0	3	0	0	0

Our experiment was carried out under a within-subject design, all our subjects were exposed to the two treatments of our independent variable (CS type) [145].

8.4.6 Procedure

We provide in this section a brief description and justification of the analysis procedure that we used.

Figure 8.7 summarizes the experimental process, which involved performing the following seven steps:

- 1) **Choose CS Subjects.** The selected subjects are described in Section 8.1.1.
- 2) **Select a Conceptual Schema and generate the test suite.** A test suite T was generated to kill CS mutants for each subject CS by following Steps 1-7 of Section 5.8, we then analysed the information on the generated test cases in order to detect problems in the generation process (e.g. repeated test cases).
- 3) **Execute Test Suites on CS.** Each test suite is executed on the respective CS subject.

We assessed whether an invalid test case required a manual setting (e.g. concretize variables that require several values because they should be unique values or adjust a negative test case so that it can create a valid sequence of events to validate constraints).

We adjusted the test cases in order to get a successful testing process with the original CS and registered the invalid test cases. For example for OCR CS, we required updating the test case number 19, which validate the precondition “context Submission:: new_submission() pre: Author->size(>0 “ with an invalid state (test case negative), so we removed the statement that previously creates an author.

Additionally, we had to concretize different values for the variable id_member used in the classes PCMember, PCChair and Author, so that there are no problems with the constraints that validate unique values.

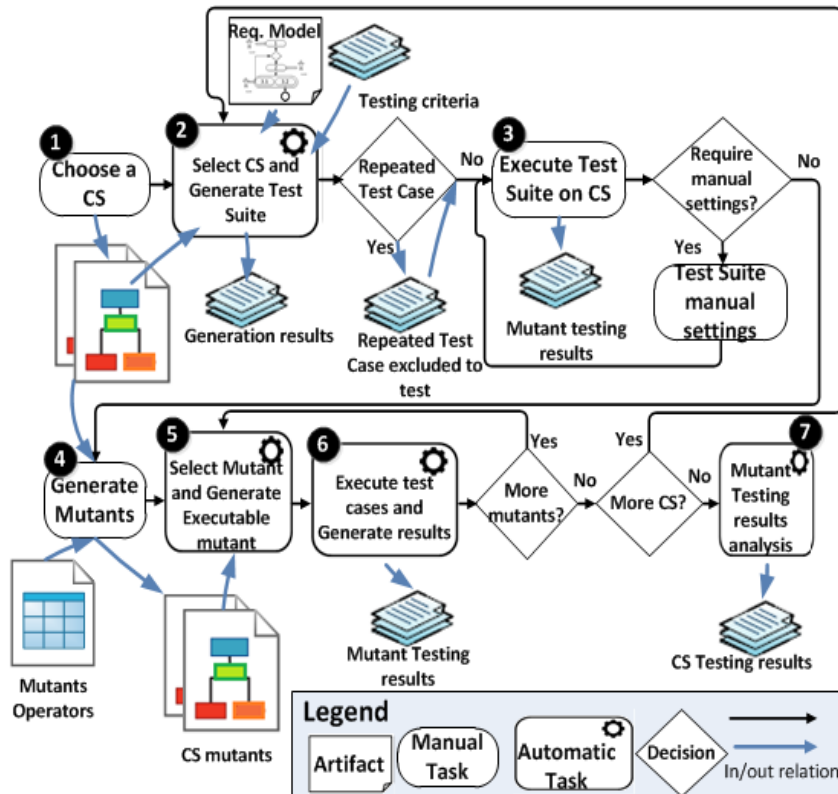


Figure 8.7. Steps taken in experimental process

- 4) **Generate CS Mutants.** As this step was quite computationally expensive, we used our *Mutant Generator* (see Section 7.7 in Chapter 7) for generating first order mutants, in contrast to the High order mutants, which were generated manually.

In this study, we used all the FOMs generated by the tool for all CS subjects (see mutation operators of Table A.1 marked with “*” in Appendix A).

In the other case, since there is no tool to automatically generate HOMs, and also due to the unmanageably large number of mutants that would result from including the set of higher order mutants [17], we tried to generate in each subject CS, 3 mutants for each mutation operator from Table A.1 (see mutation operators marked with “**” in the Appendix A). Elements were randomly selected to apply the mutation, however, some CS

subjects did not allow random selection due to the limited number of elements required by some mutation operators (e.g. WAT2, WGE and MGE).

Therefore, a random selection of elements from CSs combined with a size of 3 mutants for each mutation operator for HOM (“**”) from Table A.1 were deemed sufficient (enough variability in faulty versions do not cover in FOM). Figure 8.8 shows an excerpt of Video Club CS and the application of five mutation operators of first order.

A syntax analysis was then performed by using the ALF parser to ensure that the mutants were valid and could be used in a testing process.

- 5) **Select and generate an executable CS mutant.** Each CS mutant is transformed into an executable CS (CSUT) by using the respective CoSTest module (see Step 7 in Section 5.8).
- 6) **Execute Test Suites on CS Mutants.** We ran each test case for each mutant and maintained the test status (i.e. passing/failing/inconclusive) using our CoSTest tool. Then, we compared the output of each mutant against the output of the original version of the CS with no faults.

When the output of the mutant was different to the original CS output, the test case was labelled as failing and when the outputs were exactly the same, the test case was tagged as passing.

We then manually examined the FOM with zero kills and eliminated any that were semantically equivalent to the original CS.

The analysis of survivor mutants in order to identify equivalent mutants is a prerequisite for calculating a mutation score. An example of an equivalent mutant is shown in Figure 8.9, in which the changed operator did not influenced the result of the assignment.

We used the CoSTest option to export the results (faults and coverage analysis) of the testing process of the CS subject.

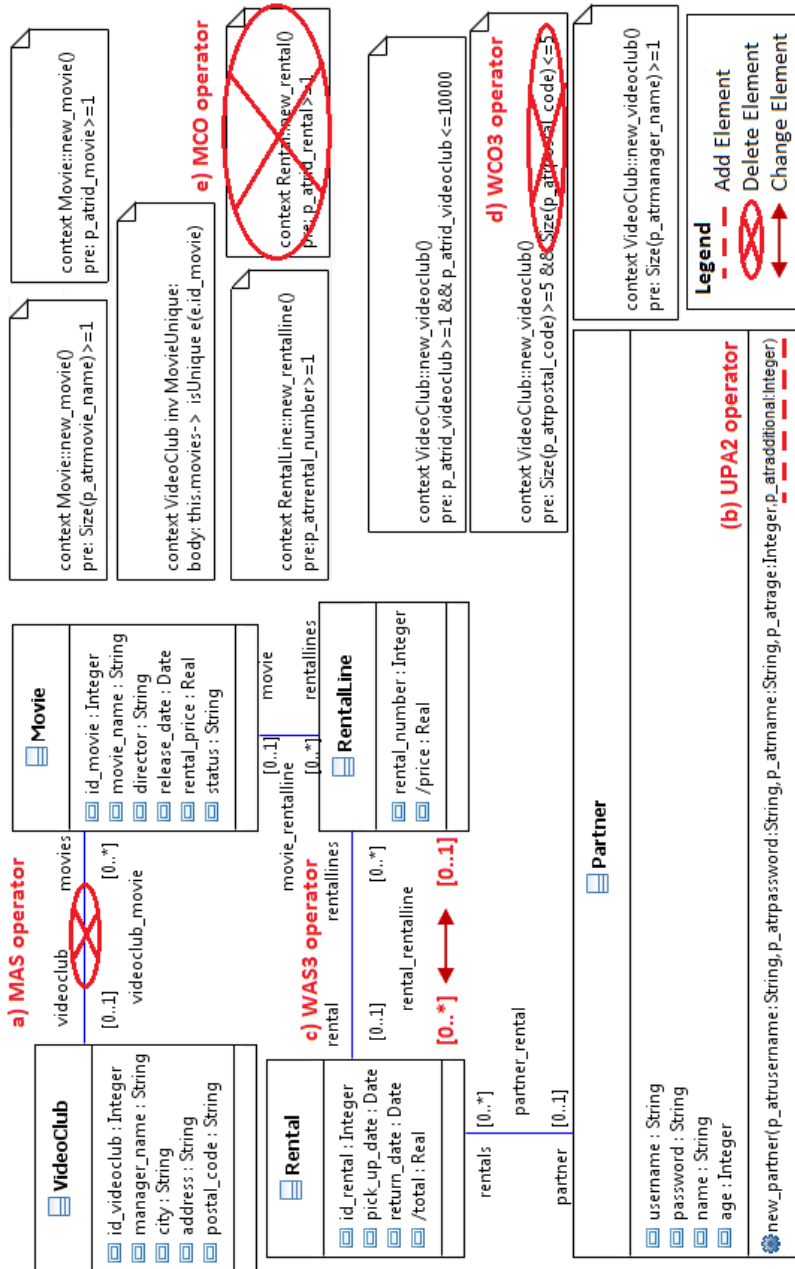


Figure 8.8. Application of five mutation operators on Video Club CS

<p>Original Constraint with relational operator “=”</p> <pre>this.employees_number=this.employees->select e(e.fired==false)->size()==0? 0: this.employees->select e(e.fired==false)->size();</pre> <p>Mutated Constraint change the relation operator to “<=”</p> <pre>this.employees_number=this.employees->select e(e.fired==false)->size()<=0? 0: this.employees->select e(e.fired==false)->size();</pre>
--

Figure 8.9. Excerpt of a Constraint mutated by WCO8 operator

If there are further CS to be studied, steps 2 to 5 are repeated with the next subject CS.

- 7) **Analysis of Testing Results.** We then determined which test case in the pool detected which mutant and fault.

Next we computed the fault detection ratios of all test suites, plotted the detection ratio distributions of mutants and faults for each subject CS. Then, CoSTest effectiveness and adequacy of the test suite were calculated from the information recorded in this process.

These results are given in the next Section.

8.4.7 Analysis of Results

This section describes the analysis and interpretation of the results related to our response variables (e) for ERQ1 and ERQ2.

The Statistical analysis was carried out on the Statistical Package for Social Sciences (SPSS) V20.0.

Fault Detection Effectiveness

Since the first question (ERQ1) was aimed at evaluating CoSTest’s Effectiveness at detecting faults, we compared the ratio of fault types detected per mutant type (i.e. FOM and HOM) in the different CS subjects. Table 8.13 shows both the number of the fault types detected in each CS subject by mutant type.

Shapiro-Wilk tests were performed to evaluate the samples normality. We used this test as our numerical means of assessing normality because it is more appropriate for small sample sizes (<50 samples).

Table 8.13. Faults and Fault Types detected by Mutant type

Defect	VC		MT		SG		ER		OCR		SS		PA		IM		
	F	H	F	H	F	H	F	H	F	H	F	H	F	H	F	H	
Extraneous Derived Attribute				3							5				3		3
Extraneous Constraint				3	1						2				3		3
Missing Class	5	1	6	3	11	2	7	2	10	2	2	9	3	15	2	6	3
Missing Constraint	52		15		50	10	36	2	37	1	1	19		84	1	21	
Missing Operation			7	2	14	4	17	6	6	3	3	23	4	14	7	7	2
Missing Association			8						13			12		8		8	
Incorrect Operation	1	6		9		9		12		13			8		7	2	9
Incorrect Parameter	3		27	1	29		58	2	16	1	1	82	1	44	2	20	1
Number of Reported Faults	65/ 92	20/ 20	63/ 85	21/ 21	105/ 168	25/ 27	118/ 166	24/ 24	82/ 134	27/ 30	145/ 196	22/ 22	165/ 255	25/ 25	64/ 110	21/ 21	
FDR	0.71	1.00	0.74	1.00	0.63	0.93	0.71	1.00	0.61	0.90	0.74	1.00	0.65	1.00	0.58	1.00	
Number of Reported Fault Types	5/6	3/3	4/5	6/6	6/7	4/4	4/5	5/5	5/5	7/8	6/8	7/7	5/7	7/7	6/7	6/6	
FTDR	0.83	1.00	0.80	1.00	0.86	1.00	0.80	1.00	1.00	0.88	0.75	1.0	0.71	1.00	0.86	1.00	

Effectiveness based on Rate of Fault detection

Since all Sig. values for Shapiro-Wilk tests were 0.219 for FOM and 0.001 for HOM, these variables have a non-normal distribution (<0.05 for HOM) (see Table 8.14).

Table 8.14. Shapiro-Wilk Normality Tests

Mutant Type		Shapiro-Wilk		
		Statistic	df	Sig.
RFD	FOM	.878	7	.219
	HOM	.648	7	.001

Given that the variables were non-normally distributed and that we considered both mutant types as independent groups, the Mann-Whitney U Test was used to test our first null hypothesis (H_{10}). Figure 8.10 shows the box-plot containing data on the rate of fault detection per mutant type.

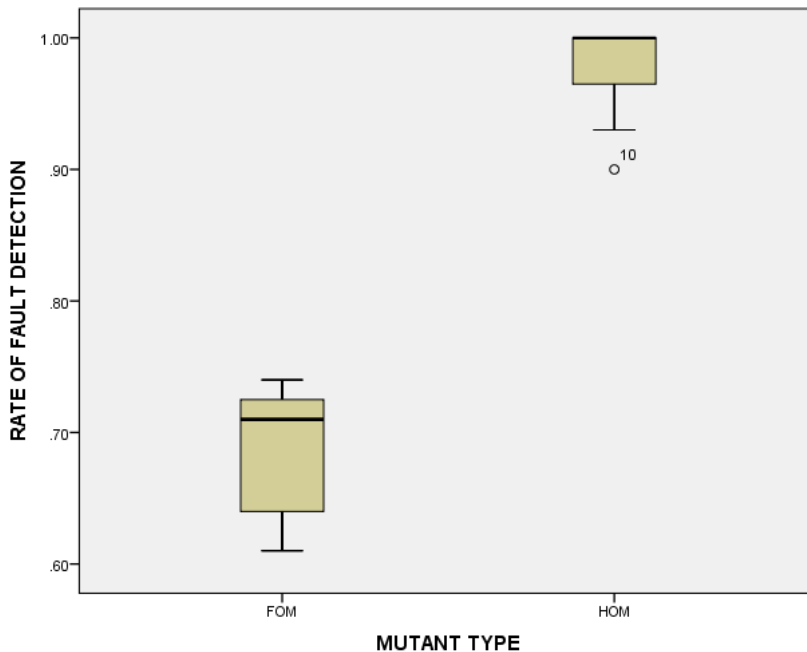


Figure 8.10. Box-plot for Rate of Fault Detection by Mutant Type

Table 8.15 shows the result of the Mann-Whitney U Test. Regarding the significance test (i.e $0.001 < 0.05$), we stated that the hypotheses H_{10} is rejected. In other words, *“the rate of Fault Detection is different for each mutant type”*.

Table 8.15. Mann-Whitney U Test for Rate of Fault Detection by Mutant Type

	Rate of Fault Detection
Mann-Whitney U	.000
Wilcoxon W	36.000
Z	-3.456
Asymp. Sig. (2-tailed)	.001
Exact Sig. [2*(1-tailed Sig.)]	.000 ^a

a. Not corrected for ties

Effectiveness based on Rate of Fault Type detection

Since all Sig. values for Shapiro-Wilk tests were 0.520 for FOM and 0.0 for HOM, these variables have a non-normal distribution (< 0.05 for HOM) (see Table 8.16).

Table 8.16. Tests of Normality of Shapiro-Wilk

Mutant Type		Shapiro-Wilk		
		Statistic	df	Sig.
Rate Fault Type	FOM	.930	8	.520
Detection	HOM	.418	8	.000

Given that the variables were non-normally distributed and that we considered both mutant types as independent groups, the Mann-Whitney U Test was used to test our second null hypothesis (H_{20}).

Figure 8.11 shows the box-plot containing data on the rate of fault type detection per mutant type.

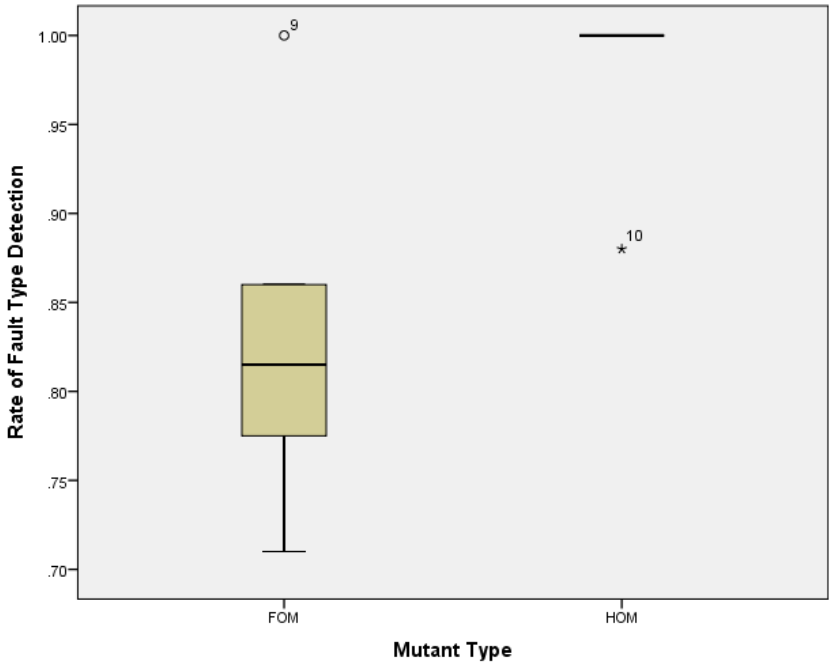


Figure 8.11. Box-plot for Rate of Fault Type Detection by Mutant Type

Table 8.17 shows the result of the Mann-Whitney U Test. Regarding the significance test (i.e $0.02 < 0.05$), we stated that the hypotheses H_{20} is rejected. In other words, “the rate of Fault Type Detection is different for each mutant type”.

Table 8.17. Mann-Whitney U Test for Rate of Fault Type Detection^a

	Rate Fault Type Detection
Mann-Whitney U	4.500
Wilcoxon W	40.500
Z	-3.090
Asymp. Sig. (2-tailed)	.002
Exact Sig. [2*(1-tailed Sig.)]	.002 ^b

a. Grouping Variable: Mutant Type

b. Not corrected for ties.

Test Suite Adequacy

In ERQ2, we aimed to verify whether the mutation score of CoSTest test suites was the same for killing the different mutant types. To do this, we compared the mutation score for HOMs and FOMs in the eight different CS subjects.

Table 8.18 shows the mutation scores summarized for each CS subject and mutant type.

Table 8.18. Mutation Score by Mutant type

Element	VC	MT	SG	ER	OCR	SS	PA	IM
FOM	0.87	0.80	0.75	0.90	0.75	0.82	0.75	0.74
HOM	1.00	1.00	0.89	1.00	0.96	1.00	1.00	1.00

Table 8.19 and Table 8.20 show the detailed mutation scores for each CS Subject and mutant type (FOM and HOM) respectively.

Figure 8.12 depicts the box-plot of our collected data for mutation score per mutant type. As the results show, the values of mutation score gave a better value for HOM than for FOM.

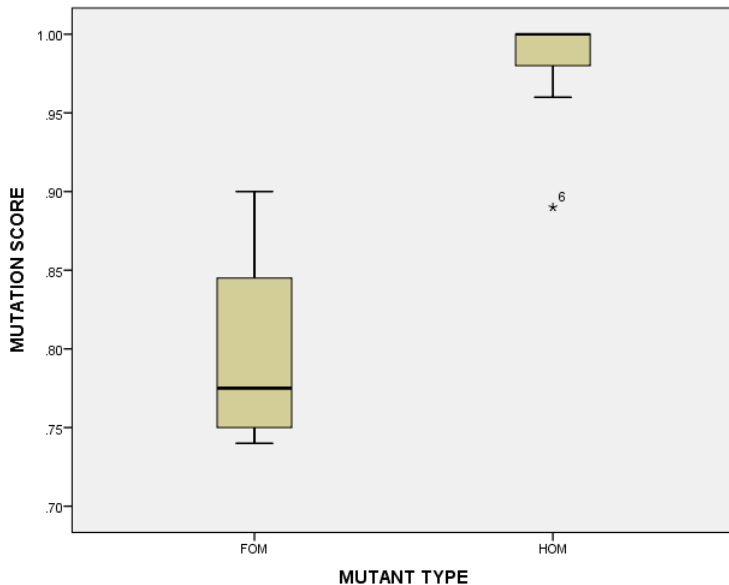


Figure 8.12. Box-plot for Mutation Score by Mutant Type

Table 8.19. Mutation Score of CoSTest Test Suites for First Order Mutants

MO	CS			VC			MT			SG			ER			OCR			SS			PA			IM							
	K	S	MS	K	S	MS	K	S	MS	K	S	MS	K	S	MS	K	S	MS	K	S	MS	K	S	MS	K	S	MS					
UPA	6	0	1.00	19	0	1.00	24	0	1.00	16	0	1.00	32	0	1.00	30	0	1.00	13	0	1.00	13	0	1.00	13	0	1.00	13	0	1.00		
WCO 1	0	2	0.00	6	1	0.86	6	3	0.67	1	0	1.00	0	3	0.00	18	15	0.55	18	15	0.55	18	15	0.55	18	15	0.55	18	15	0.55		
WCO 3	1	0	1.00				4	1	0.80				1	1	0.50	13	0	1.00				13	0	1.00								
WCO 4	2	0	1.00				7	8	0.54				2	0	1.00	20	6	0.77				20	6	0.77								
WCO 5	1	0	1.00				6	5	0.55				8	3	0.73	6	0	1.00				4	1	0.80	23	0	1.00					
WCO 6	3	0	1.00				4	8	0.36				2	0	1.00	5	0	1.00				4	0	1.00	20	0	1.00					
WCO 7							1	0	1.00																							
WCO 8	4	0	1.00				29	12	0.71				20	0	1.00	21	2	0.91				20	8	0.71								
WCO 9	0						1	0	1.00																							
WAS1	2	0	1.00													7	0	1.00				6	0	1.00	5	0	1.00	4	0	1.00		
WAS2	0	4	0.00				0	11	0.00				0	8	0.00	0	10	0.00				0	9	0.00	0	19	0.00	0	4	0.00		
WAS3	0	6	0.00													0	21	0.00				0	18	0.00	0	15	0.00	0	12	0.00		
WCL1	5	0	1.00				11	0	1.00				7	0	1.00	10	0	1.00				9	0	1.00	15	0	1.00	6	0	1.00		
WOP2	1	0	1.00				8	0	1.00				17	0	1.00	6	0	1.00				23	0	1.00	15	0	1.00	7	0	1.00		
WPA	1	0	1.00				9	0	1.00				17	0	1.00	3	0	1.00				26	0	1.00	12	0	1.00					
MCO	5	0	1.00				11	0	1.00				15	0	1.00	13	0	1.00				11	0	1.00	12	0	1.00	0	8	0.00		
MAS	2	0	1.00													7	0	1.00				6	0	1.00	5	0	1.00	0	4	0.00		
MPA	1	0	1.00				11	0	1.00				23	0	1.00	6	0	1.00				32	0	1.00	18	0	1.00	7	0	1.00		
All	8	1	0.87	68	17	0.80	123	45	0.75	149	17	0.90	101	33	0.75	161	35	0.82	191	64	0.75	80	29	0.74								

Table 8.20. Mutation Score of CoStest Test Suites for High Order Mutants

CS	VC			MT			SG			ER			OCR			SS			PA			IM		
	K	S	MS	K	S	MS	K	S	MS	K	S	MS	K	S	MS	K	S	MS	K	S	MS	K	S	MS
WCO2	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00
WGE					1	2	0.33						2	1	0.67									
WAT1	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00
WAT2	2	0	1.00				3	0	1.00	3	0	1.00	1	0	1.00	1	0	1.00	3	0	1.00	3	0	1.00
WAT3	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00
MGE							3	0	1.00				3	0	1.00									
MCL	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00
MAT	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00
MOP	3	0	1.00	3	0	1.00	2	1	0.67	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00
All	20	0	1.00	18	0	1.00	24	3	0.89	21	0	1.00	24	1	0.96	19	0	1.00	21	0	1.00	18	0	1.00

As in the analysis (ERQ1), Shapiro-Wilk tests were performed for each mutant type related to the adequacy of the test suites. Since the value of Sig. was >0.05 (0.100), this variable had a normal distribution. However, for HOM the Sig. value was 0, which meant this variable did not have a normal distribution (see Table 8.21). Considering both mutant types as independent groups, we selected the Mann-Whitney Test (non-parametric test) to evaluate the hypothesis.

Table 8.21. Shapiro-Wilk Normality Tests

Mutant Type		Shapiro-Wilk		
		Statistic	df	Sig.
MUTATION	FOM	.852	8	.100
SCORE	HOM	.576	8	.000

Table 8.22 shows the result of the Mann-Whitney U Test. The Sig. value obtained with this test was $0.01 < 0.05$, which meant that we rejected the null hypothesis H_{30} and concluded that *“The test suite adequacy (mutation score) is different for different mutant types”*.

Table 8.22. Mann-Whitney U Test for Mutation Score by Mutant Type^a

	MUTATION SCORE
Mann-Whitney U	1.000
Wilcoxon W	37.000
Z	-3.353
Asymp. Sig. (2-tailed)	.001
Exact Sig. [2*(1-tailed Sig.)]	.000 ^b

a. Grouping Variable: Mutant Type

b. Not corrected for ties.

8.4.8 Discussion

Our main results regarding CoSTest’s effectiveness and the adequacy of the test suites are the following: mutant type can influence these two variables, with better effectiveness and test suite adequacy in high order mutants than in first order mutants. This means

that test suites generated by CoSTest are effective at killing a large number of mutants. However, there are fault types that our test suites cannot detect, as explained below.

The mutants generated by the WAS2 mutation operator (changes the association type, i.e. normal, composite) and WAS3 mutation operator (changes the member end multiplicity of an Association, i.e. `*..*`, `0..1-0..1`, `*-0..1`) cannot be killed (mutation score=0) by an adequate traditional mutation test set.

Also, the fault types Incorrect Constraint and Incorrect Generalization injected by the mutation operators WCO1, WCO3, WCO4, WCO5, WCO8 and WGE were hard to detect (mutation score <0.7). This showed the weakness of test cases in testing some constraints, such as derivation rules, which needed to be executed in reverse order when there was a relation between classes that affected the computed result. For example, they first calculated the total of the expense report and then the total of the expense report details. This means these test cases will have to be improved.

Additionally, we found that a lower mutation score for some mutants related with constraints (WCOx) was because the test suites only consider coverage at element level and not at constraint level (i.e. condition branch).

We therefore plan to include test cases with values to make sure that different conditions (e.g. `>` vs `>=`) will be tested. However, the coverage analysis is important to detect defects when the assertions assert only return values and not side effects (see Figure 8.13) in which the coverage analysis is reduced, but all tests still pass.

```
if (number>0) {  
    //operation(number); //what if missing (MOP)  
    return true;  
} else { return false; }
```

Figure 8.13. Example of an assertion conditional

In addition, we found that CoSTest test suites do not test whether the cardinalities of the association ends meet a certain limit (only creating links according to the test scenario) thereby leading to missed faults, such as an Incorrect Association injected by the WAS3 mutation operator. As well as changing a navigable association to a shared aggregation or vice versa (WAS2) generates an equivalent mutant because “aggregation=shared” has no semantic effect in an executable model using ALF. Thus, another validation technique is required to validate these elements’ properties (i.e. inspection of the CS).

Finally, one of the strengths of CoSTest test cases is that it can detect types of defect about misunderstanding requirements (i.e. Missing and Unnecessary types) that are not normally detected at the CS level, by generating test cases based on user requirements. In a previous work [43] we found a tendency to report only defects related to verification, such as “Wrong” type (e.g. incorrect) rather than defects related to validation.

8.4.9 Analysis of the Threats to the Validity of the Results

There are several threats that potentially affect the validity of our study including threats to internal validity, threats to external validity, threats to construct validity and threats to conclusion validity.

Threats to internal validity are conditions that can affect the dependent variables of the experiment without the researcher’s knowledge. In our study, the selection of mutation operators is the main threat to internal validity. According to Andrews et al. [111], when using carefully selected mutation operators and after removing equivalent mutants, the mutants can provide a good indication of the fault detection ability of a test suite. Therefore, in order to minimize this threat we used an automatic process [124] to inject faults systematically, by avoiding non-valid and equivalent mutants and optimizing the testing coverage. This tool implements the mutation operators defined in a previous work [115].

Threats to external validity are conditions that limit the ability to generalize the results of our experiments to industrial practice. This threat is reduced by using seven CS of different sizes (see Subject CS Section) and domain (e.g. information systems, games). Moreover, a CS was taken from industry, some well-documented CS were found in the literature (i.e. [131], [133] and [132]), and others (i.e. ER, OCR, and VC) were selected because they contained the relevant CS elements required to inject the faults.

Threats to construct validity refer to the suitability of our evaluation metrics. We used well-known metrics to measure the effectiveness (rate of number of faults and number of detected fault types) [146] and the adequacy of the test suites (mutation score) [147]. We therefore believe there is little threat to the construct validity.

8.4.10 Conclusions and Changes to the Tool

The experiment let us to evaluate empirically the test cases generated by CoSTest tool V1.1 with respect to its effectiveness in terms of its fault detection in Conceptual Schemas.

Fault detection effectiveness was measured in terms of rate of faults detection and their causes (fault type) by the test suites. Test suite adequacy was measured in terms of the mutation score value. Our evaluation included the analysis of the variables for mutant types (FOM and HOM).

This experiment demonstrated that the effectiveness of the CoSTest test suites was affected by the mutant type and better results were obtained in detecting faults in HOM. These results suggest that the CoSTest technique is robust in detecting types of defects that are not normally detected at the CS level. However, some mutation operators achieved a value lower than 0.7 in the mutation score. These results suggest that the test suite should include a test for certain characteristics of CS elements, such as associations, and improve the coverage at the constraint level in order to enhance the effectiveness of the test suites.

Finally, some of the main changes applied to the tool including the ability to execute in reverse order some constraints, such as derivation rules, which needed to be executed in reverse order when there was a relation between classes that affected the computed result. This means these test cases were improved. We also implemented a report to track the generated test cases in a way that helps to locate the test cases and to detect if any are repeated.

8.5 Evaluating CoSTest User Perceptions

The following is a description of two experiments (i.e. pilot and industrial cases) to evaluate two properties of CoSTest: usefulness and ease-of-use CoSTest for which we recruited a set of users from both the university (i.e. pilot test) and industry.

Our main motivation was to validate CoSTest in real-world conditions as CoSTest had been conceived and evaluated only in laboratory experiments. Therefore, we aimed at discovering in a real world observational case study what kind of practical interpretations can be obtained from practitioners to identify areas of possible improvements, to explore general problems detected by the users and to define generally applicable solution strategies.

8.5.1 Experiment Research Goal

Following the template for goal definition that is suggested in [142], the goal of this study can be summarized as follows:

Analyze CoSTest
For the purpose of evaluation
With respect to usefulness and ease-of-use
From the point of view of the researchers
In the context of university and industry

8.5.2 Research Methodology

This study is an observational case study of a real-world case without performing an intervention. As a result of the case study-based research experience, we are going to collect many types of evidence: words, statements, documents, etc. that may be replicated in, or

generalised by similarity to the context of small and medium software companies where there are UML CD-based conceptual schemas to be validated. However, the context, time, participants, and problem will be different. Thus, the evidence will be linked together to support our conclusions on the user perceptions of CoSTest.

8.5.3 Experiment Context: The everis' Study Case

The case study company is everis¹⁰, a multinational firm offering business consulting, as well as IT development, maintenance and improvement in different domains and platforms (e.g. mobile, desktop embedded, web-based applications). everis is carrying out a project to improve a service-oriented architecture (SOA) platform for e-government. Within the public administration sector, everis has wide experience in projects related to modernization of public procurement management, education, e-government, health, justice, etc. everis has developed several electronic services provided by several Spanish municipal councils to citizens and companies (e.g. marriage registration application, public pool booking, taxes).

In order to compete on an international scale, everis is constantly looking for ways to reduce the time to market and increase the quality of its software products. However, they do not run CS-level tests but they do use tests of usability, integration, system, regression, acceptance and unit. In addition, everis uses a manual technique to generate the test cases from use cases, so that they require new techniques for systematization and automation of testing throughout the software and system life-cycle.

By applying the CoSTest validation, it is possible to perform early testing that facilitates the detection of defects in conceptual schemas and prevents defects from being transferred to the code, which contributes to the assurance of the quality of the product and optimizes the use of resources (e.g. time, budget) required in the

¹⁰ <http://www.everis.com>

process of software development. everis is thus a real-world environment in which the CoSTest validation can be applied.

8.5.4 Experiment Research Questions

In the following, we formulate two experiment research questions (ERQ) that guided the experiment that was performed in this study and briefly describe how we plan to gather the data to answer the question. The overall approach is based on interviews with users and observing their behaviour while interpreting defects report generated by CoSTest.

ERQ1. When the subjects are validating a UML CD-based CS with CoSTest's reports, what is their impression of its perceived usefulness?

ERQ2. When the subjects are validating a UML CD-based CS with CoSTest's reports, what is their impression of its perceived ease-of-use?

To answer these questions, we measure the perceived usefulness and perceived ease-of-use of the CoSTest tool in the future. Besides collecting evidence from interviews and observation we plan to assess the perceived usefulness for everis testers by means of a 7-point Likert scale questionnaire [148].

8.5.5 Case Selection

For this case study, we took one unit that is part of the everis' SOA development platform: the project management office (PMO). We selected this CS for our case study for two main reasons. First, it represents a simple, understandable, and realistic scenario that includes enough elements for the complete application of our validation framework. Second, a document with a specification of requirements using communicational analysis was available during the case study.

A PMO is the department or group of designated people in charge of defining the best practices and standards for project management in the portfolio of projects of an organisation or collaborative

environment. As such, the PMO can use and establish several tools, depending on the nature of the environment and the type of projects. In this case study we are focused on the incident management process of the PMO. Incidents emerge when the eGoveris' users have problems using the platform. eGoveris was created to deliver an eGovernment solution for local councils based on an SOA paradigm.

PMO offers several services to their customers and wants to improve them, but does not know how. It does not have enough technical knowledge to accomplish this, so it needs an external provider. Several companies would interact with the PMO through a contract procedure defined by the end customer. This implies that a change in the service provider, according to different context conditions, influences the value delivered to customers, activities and service provisioning. If the PMO has technicians with technical knowledge, it would be the external provider also. Otherwise, it needs to hire the services of an external provider in order to supply the lack of technical knowledge. In this case, everis is the external provider who defines the best way to apply a solution according to the requirements specified by the PMO. As a result, the PMO is modified to incorporate the proposed methodology (for further details see Appendix B).

8.5.6 Methods of Data Collection

The Technology Acceptance Model, or TAM [149] is one of the most influential usability questionnaires. According to the TAM, the primary factors that affect a user's intention to use a technology are his/her perceived usefulness and perceived ease-of-use. Actual use of technologies is affected by the intention to use, which is itself affected by the perceived usefulness and usability of the technology. A number of studies support the validity of the TAM and its satisfactory explanation of end-user system usage [150].

Thus, we used two standard questionnaires widely applied for evaluating usefulness and ease-of-use in a subjective manner [151]. The method selected to collect data was the interview.

Perceived usefulness

The extent to which a person believes a technology will enhance job performance [151]. This variable is measured using a 7-point Likert scale format to obtain users' perception. We also asked the everis' stakeholders which improvements are required to improve its usefulness.

Perceived ease-of-use

The extent to which a person believes that using the technology will be effortless. The stakeholders' perceived ease-of-use will be evaluated by means of a 7-point Likert scale questionnaire. In addition, we asked the everis' stakeholders which improvements are required to improve ease-of-use.

8.5.7 Experimental Subjects

The subjects that participated in this experiment were:

- A research and developer manager, who has 12 years of experience in the IT sector and that has led several innovation projects. This role has a mixture of knowledge about the SOA platform, development tools, and also of the results expected by public bodies.
- A junior developer and tester with five years of experience in testing processes generating test cases in JUnit. She was willing to validate the Conceptual Schema in some projects and had little initial knowledge of Communicational Analysis specification.

8.5.8 Instrumentation

We designed a set of instruments to train the subjects, collected data from the experimental task and also facilitated the subsequent data analysis. For the training in the CoSTest tool, we provided digital and textual material, such as requirements model, conceptual schema as well as a CoSTest demonstration video.

We noticed that the provision of the demonstration video was a very good motivation for learning.

For the experimental task we designed a task description document and templates to collect data about defects identified and corrected by the subjects.

Further information about the instrumentation can be found in Appendix C.

8.5.9 Experimental Procedure

Figure 8.14 presents an overview of the experimental procedure.

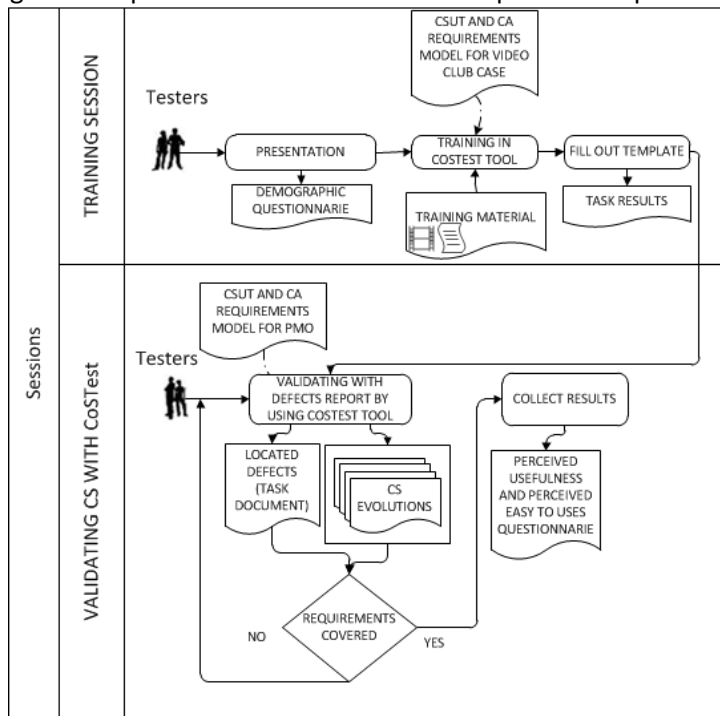


Figure 8.14. Experimental Procedure

The session was carried out in a meeting room in the everis' offices. A detail of the activities during the session is specified in the following Table 8.23. The detailed material is included in Appendix C.

Table 8.23. Detail of the Activities

Activity identification	Description
Training Session	
A1.1	Presentation of the activities to be performed during the two sessions. The objective is to describe the activities that to be performed during the session.
A1.2	Demographic questionnaire.
A1.3	Training in the use of CoSTest tool.
A1.4	Provide the subjects with textual material specifying the requirements model, conceptual schema and the task descriptions.
A1.5	Subjects fill out the template to take notes during the execution of the tasks.
Experimental Session	
A2.1	Provide the subjects with textual material specifying the requirements model, conceptual schema and the task descriptions.
A2.2	Subjects fill out the template to take notes during the execution of the tasks.
A2.3	Subjects fill out the MEM questionnaire.

8.5.10 Pilot Test

In order to verify that all the experimental material was correct and would not cause problems during the data collection, a pilot test was run on June 2016 as a Testing course in the University of San Agustín of Arequipa, Perú. This course consisted of two sessions (Friday and Monday) of four hours each session.

Objects

In this experiment we used two small UML/ALF conceptual schemas:

1. A CS of a video club (VC) system, introduced in Chapter 5; which contains information about the movies and the partners registered in the system (both of them must be registered by the salesman (supervisor of the system) before being able to use all the functionalities. Each movie is assigned to only one video club. Also, each videoclub holds its rents.

2. A Photography Agency (PA) system that represents the functionality of a photography agency that covers the management of photographers (e.g. application, selection, promotion) and publishing houses (e.g. subscriptions), as well as the management of regular reports (they are provided by photographers and become part of the agency catalogue, then publishing houses order them) and of exclusive reports (they are first requested by a publishing house and then assigned to a photographer); the delivery of both types of reports and the corresponding invoicing to publishing houses and payment to photographers are also within the scope of the case. This case is used to illustrate Communication Analysis in [21].

Participants

Software modellers/testers were the population of interest for this study; in practical settings, they are the designers of conceptual schemas and often work as testers.

The study does not require expert developers, but the subjects must have basic knowledge in software development: design of conceptual schemas, some languages and tools that support software development, and execution of testing in development projects. Additionally, we required them to be familiar with Eclipse and UML2.

A total of twenty-five people participated in our pilot experiment. Three participants were industry practitioners and the others were Computer Engineering Degree students from the University of San Agustín of Arequipa, Perú.

All the participants had a good background in modelling in UML, model-based testing and good testing and programming skills (using object oriented languages such as Java or C++).

Tasks

The participants were asked to carry out the tasks of the two planned sessions (Table 8.23). Both tasks were part of a mandatory activity (which also contained other tasks) that all students had to deliver to pass the Testing course.

Prior to carrying out the tasks, the participants were introduced to CoSTest in the form of two demo videos.

The activity was composed of eight tasks and three questionnaires (demographic, tasks template and post-task) to evaluate the CoSTest tool and covered two four-hour sessions (friday and monday). During this period the students were able to work collaboratively, ask the teacher questions and search for any kind of information to help in solving the proposed tasks.

Lessons Learnt

We performed a pilot test in order to test the material (presentation, requirements specification, conceptual schemas, task templates, questionnaires, required time, and so on).

The pilot test was performed according to the schedule of the course and we adapted the material to keep the original objectives. As a result of our pilot test, several improvements have been added, mainly consisting of the following:

1. Include a clear description of the requirements for the installation of the tool because the tool has problems with a more advanced Java version than version 7.
2. Update the task template to collect data by eliminating the timing record because the time in each iteration varies depending on several factors such as the complexity of the defect type to be corrected and the skills of the subjects in managing the modelling tool (i.e. UML2 or Papyrus tool).
3. Include an error log in the tool; this suggestion was made by an industry practitioner.

4. Highlight the failed test cases in red to differentiate them from those that are passed successfully.
5. Show all defects; this suggestion was not addressed because the testing process is incremental.

These observations were useful to update the experiment material and to be able to apply it in a more understandable way.

8.5.11 Analysis of the Threats to Validity

There are several threats that potentially affect the validity of our study including threats to internal validity, threats to external validity, threats to construct validity and threats to conclusion validity.

Conclusion Validity

Threats to conclusion validity are concerned with issues that affect the ability to draw valid conclusions about relations between the treatment and the outcome of an experiment. Threats to the validity of conclusions are typically due to low statistical power. As the method used in this research project is purely qualitative, we consider that this kind of threat does not apply here. As a result, we avoid making any conclusions from a generalisation made by inference from observations made during the research. In addition, we address the “Fishing for a specific result” by two methods (i.e. questionnaires and interviews) to ensure consistent results.

Internal Validity

Threats to internal validity are conditions that can affect the dependent variables of the experiment without the researcher’s knowledge.

In our study, the selection of mutation operators is the main threat to internal validity. According to Andrews et al. [111], when using carefully selected mutation operators and after removing equivalent mutants, the mutants can provide a good indication of the fault detection ability of a test suite. Therefore, in order to minimize this

threat, we used a random selection of mutation operators to inject faults into the selected CS and avoiding non-valid and equivalent mutants.

Regarding the instrumentation threat, we reduced this threat by validating the instruments used in the study by means of a pilot test. Another threat deal with was maturation, which implies that subjects may react differently as time passes (e.g., due to boredom or tiredness). To minimize this threat, we selected a set of tasks that allowed the subjects to finish them in less than two hours. Finally, social threats were avoided because the tasks were individual and the subjects were not allowed to talk to each other about the tasks. Also, since they were not aware of the experimental research goal, this they did not affect their performance.

External Validity

Threats to external validity are conditions that limit the ability to generalize the results of our study to industrial practice. This threat is reduced by using a real case and involving all the engineers of the company concerned with the analysed CSs rather than using a random sample. In addition, this threat involves having an experimental setting that is not representative of industrial practice. To minimize this threat, we utilized tools that are commonly used in industrial environments (e.g., UML2 tools, the Eclipse platform).

Construct Validity

This threat focuses on whether the theoretical constructs are suitably interpreted and measured fore our evaluation metrics. We increased the reliability of subjective measures by using questionnaires with scales previously validated in other studies [148].

8.5.12 Answers to Experiment Research Questions

To answer the experiment research questions, we established a set of preliminary hypotheses. Table 8.24 presents the corresponding hypotheses.

Table 8.24. Specification of hypotheses

Null hypothesis	Statement: The application of CoSTest's report does not influence the subject ...
H1 ₀ (ERQ1)	... perceived usefulness of CoSTest in detecting faults in Conceptual Schemas
H2 ₀ (ERQ2)	... perceived ease-of-use of CoSTest in detecting faults in Conceptual Schemas

As there were only 2 subjects in the research we did not apply any statistical test to analyse, interpret the collected data or generalize. We analysed the responses of each subject for each experiment research question obtained from the aforementioned instruments containing the questionnaires filled in by the subjects.

Regarding ERQ1, the results obtained from the questionnaires show that both subjects agreed that CoSTest was useful for correcting the defects found in a CS. These positive results were reinforced by the qualitative feedback obtained during the interviews. All the subjects considered that CoSTest was useful, since it allowed them to perform the tasks more effectively; for instance, one subject stated: *"The tool seems very useful, it can help a lot in the creation of test cases and validation of conceptual schemas"*, while the other said: *"CoSTest reduces the possibility of omitting test cases"*. The usefulness of the test cases generation capabilities was also emphasized by some subjects *"Reduces the amount of effort required to produce all test cases in a systematic way"* and *"the feedback to localize and correct the defects is valuable"*.

Regarding ERQ2, the results obtained from the questionnaires show that both subjects think that the correction of defects using CoSTest is perceived as easy to use. These positive results were

reinforced by the qualitative feedback obtained during the interviews. All of the subjects considered that CoSTest was easy to use, since it allowed them to perform the tasks easily. Most subjects emphasized the execution method; for instance, one subject stated: *“Generating test cases with CoSTest requires just a few clicks to get them ... and the localization of defects is done in an easier and more direct way”*, while another subject said: *“When you execute CoSTest, the process to follow is quite intuitive”*.

8.5.13 Discussion

The subjective perception expressed by the subjects of the study indicates their willingness to accept and use CoSTest. They perceived CoSTest to be a useful and easy to use to generate test cases and correct the defects found in conceptual schemas. Further studies using CSs of different sizes and domains (e.g. information systems, games) will be required to generalize these results. However, this observational case study done in everis has taught us several lessons regarding putting CoSTest into practice of the and research. We would like to highlight the following:

Models are vital for the application of our validation framework

CoSTest is designed for validation of conceptual schemas using a model as the functional requirements specification. The generation of test cases is based on a model-driven paradigm. In this context, the two subjects of the study were highly satisfied with CoSTest level of automation. This level was achieved thanks to model transformations, which reduce the complexity of test case generation by automating the process. This is in line with the benefits of MDE: the reduction of complexity by means of the automation of labour-intensive and error-prone tasks [152]. Therefore, the assistance provided by CoSTest allowed the subjects to perform the validations without deviations, and this led to a significant increase in usefulness and ease of use perception. In this context, the everis’ developer/tester said: "Although CoSTest uses an interesting strategy to validate conceptual schemas,

the industry needs to adopt the MDD paradigm, particularly the Communicational Analysis method, which may require time in cases of low modelling experience". However, they indicate that they do not want to miss the advantages of CoSTest and are receptive to its use; "I am interested in investing time in modelling, seeing the benefits it has in the tests and correction of defects phase".

In future research studies on the advantages of CoSTest in real projects is needed to convince more companies like everis to apply Communication Analysis to take advantage of this requirements method in their projects.

In order to reduce this barrier and to show the facilities of CoSTest, we plan to improve our work in two ways. First, we will increase our repository (<https://staq.dsic.upv.es/webstaq/costest.html>) containing examples of conceptual schemas and requirements models that can be validated by CoSTEst. Secondly, we will incorporate a way to specify the requirements using a textual specification, so that the use of both types of specifications can be evaluated and compared.

An open source tool is required for adoption

The subjects participating in this research emphasized developing open-source and free solutions as a means of allowing free access for experimentation and reduce the cost of adoption. In addition, they think that the development of tools based on industry-accepted open platforms, such as Eclipse, has provided benefits, such as easier integration. Therefore, we plan to invite more companies to use our tool and to probe its benefits.

The use of the ALF language is required

Since our validation tool uses ALF as the language to generate test cases and execute them (see Chapter 5), the testers need to know the syntaxes and semantics of the ALF language to edit or modify the test cases involving some complex negative constraints.

Since the case study used in this research does not require modifying the generated test cases, this knowledge was not required. However, it can be varied depending on the complexity of the formalized stories, according to each testing objective. These difficulties can be mitigated by enhancing CoSTest with appropriate assistance for updating/editing its test cases. To do this, we plan to include a set of guidelines that will free users from having to be ALF experts, allowing them to create/update test cases following a set of intuitive steps.

Finally, we observed that as ALF is a script language, it was familiar to the subjects.

Validation should be extended at code level

The subjects also considered that they would like CoSTest to be able to generate test cases using other programming languages (e.g. java, C#) at code level. In this way the model-driven process for generating test cases can be used for two levels of abstraction: model and code level. To do this, we plan to include an option that will allow test cases to be generated for execution in Java language [153] using JUnit test cases [154]. This result is in line with one of the most widely recognized benefits of MDD: development of Platform Independent Models (PIMs) that have a long lifespan and may be ported to multiple platforms or languages [152].

8.6 Summary and Conclusions

In this chapter we have reported six experiences in order to evaluate and validate the CoSTest framework. We performed two comparative laboratory experiments to evaluate the transformation rules used in CoSTest, generating the test cases and CSUT, two mutation-based laboratory experiments to evaluate the mutation operators implemented in the tool. This was done to identify the test cases that should be prioritized in CoSTest as well as to evaluate the effectiveness of the CoSTest test cases. A mutation-based laboratory experiment was used to validate CoSTest effectiveness in killing

mutants as well as the defects detected in these mutants. The last evaluation experience included an observational case study to gather user perceptions on using CoSTest for correction of defects.

Mutated CSs are like virtual laboratories where injected defects can be detected, and test cases and corrective procedures can be experimented with before they are used and implemented in the real system. Experience from applications in other fields than software engineering indicates that significant benefits can be drawn from introducing the use of mutation for management decision support. Mutation-based software engineering laboratories can help focus experimentation in both industry and academia for this purpose, while saving effort by avoiding experiments in real-world settings that have little chances of generating significant new knowledge.

The results of the first two comparative experiments to validate the model-to-text and the first two model-to-model transformations helped the researchers to improve the tool support as well as to identify the transformation rules that should be improved.

The results of the next two mutation-based experiments suggest that the CoSTest mutant generator is effective and efficiency in generating first order mutants using the 18 mutation operators defined for this purpose.

The results of the fifth experiment suggest that most of CoSTest's test cases are quite effective (i.e. detection ratio > 70%) in detecting defects at the CS level. However, some test cases achieved a value lower than 0.7 in the mutation score. These results suggest that the test suite should include a test for certain characteristics of CS elements, such as associations, and improve the coverage at the constraint level in order to enhance the effectiveness of the test suites.

The results of the observational case study are also encouraging. All of the subjects agreed, or strongly agreed, about each of the items of the usefulness scale. We also obtained positive results for perceived

ease-of-use. These subjective results were reinforced by positive results about the intention of the subjects to use the tool. We believe that these results were obtained thanks to the use of MDD techniques (such as metamodeling, model transformations and independent platform), which reduce the complexity of the four main phases of the test cases generation process: design, generation, execution and evaluation.

In contrast to these positive findings, we also found several challenges that are inherent to CoSTest usage. With the aim of providing better tool support for model-driven testing, we will address these challenges in the near future. For instance, as Section 8.5.13 describes, we will incorporate support for a textual specification of requirements, include a help that enables guided test case edition, enhance the validation at code level and allow free access to our validation tool. The main goal of these enhancements is to facilitate the adoption of CoSTest in the industry.

PART V.

FINAL DISCUSSION

Chapter 9

FINAL DISCUSSION

Unlike traditional Software Development in which the software is the main artefact, the main artefact in MDE is a model (conceptual schema). Conceptual modelling is an essential activity in the requirements phase of the software development life cycle, which is aimed at eliciting, specifying and validating the conceptual schema of an information system (Chapter 1). The aim of Conceptual Schema Validation is to check the alignment between the knowledge specified in the CS and the stakeholder's expectations.

9.1 Summary of the Contributions of this Thesis

This thesis has presented a Testing-Based Validation Framework for Conceptual Schema in a Model-Driven Environment as a contribution to the challenge of conceptual schema validation. We describe how to use each framework method and how they are integrated. The contributions of the thesis consist of the evidence for the achievement of the research goals, as well as the answers to the established research questions as described below:

Contribution 1. Establishment of the fundamentals for our validation framework, which are very important because they establish the

requirements and challenges addressed in the thesis (Chapters 3 and 4). This is a knowledge contribution related with the RQ1 (see Section 2.3) and it is based on the existing state of knowledge in both the problem and solution domains for the research opportunity under study.

Thus, we have described the fundamentals of conceptual schema testing in a model-driven environment (Chapter 3). We have explained the main quality models and validation practices to improve the quality of conceptual schemas (Chapter 4). In addition, some concepts were further defined in Chapters 5 and 6, which helped the researchers to establish the requirements and challenges to be faced in this thesis. These concepts and challenges are related to the design of the CoSTest framework methods (Chapters 5 and 6).

Contribution 2. This contribution is very important because provides a new validation framework to improve the quality of the conceptual schemas in a model-driven environment (Chapters 5 and 6). This is the main research contribution of the thesis and it is related with the RQ2.2 (see Section 2.3).

We show how MDD techniques (such as metamodeling, and model transformations), improve abstraction, automation and reuse, which allows us to alleviate the complexity of our validation framework. So that, our framework supports four phases of the testing process: test design, test case generation, test case execution and the evaluation of the results. We described the work involved in designing each phase of the model-driven testing framework, as well as the decisions made to obtain the expected results. The design can be summarised as follow:

The test case generation is based on related works and knowledge from relevant solutions in model management, model-driven development and testing, such as Communication Analysis (a communication-oriented business process modelling and requirements method), model-to-model transformations, the classic pathfinder or graph traversal algorithm, and the OO-Method (object-oriented

model-driven development method). To generate the executable test cases and create the testing environment we selected a platform independent language (i.e. OMG Standard - ALF), that works at the same semantic level as the rest of the UML-based CS and can be consistently implemented across a number of tools, promoting the same sort of interoperability for textual behavioural specification that the UML standard already does for graphical modelling.

For the test selection and prioritization of test cases, we used mutation strategies in order to identify the types of defect that can be detected in the conceptual schemas using our testing strategy, as well as, the test cases that should be selected and prioritized. To generate the executable Conceptual Schemas, we applied model-to-test transformations to generate the ALF execution units and integrate them into our testing framework.

In order to make the corrective feedback understandable to the modeller/tester, the report generated by our framework identifies the defect type and the source of the problems and assists the modeller/tester to repair them, which was one of the goals of our proposal.

Contribution 3. Prototype that implements the validation framework supporting the facilities to test conceptual schemas (Chapter 7). This contribution is related with the RQ2.1 (see Section 2.3) showing how the proposed validation framework can be applied in practice and making ideas tangible to then transfer this proposal to industrial applications.

We have implemented a supporting tool (CoSTest) for automated generation of test cases and automated testing of conceptual schemas (Chapter 7). This tool contains the modules that manage and generate the executable test cases from requirements. These include a CSUT processor that transforms a conceptual schema into an executable CS, a test data-manager to concretize the test case values, and a test processor that coordinates the execution of the tests and reports the

found defects as well the elements covered by the test cases. Tests written in ALF Language may be automatically executed as many times as needed. We have also shown that our testing framework has been extended with the mutant generator in order to be able to deal with first-order mutant generation and provide the facilities to test them.

Contribution 4. Some experiences in evaluating and validating the CoSTest tool (Chapter 8). This is a knowledge contribution related to show how our validation framework works in practice; what are its limitations and the solution's effectiveness. This contribution is related with the RQ3 (see Section 2.2).

We validated the proposed framework in the context of Design Science Research, which was the framework adopted in this PhD thesis (Chapter 2). Various laboratory demonstrations were performed for some methods of CoSTest. We tested all CoSTest methods in a controlled laboratory environment and evaluated their feasibility before applying them to empirical tasks.

We validated the transformation rules used in the CoSTest model-driven strategy to generate the test cases by means of their application in a comparative experiment with cases taken from the literature and others selected with the relevant CS elements required to evaluate all the rules. The results helped the researchers to improve the tool support and to identify the transformation rules that need to be improved.

Since our validation framework includes the component to generate first order mutants of UML CD –based conceptual schemas, we evaluated some properties of the mutation operators used for generating mutants and also validated the effectiveness and efficiency of the mutant generation process. The results were positive in terms of the percentage of valid and non-equivalent mutants generated by the tool and the time that can be saved by using it.

We also evaluated CoSTest effectiveness by means of its application in a comparative experiment using mutation with cases taken from the literature and industrial practice and other cases selected because they contained the relevant CS elements required to inject the faults. The results helped the researchers to improve and extend the tool support as well as to identify the test cases that need to be improved and prioritized.

Finally, we evaluated the stakeholder's perceptions by using our tool support in the correction process of the defects found on UML CD-based in an industrial case (Chapter 8). The perceptions of the usefulness and ease-of-use of our tool are very positive and have provided ideas to be addressed in future work. We have seen that the main quality goal of conceptual schemas is completeness and that this may be improved by testing, and that other quality goals such as correctness, consistency, comprehensibility, confinement and changeability are also positively influenced. We have also shown that our testing framework can be used in combination with existing conceptual schema validation and verification techniques.

In summary, this thesis contributes new knowledge and artefacts to the software quality field and model-driven development. The evidence provided by the evaluations and all the validations and tool developments have pointed us in the right direction to further transfer this method to industrial applications.

9.2 Thesis Impact

9.2.1 Publications

Book Chapter

1. *Granda, M.F., Condori-Fernández, N., Vos, T. E. J., Pastor, O.*
A Model-Level Mutation Tool to Support the Assessment of the Test Case Quality – Lecture Notes Information Systems.
Publication: Print ISBN 978-3-319-52592-1, volume 22, 2017.

Journals

1. *Granda, M.F., Condori-Fernández, N., Vos, T. E. J., Pastor, O.*
Model Transformations Rules within a Model-Driven Testing Environment: Definition and Validation – Submitted to Software Quality Journal.
2. *Granda, M.F., Condori-Fernández, N., Vos, T. E. J., Pastor, O.*
CoSTest: A model-driven framework for validation of conceptual schemas – Submitted to Systems and Software Journal.

Conference Papers

1. *Granda, M.F., Condori-Fernández, N., Vos, T. E. J., Pastor, O.*
Effectiveness Assessment of an Early Testing Technique using Model-Level Mutants. Evaluation and Assessment in Software Engineering (EASE 2017). Core Index A. Karlskrona, Sweden, June 16, 2017.
2. *Granda, M.F., Condori-Fernández, N., Vos, T. E. J., Pastor, O.*
Mutation Operators for UML Class Diagrams. Advanced Information Systems Engineering - 28th International Conference (CAiSE 2016). Core Index: A. Publication: Print ISBN 978-3-319-39695-8, pp. 325-341. Ljubljana, Slovenia, June 13-17, 2016.
3. *Granda, M.F., Condori-Fernández, N., Vos, T. E. J., Pastor, O.*
A Model-level Mutation Tool to Support the Assessment of the Test Case Quality. 25th International Conference on Information Systems Development (ISD 2016). Core Index: A. Online ISBN 978-83-7875-307-0. Katowice, Poland, August 25-27, 2016
4. *Granda, M.F., Condori-Fernández, N., Vos, T. E. J., Pastor, O.*
What do we know about the defect types detected in conceptual models? 9th IEEE International Conference on Research Challenges in Information Science (RCIS 2015). Core Index: B. Publication: print ISBN 978-1-4673-6630-4, pp. 88-99. Athens, Greece, May 13-15, 2015.

Workshops Papers

1. *Granda, M.F., Condori-Fernández, N., Vos, T. E. J.*
Using ALF within the CoSTest process for Validation of UML-based Conceptual Schemas. 36th International Conference on Conceptual Modeling (ER2017). Valencia, Spain, November 8, 2017.
2. *Granda, M.F., Condori-Fernández, N., Vos, T. E. J., Pastor, O.*
Towards the automated generation of abstract test cases from requirements models. 1st International Workshop on Requirements Engineering and Testing (RET 2014). Online ISBN 978-1-4799-6334-8, pp. 39-46. Karlskrona, Sweden, August 26, 2014.
3. *Granda, M. F.*
An experiment design for validating a test case generation strategy from requirements models. 4th IEEE International Workshop on Empirical Requirements Engineering (EmpiRE 2014). Online ISBN 978-1-4799-6337-9, pp. 44-47. Karlskrona, Sweden, August 25, 2014.

Poster and Demo Tool

1. *Granda, M.F., Condori-Fernández, N., Vos, T. E. J., Pastor, O.*
CoSTest: A tool for Validation of Requirements at Model Level. 25th IEEE International Requirements Engineering Conference. (RE 2017). Lisbon, Portugal, September 7, 2017.
2. *Granda, M. F.*
Testing-Based Conceptual Schema Validation in a Model-Driven Environment. I Encuentro de Estudiantes de Doctorado de la Universitat Politècnica de València, Valencia, Spain, June 12, 2014.

Doctoral Consortium

1. *Granda, M. F.*
Testing-Based Conceptual Schema Validation in a Model-Driven Environment. Doctoral Consortium of the 25th International Conference on Advanced Information Systems Engineering (CAiSE 2013), Valencia, Spain, June 21, 2013.

9.2.2 Academic Project Participation

1. *CaaS: Capability as a Service in digital enterprises. European Project FP7-ICT 2009-5*. Reference: INFSO-ICT-257574. 2013-2016.
2. *IDEO: Innovative services for Digital Enterprises with ORCA (Servicios Innovadores para Empresas Digitales con ORCA)*. Reference: PROMETEOII/2014/039.

9.2.3 Research Stay

Erasmus Stay in the Department of Computer Science, Faculty of Sciences of VU University, Amsterdam, Netherlands. June - September 2014. Project: An experiment design for validating a test case generation strategy from requirements models.

9.3 A Work that Opens New Research Lines

The work carried out in the course of this thesis can be extended in many ways. In this section we suggest several directions for further research in this area, according to the three dimensions of our framework and the limitations found in the validation phase.

9.3.1 Domain

Regarding the kind of model to be validated, the conceptual schemas addressed in this thesis could be extended. In order to be more expressive, new types of constraints could be considered. Adding constraints means identifying their representation in ALF language. However, -as we pointed out in Chapter 5 – not all the possible constraints can be tested by our method.

The methods described in this thesis could also be applied to other types of executable models. In the context of UML, for instance, other model behaviour (such as activity diagrams or statechart diagrams) could be analysed in terms of the testing method addressed in this thesis.

9.3.2 Quality Goal

Regarding the conceptual schema quality goals some of these could be improved with our validation framework. In particular, as we stated in Chapter 3, the meaning of the consistency goal could be extended to consider not only the structural diagram appearing in the conceptual schema but also other behavioural diagrams (such as activity diagrams or statechart diagrams) reasoning over the consistency between the structural and behavioural parts.

9.3.3 Method

The validation framework should integrate with other verification methods to allow the validation of more complex and specific elements such as verifying weak and strong executability of the model operations [155].

Regarding framework inputs, two concrete research lines could be addressed:

- a) The first line consists of specifying the requirements with other types of models, for instance, BPM, i* or concept maps or a textual specification in order to extend the facilities to specify requirements for our validation framework.
- b) The second line consists of providing an automatic translation into an executable CSUT of other types of conceptual schemas compliant with UML, such as Integranova models [6] to allow designers to perform validation on these types of conceptual schemas.

In addition, we plan to develop and include in our tool a set of guidelines that will support users in creating/updating test cases following a set of intuitive steps. We plan to include in the tool an option that will allow test cases to be generated to be executed in Java language using JUnit test cases.

Further developments should be performed on the developed prototypes to make them more stable and usable. Currently, with

these tools we consider that it is possible to implement CoSTest in real world conditions. The open source provision of tools for CoSTest ensures the future execution of the engineering cycle to bring CoSTest to industry.

A set of guidelines should be proposed on the use of CoSTest, to provide useful advice to the conceptual modeler/tester in at least the most basic situations.

The proposed further work will help to extend our validation framework and make it more complete. Thus, we could perform a large-scale empirical study on several industrial subject CS to predict how the validation framework will improve the performance of stakeholders in their tasks of testing of conceptual schemas and evaluate if the use of CoSTest reduces the development costs and improve the quality of delivered software systems (see especulatives goals G7 and G8 in Section 2.2).

In summary, given the increasing importance of models in the most relevant software development methods currently in use, the validation of the requirements on such models is a research topic that needs further in-depth study.

REFERENCES

- [1] A. Olivé, *Conceptual Modeling of Information System*. Springer, 2007.
- [2] J. Johnson and A. Henderson, *Conceptual Models: Core to Good Design*. Morgan & Claypool, 2012.
- [3] R. France and B. Rumpe, "Model-driven Development of Complex Software: A Research Roadmap," in *International Conference on Software Engineering*, 2007, no. 2, pp. 37–54.
- [4] M. Staron, "Adopting Model Driven Software Development in Industry – A Case Study at Two Companies," *Model Driven Eng. Lang. Syst.*, pp. 57–72, 2006.
- [5] J. Hutchinson, M. Rouncefield, and J. Whittle, "Model-driven Engineering Practices in Industry," *Proc. 33rd Int. Conf. Softw. Eng.*, pp. 633–642, 2011.
- [6] O. Pastor and J. C. Molina, *Model-Driven Architecture in Practice*. Cambridge: Springer Berlin Heidelberg, 2007.
- [7] R. Van Der Straeten and T. Mens, "Challenges in model-driven software engineering," in *Model Driven Engineering Languages and Systems - MODELS 2008*, 2009, pp. 35–47.
- [8] A. Olivé and J. Cabot, "A Research Agenda for Conceptual Schema- Centric Development," in *Conceptual Modelling in Information Systems Engineering*, 2007, pp. 319–334.
- [9] P. Mohagheghi, V. Dehlen, and T. Neple, "Definitions and approaches to model quality in model-based software development - A review of literature," *Inf. Softw. Technol.*, vol. 51, no. 12, pp. 1646–1669, 2009.
- [10] J. Krogstie, *Model-Based Development and Evolution of*

- Information Systems: A Quality Approach*. 2012.
- [11] M. Genero, A. M. Fernández-Saez, H. J. Nelson, and G. Poels, "A Systematic Literature Review on the Quality of UML Models," *J. Database Manag.*, vol. 22, no. September, pp. 46–70, 2011.
- [12] I. Sommerville, *Software Engineering*, 9th edn. Boston: Addison-Wesley, 2011.
- [13] K. El Emam and G. A. Koru, "A replicated survey of IT software project failures," *IEEE Softw.*, vol. 25, no. 5, pp. 84–90, 2008.
- [14] A. Hevner, S. T. March, J. Park, and S. Ram, "Design Science in Information Systems Research," vol. 32, no. 4, pp. 725–730, 2008.
- [15] R. Wieringa, *Design Science Methodology for Information Systems and Software Engineering*. 2014.
- [16] Y. Labiche, "The UML Is More Than Boxes and Lines," in *Models 2008*, pp. 375–386.
- [17] Object Management Group, "OCL: Object Constraint Language," 2014.
- [18] B. Unhelkar, *Verification and Validation for Quality of UML 2.0 Models*. WILEY, 2005.
- [19] P. Skoković and M. Rakić-Skoković, "Requirements-Based Testing Process in Practice," vol. 1, no. 4, pp. 155–161, 2010.
- [20] P. Loucopoulos and V. Karakostas, *System Requirements Engineering*. McGraw-Hill Publishing Company, 1995.
- [21] S. España, A. González, and Ó. Pastor, "Communication Analysis: A Requirements Engineering Method for Information Systems," in *21st International Conference on Advanced Information Systems Engineering*, 2009, vol. 5565, pp. 530–545.
- [22] I. S. O. (ISO), *ISO Standard 9126: Software Product Quality*, vol. 2000. 2001, pp. 1–26.
- [23] I. S. O. (ISO), *ISO 9000:2000*, no. 70. 2001, pp. 1–135.

-
- [24] V. Process, V. Process, and I. Levels, *IEEE Standard for Software Verification and Validation*. 2004.
- [25] I. S. O. (ISO), *ISO Standard 9126: Software Product Quality*, vol. 2000. 2001, pp. 1–26.
- [26] M. D. Ernst, “Static and dynamic analysis: synergy and duality,” in *WODA 2003 ICSE Workshop on Dynamic Analysis*, 2003, pp. 24–27.
- [27] M. D. Ernst, “Static and dynamic analysis: synergy and duality,” in *WODA 2003 ICSE Workshop on Dynamic Analysis*, 2003, pp. 24–27.
- [28] J. M. Wing, “A Specifier’s Introduction to Formal Methods,” *IEEE Comput.*, vol. 23, no. 9, pp. 8–24, 1990.
- [29] IEEE, “IEEE Standard for Software Reviews, IEEE std 1028-1997,” 1997.
- [30] D. P. Freedman and G. M. Weinberg, *Handbook of Walkthroughs, Inspections, and Technical Reviews: Evaluating Programs, Projects, and Products*, 3rd ed. New York, USA: Dorset House Publishing Co., 2000.
- [31] A. a. Porter, J. Votta, L.G., and V. R. Basili, “Comparing Detection Methods for Software Requirements inspections: A Replicated Experiment,” *Empir. Softw. Eng.*, vol. 3, no. 4, pp. 355–379, 1998.
- [32] G. H. Travassos, F. Shull, and J. Carver, “Working with UML: A Software Design Process Based on Inspections for the Unified Modeling Language,” *Adv. Comput.*, vol. 54, pp. 35–98, 2001.
- [33] M. E. Fagan, “Design and code inspections to reduce errors in program development,” *IBM Syst. J.*, vol. 15, no. 3, pp. 182–211, 1976.
- [34] C. Rolland and C. Proix, “A Natural Language Approach for Requirements Engineering,” in *The 5th International Conference on Advanced Information Systems Engineering (CAISE’93)*, 1993, pp. 257–277.
-

- [35] J. A. Gulla, "A general explanation component for conceptual modeling in CASE environments," *ACM Trans. Inf. Syst.*, vol. 14, no. 3, pp. 297–329, 1996.
- [36] B. J., D. J., M. B., and W. E., "Making the most of formal specification through animation, testing and proof," *Sci. Comput. Program.*, vol. 29, no. 1–2, pp. 53–78, 1997.
- [37] S. J. Mellor and M. J. Balcer, *Executable UML: A Foundation for Model-Driven Architecture*. Addison Wesley, 2002.
- [38] O. J.S. and T. F. A., "Testable Requirements and Specifications," 2007, pp. 17–40.
- [39] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice*. Morgan & Claypool, 2012.
- [40] Object Management Group, "Unified Modeling Language (UML)," 2015.
- [41] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool," *Sci. Comput. Program.*, vol. 72, no. 1–2, pp. 31–39, 2008.
- [42] IEEE, "IEEE Standard Classification for Software Anomalies," 2010.
- [43] M. F. Granda, N. Condori-fernández, T. E. J. Vos, and O. Pastor, "What do we know about the Defect Types detected in Conceptual Models?," in *IEEE 9th Int. Conference on Research Challenges in Information Science (RCIS)*, 2015, pp. 96–107.
- [44] A. Van Lamsweerde, *Requirements Engineering: From System Goals to UML Models to Software Specifications*. 2009.
- [45] O. I. Lindland, G. Sindre, and A. Sølvsberg, "Understanding Quality in Conceptual Modeling," *IEEE Softw.*, vol. 11, no. 2, pp. 42–49, 1994.
- [46] C. Lange, C. M. R. V., M. J., L. J. Somers, and D. H. M., "An empirical investigation in quantifying inconsistency and incompleteness of UML designs," in *Workshop Consistency*

-
- Problems in UML-based Software Development II*, 2003, pp. 26–34.
- [47] H. J. Nelson and D. E. Monarchi, “Ensuring the quality of conceptual representations,” *Softw. Qual. J.*, vol. 15, no. 2, pp. 213–233, 2007.
- [48] F. Leung and N. Bolloju, “Analyzing the Quality of Domain Models Developed by Novice Systems Analysts,” in *Proceedings of the 38th Hawaii International Conference on System Sciences*, 2005.
- [49] S. Mijatov, P. Langer, T. Mayerhofer, and G. Kappel, “A framework for testing UML activities based on fUML,” in *MoDEVVa*, 2013, vol. 1069, pp. 1–10.
- [50] R. Conradi, P. Mohagheghi, T. Arif, L. C. Hegde, G. A. Bunde, and A. Pedersen, “Object-oriented reading techniques for inspection of UML models - An industrial experiment,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 2743, pp. 483–500, 2003.
- [51] S. Ali, T. Yue, and Z. I. Malik, “Comprehensively evaluating conformance error rates of applying aspect state machines,” in *Proceedings of the 11th annual international conference on Aspect-oriented Software Development - AOSD '12*, 2012, p. 155.
- [52] D. Dotan and A. Kirshin, “Debugging and testing behavioral UML models,” in *Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion - OOPSLA '07*, 2007, p. 838.
- [53] A. Tort and A. Olivé, “An approach to testing conceptual schemas,” *Data Knowl. Eng.*, vol. 69, no. 6, pp. 598–618, 2010.
- [54] T. Dinh-Trong, N. Kawane, S. Ghosh, and R. France, “A tool-supported approach to testing UML design models,” in *10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, 2005.
- [55] M. Gogolla, F. Büttner, and M. Richters, “USE: A UML-based
-

- specification environment for validating UML and OCL,” *Sci. Comput. Program.*, vol. 69, no. 1–3, pp. 27–34, 2007.
- [56] O. Pilskalns, A. Andrews, A. Knight, S. Ghosh, and R. France, “Testing UML designs,” *Inf. Softw. Technol.*, vol. 49, no. 8, pp. 892–912, Aug. 2007.
- [57] B. Berenbach, “The evaluation of large, complex UML analysis and design models,” in *26th International Conference on Software Engineering*, 2004, no. January 2004, pp. 232–241.
- [58] O. I. Lindland and J. Krogstie, “Validating conceptual models by transformational prototyping,” in *5th International Conference on Advanced Information Systems*, 1993, pp. 213–254.
- [59] Ö. Albayrak, “An experiment to observe the impact of UML diagrams on the effectiveness of software requirements inspections,” in *3rd International Symposium on Empirical Software Engineering and Measurement, ESEM 2009*, 2009, pp. 506–510.
- [60] B. Berenbach, “The evaluation of large, complex UML analysis and design models,” in *26th International Conference on Software Engineering*, 2004, no. January 2004, pp. 232–241.
- [61] T. Dinh-Trong, S. Ghosh, R. B. France, M. Hamilton, and B. Wilkins, “UMLAnT: an Eclipse plugin for animating and testing UML designs,” in *eclipse '05: Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, 2005, pp. 120–124.
- [62] Y. Zhang, “Test-driven modeling for model-driven development,” *IEEE Softw.*, vol. 21, no. 5, pp. 80–86, 2004.
- [63] A. Teilans, A. Kleins, Y. Merkurjev, and A. Grinbergs, “Design of UML models and their simulation using ARENA,” *WSEAS Trans. Comput. Res.*, vol. 3, no. 1, pp. 67–73, 2008.
- [64] F. Dignum, T. Kemme, W. Kreuzen, H. Weigand, and R. P. van de Riet, “Knowledge Base Modelling Based on Linguistics and Founded in Logic,” *Data Knowl. Eng.*, vol. 2, no. 3, pp. 213–254, 1987.

-
- [65] M. Gogolla, S. Conrad, R. Herzig, and N. Vlachantonis, "A Development Environment for an Object Specification Language," *Trans. Knowl. Data Eng.*, vol. 7, no. 3, pp. 505–508, 1995.
- [66] A. Tort and A. Olivé, "CSTL Processor tool, prototype for automated testing of UML/OCL conceptual schemas," 2011. [Online]. Available: <http://www.essi.upc.edu/~atort/cstlprocessor/>.
- [67] Object Management Group, "Semantics of a Foundational Subset for Executable UML Models (fUML)," 2012.
- [68] A. Queralt and E. Teniente, "Verification and validation of UML conceptual schemas with OCL constraints," in *ACM Transactions on Software Engineering and Methodology*, 2012, vol. 21, no. 2.
- [69] G. Bergmann, A. Hegedus, A. Horvath, I. Rath, Z. Ujhelyi, and D. Varro, "Implementing efficient model validation in EMF tools," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, 2011, pp. 580–583.
- [70] M. Gogolla, M. Kuhlmann, and L. Hamann, "Consistency, independence and consequences in UML and OCL models," *Lect. Notes Comput. Sci.*, vol. 5668 LNCS, pp. 90–104, 2009.
- [71] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," *Softw. Testing, Verif. Reliab.*, vol. 22, no. 5, pp. 297–312, 2012.
- [72] M. Mussa, S. Ouchani, W. Al Sammane, and A. Hamou-Lhadj, "A survey of model-driven testing techniques," in *Proceedings - International Conference on Quality Software*, 2009, pp. 167–172.
- [73] J. J. Gutiérrez, M. J. Escalona, and M. Mejías, "A Model-Driven approach for functional test case generation," *J. Syst. Softw.*, vol. 109, pp. 214–228, 2015.
- [74] C. Denger and M. M. Mora, "Test case derived from Requirement Specifications," 2003.
-

REFERENCES

- [75] R. Ibrahim, M. Z. Saringat, N. Ibrahim, and N. Ismail, "An automatic tool for generating test cases from the system's requirements," in *7th IEEE Int. Conference on Comp. and Info. Technology*, 2007, pp. 861–866.
- [76] S. Nogueira, A. Sampaio, and A. Mota, "Test generation from state based use case models," *Form. Asp. Comput.*, vol. 26, no. 3, pp. 441–490, 2014.
- [77] P. Samuel and R. Mall, "A Novel Test Case Design Technique Using Dynamic Slicing of UML Sequence Diagrams," *E-Infomatica Softw. Eng. J.*, vol. 2, no. 1, 2008.
- [78] E. Yu, "Modelling Strategic Relationships for Process Reengineering," University of Toronto, 1995.
- [79] D. Leon and A. Podgurski, "A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases," in *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*, 2003, pp. 442–453.
- [80] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test Case Prioritization: an Empirical Study," in *Proceedings of the IEEE International Conference on Software Maintenance*, 1999, p. 179.
- [81] J. Cabot, "List of Executable UML tools," 2011. [Online]. Available: <http://modeling-languages.com/list-of-executable-uml-tools/>.
- [82] Object Management Group, "Action Language for Foundational UML (ALF)," 2013.
- [83] Model Driven Solutions, "Action Language for UML (Alf) Open Source Implementation Version 0.5.1," 2011. [Online]. Available: <http://modeldriven.org/alf/>.
- [84] Project Technology, "Object Action Language Manual."
- [85] Project Technology, "Shlaer-Mellor Action Language," 1997.
- [86] I. Wilkie, A. King, M. Clarke, C. Raistrick, and P. Francis, "UML

-
- ASL Reference Guide,” 2003.
- [87] J. Cabot, “History of Executable UML – Action Language: An OMG Journey,” 2011. [Online]. Available: <http://modeling-languages.com/uml-action-language-omg-journey/>.
- [88] G. Graw and P. Herrmann, “Transformation and Verification of Executable UML Models,” *Electron. Notes Theor. Comput. Sci.*, vol. 101, pp. 3–24, 2004.
- [89] H. H. Hansen, J. Ketema, B. Luttik, M. Mousavi, J. Van de Pol, and O. Marchi dos Santos, “Automated Verification of Executable UML Models,” in *International Symposia on Formal Methods for Components and Objects*, 2010, pp. 225–250.
- [90] Y. Laurent, R. Bendraou, S. Baarir, and M.-P. Gervais, “Formalization of fUML : An Application to Process Verification,” in *International Conference on Advanced Information Systems Engineering*, 2014, pp. 347–363.
- [91] F. Xie, V. Levin, and J. C. Browne, “Model Checking for an Executable Subset of UML,” in *16th IEEE International Conference on Automated Software Engineering*, 2001.
- [92] F. Craciun, S. Motogna, and I. Lazar, “Towards Better Testing of fUML Models,” in *Sixth International Conference on Software Testing, Verification and Validation*, 2013, pp. 485–486.
- [93] Q. Lai and A. Carpenter, “Defining and Verifying Behaviour of Domain Specific Language with fUML Categories and Subject Descriptors,” in *Proceedings of the Fourth Workshop on Behaviour Modelling - Foundations and Applications*, 2012.
- [94] Z. Micskei, R. Konnerth, H. Benedek, O. Semeráth, A. Vörös, and D. Varró, “On Open Source Tools for Behavioral Modeling and Analysis with fUML and Alf,” in *1st Workshop on Open Source Software for Model Driven Engineering*, 2014, pp. 31–41.
- [95] E. Planas, J. Cabot, and C. Gómez, “Lightweight and static verification of UML executable models,” *Comput. Lang. Syst. Struct.*, vol. 46, pp. 66–90, 2016.
-

- [96] S. Guerhazi, J. Tatibouet, A. Cuccuru, S. Dhouib, S. Gérard, and E. Seidewitz, "Executable Modeling with fUML and Alf in Papyrus: Tooling and Experiments," in *1st International Workshop on Executable Modeling*, 2015, pp. 3–8.
- [97] E. Seidewitz and J. Tatibouet, "Tool Paper : Combining Alf and UML in Modeling Tools – An Example with Papyrus –," in *OCL@MoDELS*, 2015, pp. 105–119.
- [98] J. Tatibouët, A. Cuccuru, Sébastien Gérard, and F. Terrier, "Formalizing Execution Semantics of UML Profiles with fUML Models," in *International Conference on Model Driven Engineering Languages and Systems*, 2014, pp. 133–148.
- [99] T. Mayerhofer, P. Langer, and M. Wimmer, "xMOF : A Semantics Specification Language for Metamodeling," in *Satellite Events of MODELS*, 2013.
- [100] A. Tort, A. Olive, and M.-R. Sancho, "The CSTL Processor: A Tool for Automated Conceptual Schema Testing," in *30th International Conference on Conceptual Modeling*, 2011, vol. 6999, pp. 349–352.
- [101] A. Tort, A. Olivé, and M.-R. Sancho, "An approach to test-driven development of conceptual schemas," *Data Knowl. Eng.*, vol. 70, no. 12, pp. 1088–1111, 2011.
- [102] F. Weber, M. Wunram, J. Kemp, M. Pudlatz, and B. Bredehorst, "Standardisation in knowledge management – towards a common KM framework in Europe," in *Proceedings of UNICOM Seminar "Towards Common Approaches & Standards in KM,"* 2002.
- [103] S. España, "Methodological Integration of Communication Analysis into a Model-Driven Software Development Framework," Universitat Politècnica de València, 2011.
- [104] S. España, M. Ruiz, and A. González, "Systematic derivation of conceptual models from requirements models: A controlled experiment," in *Proceedings - International Conference on Research Challenges in Information Science*, 2012, no. April.

-
- [105] A. González, M. Ruiz, S. España, and Ó. Pastor, “Message structures: A modelling technique for information systems analysis and design1,” in *14th Ibero-American Conference on Software Engineering and 14th Workshop on Requirements Engineering, CIBSE 2011*, 2011, pp. 407–418.
- [106] M. Shahbaz, P. McMinn, and M. Stevenson, “Automated discovery of valid test strings from the web using dynamic regular expressions collation and natural language processing,” in *Proceedings - International Conference on Quality Software*, 2012, pp. 79–88.
- [107] N. Li, F. Li, and J. Offutt, “Better algorithms to minimize the cost of test paths,” in *Proceedings - IEEE 5th International Conference on Software Testing, Verification and Validation, ICST 2012*, 2012, pp. 280–289.
- [108] A. Andrews, R. France, S. Ghosh, and G. Craig, “Test adequacy criteria for UML design models,” *Softw. Test. Verif. Reliab.*, vol. 13, no. 2, pp. 95–127, 2003.
- [109] Object Management Group (OMG), “UML Testing Profile (UTP) Version 1.2,” 2013.
- [110] Object Management Group, “OMG Unified Modeling Language (OMG UML), SuperStructure,” 2011.
- [111] J. H. Andrews, L. C. Briand, and Y. Labiche, “Is mutation an appropriate tool for testing experiments?,” in *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, 2005, pp. 402–411.
- [112] Y. Jia and M. Harman, “Higher Order Mutation Testing,” *Inf. Softw. Technol.*, vol. 51, no. 10, pp. 1379–1393, 2009.
- [113] Y. Jia and M. Harman, “An Analysis and Survey of the Development of Mutation Testing,” *Softw. Eng. IEEE Trans.*, vol. 37, no. 5, pp. 1–31, 2011.
- [114] H. Do and G. Rothermel, “A controlled experiment assessing test case prioritization techniques via mutation faults,” in *IEEE International Conference on Software Maintenance, ICSM*, 2005,
-

- vol. 2005, pp. 411–420.
- [115] M. F. Granda, N. Condori-Fernandez, T. E. J. Vos, and Ó. Pastor, “Mutation Operators for UML Class Diagrams,” in *CAiSE 2016*, 2016.
- [116] “Mutation Operators for UML CD-based CS,” 2015. [Online]. Available: <https://staq.dsic.upv.es/webstaq/costest/FOMs.html>.
- [117] “An open-source implementation of the OMG Action Language for fUML.” [Online]. Available: <http://modeldriven.github.io/Alf-Reference-Implementation/>.
- [118] T. Massimo, F. Jouault, Z. Saidi, and J. Delatour, “Enabling OCL and fUML Integration by Transformation,” in *European Conference on Modelling Foundations and Applications*, 2016, vol. 2, pp. 156–172.
- [119] E. Seidewitz, “Model execution using the fUML Reference Implementation,” 2016. [Online]. Available: <https://github.com/ModelDriven/Alf-Reference-Implementation/wiki/Command-Line-Scripts>.
- [120] M. Ruiz, “A Model-Driven Framework to Integrate Communication Analysis and OO-Method,” Universitat Politècnica de València, 2011.
- [121] M. F. Granda, N. Condori-Fernandez, T. E. J. Vos, and O. Pastor, “Towards the automated generation of abstract test cases from requirements models,” in *1st International Workshop on Requirements Engineering and Testing*, 2014, pp. 39–46.
- [122] O. Pastor, “Diseño y Desarrollo de un Entorno de Producción Automática de Software basado en el Modelo Orientado a Objetos,” Universitat Politècnica de València., 1992.
- [123] M. Loy, R. Eckstein, D. Wood, J. Elliott, and B. Cole, *Java Swing*. O’Reilly Media, 2002.
- [124] M. F. Granda and N. Condori-fernández, “A Model-level Mutation Tool to Support the Assessment of the Test Case Quality,” in *25TH International Conference on Information*

Systems Development (ISD2016 POLAND), 2016.

- [125] F. Shull, J. Singer, and D. I. K. Sjøberg, *Guide to Advanced Empirical Software Engineering*. 2008.
- [126] G. H. Travassos, P. Sérgio, P. G. Mian, A. Cláudio, D. Neto, and J. Biolchini, "An Environment to Support Large Scale Experimentation in Software Engineering," in *13th IEEE International Conference on Engineering of Complex Computer Systems*, 2008, pp. 193–202.
- [127] N. Juristo and A. M. Moreno, *Basics of Software Engineering Experimentation*, 1st ed. Springer Publishing Company, 2010.
- [128] V. R. Basili, "The role of Experimentation in Software Engineering Past, Current, and Future," in *Proceedings of the 18th international conference on Software engineering*, 1996, pp. 442–449.
- [129] J. Cabot, R. Clarisó, E. Guerra, and J. de Lara, "Verification and validation of declarative model-to-model transformations through invariants," *J. Syst. Softw.*, vol. 83, no. 2, pp. 283–302, 2010.
- [130] M. Gogolla and A. Vallecillo, "Tractable model transformation testing," *Lect. Notes Comput. Sci.*, vol. 6698 LNCS, pp. 221–235, 2011.
- [131] S. España, A. González, Ó. Pastor, and M. Ruiz, "Technical Report Communication Analysis and the OO-Method : Manual Derivation of the Conceptual Model the SuperStationery Co. Lab Demo," Valencia, 2011.
- [132] S. España, A. González, Ó. Pastor, and M. Ruiz, "Integration of Communication Analysis and the OO-Method: Rules for the manual derivation of the Conceptual Model," Valencia, 2011.
- [133] A. Tort and A. Olivé, "Case Study: Conceptual Modeling of Basic Sudoku," 2006. [Online]. Available: <http://guifre.lsi.upc.edu/Sudoku.pdf>.
- [134] A. Tort, "A Basic Set of Test Cases for a Fragment of the

- osCommerce Conceptual Schema,” *UPC*, 2009. [Online]. Available: <http://hdl.handle.net/2117/6130>.
- [135] E. Planas and A. Olivé, “The DBLP Case Study,” 2006. [Online]. Available: <http://guifre.lsi.upc.edu/DBLP.pdf>.
- [136] R. Van de Stadt, “CyberChair.” [Online]. Available: <http://www.borbala.com/cyberchair/>.
- [137] H. Ehrig and C. Ermel, “Semantical correctness and completeness of model transformations using graph and rule transformation,” in *Lecture Notes in Computer Science*, 2008, vol. 5214 LNCS, pp. 194–210.
- [138] A. Kleppe, J. Warmer, and W. Bast, *MDA Explained, The Model-Driven Architecture: Practice and Promise*. Addison Wesley, 2003.
- [139] T. Yue and S. Ali, “A MOF-based framework for defining metrics to measure the quality of models,” in *Lecture Notes in Computer Science*, vol. 8569 LNCS, 2014, pp. 213–229.
- [140] V. R. Basili, G. Caldiera, and H. D. Rombach, “Goal Question Metric Paradigm,” *Encyclopedia of Software Engineering*. 1994.
- [141] U. Rueda, S. España, and M. Ruiz, “GREAT Process Modeller user manual,” 2015.
- [142] C. Wholin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*, vol. 1. 2012.
- [143] R. van Solingen and E. Berghout, *The Goal/Question/Metric Method – A Practical Guide for Quality Improvement of Software Development*. McGraw-Hill, 1999.
- [144] J. A. Morgan, G. J. Knafl, and W. E. Wong, “Predicting fault detection effectiveness,” in *Proceedings Fourth International Software Metrics Symposium*, 1997, pp. 82–89.
- [145] G. Charness, U. Gneezy, and M. A. Kuhn, “Experimental methods: Between-subject and within-subject design,” *J. Econ.*

-
- Behav. Organ.*, vol. 81, no. 1, pp. 1–8, 2012.
- [146] T. E. J. Vos, B. Marin, M. J. Escalona, and A. Marchetto, “A Methodological Framework for Evaluating Software Testing Techniques and Tools,” *2012 12th Int. Conf. Qual. Softw.*, pp. 230–239, 2012.
- [147] C. Jing, Z. Wang, X. Shi, X. Yin, and J. Wu, “Mutation Testing of Protocol Messages Based on Extended TTCN-3,” in *22nd International Conference on Advanced Information Networking and Applications*, 2008, pp. 667–674.
- [148] S. Jamieson, “Likert scales : how to (ab) use them,” *Med. Educ.*, vol. 38, pp. 1217–1218, 2004.
- [149] Fred Davis, “Perceived Usefulness , Perceived Ease Of Use , And User Acceptance of Information Technology,” *MIS Q.*, vol. 13, no. 3, pp. 319–340, 1989.
- [150] J. Wu, Y. Chen, and L. Lin, “Empirical evaluation of the revised end user computing acceptance model,” *Comput. Human Behav.*, vol. 23, pp. 162–174, 2007.
- [151] J. Sauro and J. R. Lewis, *Quantifying the User Experience: Practical Statistics for User Research*. 2012.
- [152] P. Mohagheghi, W. Gilani, A. Stefanescu, and M. A. Fernández, “An Empirical Study of the State of the Practice and Acceptance of Model-Driven Engineering in four Industrial Cases,” *Empir. Softw. Eng.*, vol. 18, no. 1, pp. 89–116, 2013.
- [153] Eclipse, “Eclipse IDE for Java Developers.” [Online]. Available: www.eclipse.org.
- [154] JUnit, “JUnit framework.” [Online]. Available: junit.org.
- [155] E. Planas, J. Cabot, and C. Gómez, “Lightweight verification of executable models,” in *Conceptual Modelling -ER2011*, 2011, pp. 467–475.
-

APPENDICES

Appendix A

Mutation Operators for UML CD-based Conceptual Schemas

This Appendix contains the mutation operators used to mutate the UML CD-based conceptual schemas during the process of prioritization (Chapter 5) and evaluation of effectiveness of CoSTest test cases (Chapter 8). The process to define these mutation operators is described in [115].

Table A.1. Mutation Operators defined for a UML CD-based CS taken from [115]

#	Code	Mutation Operator rule and relation with other mutation operators
1	UCO1	Adds a redundant constraint to the CD
2	UCO2	Adds an extraneous constraint to the CD
3	UAS1	Adds a redundant association to the CD
4	UAS2	Adds a redundant derived association to the CD. Relation: UCO2
5	UAS3	Adds an extraneous association to the CD
6	UAS4	Adds an extraneous derived association to the CD. Relation: UCO2
7	UGE1	Adds a redundant generalization to the CD
8	UGE2	Adds an extraneous generalization to the CD
9	UCL1	Adds a redundant class to the CD
10	UCL2	Adds an extraneous class to the CD
11	UCL3	Adds a redundant association class to the CD
12	UCL4	Adds an extraneous association class to the CD
13	UAT1	Adds a redundant attribute to a Class
14	UAT2	Adds an extraneous attribute to a Class
15	UOP1	Adds a redundant operation to a Class
16	UOP2	Adds an extraneous operation to a Class
17	UPA1	Adds a redundant parameter to an Operation
18	UPA2*	Adds an extraneous Parameter to an Operation
19	WCO1*	Changes the constraint by deleting the references to a class Attribute
20	WCO2**	Changes the Attribute data type in the constraint. Relation: WPA, WAT3
21	WCO3*	Change the constraint by deleting the calls to specific operation.
22	WCO4*	Changes an arithmetic operator for another and supports binary operators: +, -, *, /
23	WCO5*	Changes the constraint by adding the conditional operator "not"
24	WCO6*	Changes a conditional operator for another and supports operators: or, and
25	WCO7*	Changes the constraint by deleting the conditional operator "not"
26	WCO8*	Changes a relational operator for another operators: <, <=, >, >=, =, !=
27	WCO9*	Changes a constraint by deleting a unary arithmetic operator (-).
28	WAS1*	Interchange the members (memberEnd) of an Association.
29	WAS2*	Changes the association type (i.e. normal, composite).
30	WAS3*	Changes the memberEnd multiplicity of an Association (i.e. *-*, 0..1-0..1, *-0..1)
31	WGE**	Changes the Generalization member ends. Relation: MPA, UPA
32	WCL1*	Changes visibility kind of the Class (i.e. private)

APPENDIX A

33	WCL2	Changes Class by an Association Class
34	WCL3	Changes Association Class for a Class
35	WCL4	Changes the Class feature "isAbstract " to true.
36	WAT1**	Changes the Attribute feature "Is Derived" to true. Relation: UCO2
37	WAT2**	Changes the Attribute property "Is Derived" to false. Relation: MCO
38	WAT3**	Changes the Attribute data type. Relation: WPA, WCO2
39	WAT4	Changes the Attribute visibility property
40	WOP1	Changes the order of the parameters
41	WOP2*	Changes the visibility kind of an operation. Restriction. WOP2 has to be applied to operations that are not related with any constraints. Relation: MCO
42	WOP3	Changes the data type returned by operation. Relation: WAT3
43	WPA*	Changes the Parameter data type (i.e. String, Integer, Boolean, Date, Real). Restriction. WPA has to be applied to parameters that are not related with attributes in a constructor operation. To reduce mutants only a change is counted.
44	MCO*	Deletes a constraint (i.e. pre-condition, post-condition constraint, body constraint)
45	MAS*	Deletes an Association. Restriction. MAS has to be applied to associations that are not related with any constraints. Relation: MCO
46	MGE**	Deletes a Generalization relation. Relation: MPA, UPA
47	MCL**	Deletes the class (i.e. normal or association class). Relation: MCO, MAT, MOP, MGE.
48	MAT**	Deletes an Attribute. Relation: MPA, MCO
49	MOP**	Deletes the operation. Relation: MPA, MCO, WCO3
50	MPA*	Deletes a Parameter from an Operation. Restriction. This mutation operator has to be applied to operations without related constraints. Relation: MCO

Appendix B

Case Study: The Incident Management System

This appendix describes how we applied our CoSTest validation framework using the Incident Management case study, which was carried out in the context of the everis company. Within this private entity, we put into practice the validation framework that is presented in Chapter 5: we used CoSTest to generate the test cases, generate mutants from the conceptual schema that represents the system and also to execute the test cases against the mutants and evaluate the results.

Overall, the application of CoSTest in an industrial context was successful and showed the effectiveness of the model-driven validation framework described in this thesis.

The remainder of the appendix is structured as follows: Sections B.1-B.6 show the application of the phases that comprise our validation framework (i.e. design, generation, prioritization, execution and evaluation) and Section B.7 outlines some conclusions from the case study.

B.1 Test Analysis

This Section introduces the requirements of the Incident Management (IM) System using the Communicational Analysis instruments (i.e. the event description templates and the event diagram).

B.1.1 Event Description Templates

The event description templates for the communicative events using España et al's notation [21] are described below:

TECH1. Technician Registration

Description

The technician is described and registered in the system. The PMO has a technician management tool to record and keep track of all the technicians.

Contact Requirements

- **Primary actor:** Technician
- **Communication channel:** Face to face
- **Temporal restrictions:** none
- **Frequency:** none

Communicational content requirements

- **Support Actor:** technician

Table B.1. Communication Structure for TECH1

FIELD	OP	DOMAIN	EXTENDS BUSINESS OBJECT
TECHNICIAN = < id Technician + Name >	g i	text text	False False

USR1. User Registration

Description

The user is described and registered in the system. The PMO has a user management tool to record and keep track of all the users.

Contact Requirements

- **Primary actor:** User
- **Communication channel:** Face to face
- **Temporal restrictions:** none
- **Frequency:** none

Communicational content requirements

- **Support Actor:** technician

Table B.2. Communication Structure for USR1

FIELD	OP	DOMAIN	EXTENDS BUSINESS OBJECT
USER = < id User + Name >	g i	text text	False False

PLAN1. Plan RegistrationDescription

A set of steps for the incident resolution are registered in the system as a resolution plan. The PMO has a plan management tool to record and keep track of all the resolution plans.

Contact Requirements

- **Primary actor:** Technician
- **Communication channel:** Face to face
- **Temporal restrictions:** none
- **Frequency:** none

Communicational content requirements

- **Support Actor:** Technician

Table B.3. Communication Structure for PLAN1

FIELD	OP	DOMAIN	EXTENDS BUSINESS OBJECT
RESOLUTION PLAN = < id Plan + Name+ Step sequence >	g i i	number text text	False False False

INC1. Incident RegistrationDescription

The incident is described and registered in the system. The PMO has an incident management tool to record and keep track of all the incidents.

Contact Requirements

- **Primary actor:** User
- **Communication channel:** phone, face to face
- **Temporal restrictions:** none

- **Frequency:** none

Communicational content requirements

- **Support Actor:** PMO Responsible

Table B.4. Communication Structure for INC1

FIELD	OP	DOMAIN	EXTENDS BUSINESS OBJECT
INCIDENT =			
< id Incident +	g	text	False
Request type +	i	text [incident] request]	False
Component +	i	Text	False
User +	i	User	False
Contact information +	i	Text	False
Initial Scope	i	Text	False
Subject +	i	Text	False
Description +	i	Text	False
Step sequence +	i	Text	False
>			

INC2. Incident Priority Assignment

Description

After that, an initial analysis of the incident is done by the PMO in order to find risks and additional information, and a priority is assigned.

Contact Requirements

- **Primary actor:** Phone operator
- **Communication channel:** Incident management tool
- **Temporal restrictions:** Phone Operator
- **Frequency:** none

Communicational content requirements

- **Support Actor:** PMO Technician

Table B.5. Communication Structure for INC2

FIELD	OP	DOMAIN	EXTENDS BUSINESS OBJECT
INCIDENT PRIORITY= < State + Progress + Initial Scope + Incident+ >	i i i i	Text Text Text Incident	False False False True

INC3. Register Scope

Description

The incident is analysed and the work is described. The incident scope is calculated taking into account the incident details and PMO background.

Contact Requirements

- **Primary actor:** PMO Technician
- **Communication channel:** Incident management tool
- **Temporal restrictions:** none
- **Frequency:** none

Communicational content requirements

- **Support Actor:** PMO Technician

Table B.6. Communication Structure for INC3

FIELD	OP	DOMAIN	EXTENDS BUSINESS OBJECT
INITIAL SCOPE = < State + Progress + Estimated Scope + Incident >	i i i i	Text Text Text Incident	False False False True

INC4. Assess PMO Capacity to Solve

Description

After that, the incident estimation is calculated based on the PMO experience and depending on human non-calculated estimation. Then the incident is reassigned to the PMO, to the

municipality or to an external company. The reassignment depends on the PMO’s capability of solving it and the incident scope.

Contact Requirements

- **Primary actor:** PMO Technician
- **Communication channel:** Face to face
- **Temporal restrictions:** none
- **Frequency:** none

Communicational content requirements

- **Support Actor:** PMO Technician

Table B.7. Communication Structure for INC4

FIELD	OP	DOMAIN	EXTENDS BUSINESS OBJECT
EQUIPMENT AVAILABILITY =			
< Incident +	i	Incident	True
State+	i	Text	False
Progress	i	Text	False
>			

INC5. Resource Allocation

Description

A technician for the incident is assigned.

Contact Requirements

- **Primary actor:** PMO Responsible
- **Communication channel:** incident management tool
- **Temporal restrictions:** none
- **Frequency:** none

Communicational content requirements

- **Support Actor:** PMO Responsible

Table B.8. Communication Structure for INC5

FIELD	OP	DOMAIN	EXTENDS BUSINESS OBJECT
RESOURCE ALLOCATION= < id allocation+ Incident + Technician >	g i i	number Incident Technician	False False False

INC6. Check PlanDescription

A plan for the incident type is checked.

Contact Requirements

- **Primary actor:** PMO Technician
- **Communication channel:** Face to face, incident management tool
- **Temporal restrictions:** none
- **Frequency:** none

Communicational content requirements

- **Support Actor:** PMO Responsible

Table B.9. Communication Structure for INC6

FIELD	OP	DOMAIN	EXTENDS BUSINESS OBJECT
CHECK PLAN= < Incident + Plan Response >	i i	Incident Text	True False

INC7. PMO Incident ResolutionDescription

If the plan already exists, the resolution proceeds following the described steps. If there is no plan defined for the given incident type, no further actions are carried out.

Contact Requirements

- **Primary actor:** PMO Technician
- **Communication channel:** Face to face, incident management tool, phone

- **Temporal restrictions:** none
- **Frequency:** none

Communicational content requirements

- **Support Actor:** PMO Technician

Table B.10. Communication Structure for INC7

FIELD	OP	DOMAIN	EXTENDS BUSINESS OBJECT
PMO RESOLUTION = < Incident + Step sequence+ State+ Progress >	i i i i	Incident Text Text Text	True False False False

INC8. PMO Resolution Validation

Description

The incident is checked to determine whether or not it has been solved. If the incident solution solves the incident the incident is solved and is updated as “Solved”. Else further actions are required and the incident is updated as “Reallocation pending”.

Contact Requirements

- **Primary actor:** PMO Technician
- **Communication channel:** Incident management tool
- **Temporal restrictions:** none
- **Frequency:** none

Communicational content requirements

- **Support Actor:** PMO Technician

Table B.11. Communication Structure for INC8

FIELD	OP	DOMAIN	EXTENDS BUSINESS OBJECT
PMO RESOLUTION VALIDATION= < Incident + State+ Progress >	i i i	Incident Text Text	True False False

INC9. Municipality Assignment Evaluation

Description

Then the incident is reassigned to the municipality. The reassignment depends on the PMO’s capability of solving it and the incident scope.

Contact Requirements

- **Primary actor:** PMO Technician
- **Communication channel:** Face to face, Incident management tool
- **Temporal restrictions:** none
- **Frequency:** none

Communicational content requirements

- **Support Actor:** PMO Technician

Table B.12. Communication Structure for INC9

FIELD	OP	DOMAIN	EXTENDS BUSINESS OBJECT
ASSIGNMENT TO MUNICIPALITY = < State + Progress + Incident+ >	i i i	Text Text Incident	False False True

INC10. Municipality Resolution

Description

If the PMO has not enough capacity to solve the incident and the responsibility belongs to the municipality, then the incident will be assigned to it. In this case the municipality will solve the incident.

Contact Requirements

- **Primary actor:** Municipality Responsible
- **Communication channel:** Incident management tool
- **Temporal restrictions:** none
- **Frequency:** none

Communicational content requirements

- **Support Actor:** PMO Responsible

Table B.13. Communication Structure for INC10

FIELD	OP	DOMAIN	EXTENDS BUSINESS OBJECT
MUNICIPALITY RESOLUTION= < Incident + Step Sequence+ State + Progress >	i i i i	Incident Text Text Text	True False False False

INC11. Municipality Resolution Validation

Description

The incident is checked to know if it is solved or not. If the incident solution solves the incident the incident is solved and it is updated as “Solved”. Else further actions are required and the incident is updated as “Reallocation pending”. It is necessary to validate the incident solution depending on the legal framework and quality standards.

Contact Requirements

- **Primary actor:** PMO Technician
- **Communication channel:** Incident management tool
- **Temporal restrictions:** none
- **Frequency:** none

Communicational content requirements

- **Support Actor:** PMO Responsible

Table B.14. Communication Structure for INC11

FIELD	OP	DOMAIN	EXTENDS BUSINESS OBJECT
MUNICIPALITY RESOLUTION VALIDATION= < Incident + State+ Progress >	i i i	Incident Text text	True False False

INC12. Company Assignment Evaluation

Description

In case the incidents are higher than the PMO capacity + Municipality capacity, then it will reallocate to the external company.

Contact Requirements

- **Primary actor:** PMO Responsible
- **Communication channel:** Incident management tool
- **Temporal restrictions:** none
- **Frequency:** none

Communicational content requirements

- **Support Actor:** PMO Responsible

Table B.15. Communication Structure for INC12

FIELD	OP	DOMAIN	EXTENDS BUSINESS OBJECT
ASSIGNMENT TO COMPANY = < State + Progress + Incident+ >	i i i	Text Text Incident	False False True

INC13. Incident and Impact External Company Analysis

Description

The company will provide an impact report with the incident analysis, implications, possible solutions and time estimation.

Contact Requirements

- **Primary actor:** PMO Technician
- **Communication channel:** Incident management tool
- **Temporal restrictions:** none
- **Frequency:** none

Communicational content requirements

- **Support Actor:** PMO Responsible

Table B.16. Communication Structure for INC13

FIELD	OP	DOMAIN	EXTENDS BUSINESS OBJECT
EXTERNAL COMPANY ANALYSIS= < Incident +	i	Incident	False
Subject+	i	Text	False
Description+	i	Text	False
Analysis+	i	Text	False
Implications+	i	Text	False
Possible Solutions+	i	Text	False
Time estimation	i	Text	False
>			

INC14. Action Plan DefinitionDescription

The PMO responsible analyses the impact report with the technicians. If the incident is a bug, the external company will fix it. If it becomes an improvement, the decision to carry it out will be taken in the next steps.

Contact Requirements

- **Primary actor:** PMO Responsible
- **Communication channel:** Incident management tool
- **Temporal restrictions:** none
- **Frequency:** none

Communicational content requirements

- **Support Actor:** PMO Responsible

Table B.17. Communication Structure for INC14

FIELD	OP	DOMAIN	EXTENDS BUSINESS OBJECT
PLAN DEFINITION = < Incident +	i	Incident	True
State +	i	Text	False
Progress >	i	Text	False

INC15. Improvement EvaluationDescription

The PMO's director and Project leader are involved in the deciding which option should be used to solve the incident and if a deeper analysis is needed.

Contact Requirements

- **Primary actor:** PMO Responsible
- **Communication channel:** Incident management tool
- **Temporal restrictions:** none
- **Frequency:** none

Communicational content requirements

- **Support Actor:** PMO Responsible

Table B.18. Communication Structure for INC15

FIELD	OP	DOMAIN	EXTENDS BUSINESS OBJECT
IMPROVEMENT EVALUATION= < Incident + State + Progress >	i i i	Incident Text Text	True False False

INC16. Impact AnalysisDescription

If the chosen option is a new development, the external company will make the functional and technical designs, and estimation in time and cost.

Contact Requirements

- **Primary actor:** PMO Responsible
- **Communication channel:** Incident management tool
- **Temporal restrictions:** none
- **Frequency:** none

Communicational content requirements

- **Support Actor:** PMO Responsible

Table B.19. Communication Structure for INC16

FIELD	OP	DOMAIN	EXTENDS BUSINESS OBJECT
IMPACT ANALYSIS= < Incident + State+ Progress + Time Estimation + Cost Estimation >	i i i i i	Incident Text Text Text Text	True False False False False

INC17. Functional and Technical Design Documents Revision

Description

The responsible PMO and technicians revise the documents provided by the external company to check if the requirements are well specified.

Contact Requirements

- **Primary actor:** Company Responsible
- **Communication channel:** Incident management tool
- **Temporal restrictions:** none
- **Frequency:** none

Communicational content requirements

- **Support Actor:** PMO Responsible

Table B.20. Communication Structure for INC17

FIELD	OP	DOMAIN	EXTENDS BUSINESS OBJECT
DOCUMENTS REVISION= < Incident + State+ Progress >	i i i	Incident Text Text	True False False

INC18. Need and Viability Evaluation

Description

The PMO responsible shows the chosen option to the PMO’s Project Leader and Director. Then it is decided if it is approved or not.

Contact Requirements

- **Primary actor:** PMO Director
- **Communication channel:** Incident management tool
- **Temporal restrictions:** none
- **Frequency:** none

Communicational content requirements

- **Support Actor:** PMO Responsible

Table B.21. Communication Structure for INC18

FIELD	OP	DOMAIN	EXTENDS BUSINESS OBJECT
VIABILITY EVALUATION=			
< Incident +	i	Incident	True
State+	i	Text	False
Progress	i	Text	False
>			

INC19. Company Incident Resolution

Description

Once the improvement or the new development is approved, the external Company proceeds with the development.

Contact Requirements

- **Primary actor:** Company Responsible
- **Communication channel:** Incident management tool
- **Temporal restrictions:** none
- **Frequency:** none

Communicational content requirements

- **Support Actor:** PMO Technician

Table B.22. Communication Structure for INC19

FIELD	OP	DOMAIN	EXTENDS BUSINESS OBJECT
COMPANY RESOLUTION =			
< Incident +	i	Incident	True
Step Sequence+	i	Text	False
State+	i	Text	False
Progress	i	Text	False
>			

INC20. Company Resolution Validation

Description

It is necessary to validate the incident solution depending on the legal framework and quality standards.

Contact Requirements

- **Primary actor:** PMO technician
- **Communication channel:** Incident management tool
- **Temporal restrictions:** none

- **Frequency:** none

Communicational content requirements

- **Support Actor:** PMO Technician

Table B.23. Communication Structure for INC20

FIELD	OP	DOMAIN	EXTENDS BUSINESS OBJECT
COMPANY RESOLUTION VALIDATION= < Incident + State+ Progress >	i i i	Incident Text Text	True False False

INC21. Incidence Closure

The incident is marked as “Closed” either if it is marked as “solved” or “Implementation not allowed”.

Contact Requirements

- **Primary actor:** PMO Technician
- **Communication channel:** Incident management tool
- **Temporal restrictions:** none
- **Frequency:** none

Communicational content requirements

- **Support Actor:** PMO Technician

Table B.24. Communication Structure for INC21

FIELD	OP	DOMAIN	EXTENDS BUSINESS OBJECT
INCIDENT CLOSURE= < Incident + State >	i i	Incident Text	True False

Each Event Specification Template has a **Message Structure** in the GREAT tool modeller [141] to define the information that is communicated in the event. Figure B.1 shows a partial view of the message structure for the last communicative event “INC20. Company Resolution Validation”

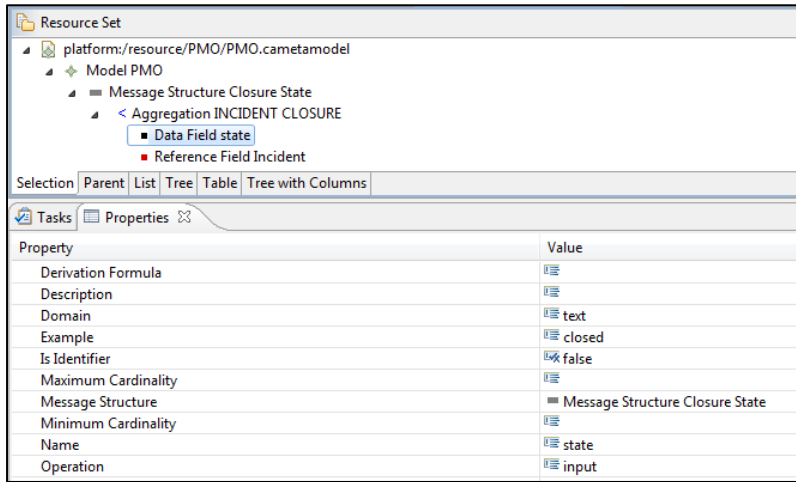


Figure B.1. Partial view of the message structure in the GREAT tool [141]

B.1.2 Events Diagram

Figure B.2 presents part of the communicative event diagram (CED) of the Incident Management (IM) of the PMO business process.

B.2 Test Design

This section describes how we applied our CoSTest validation framework to generate the Test Model and the Test Scenario Model using the Incident Management case study.

We divide this section into three main subsections (B.2.1, B.2.2 and B.2.3), each of which focuses on a specific phase of the framework.

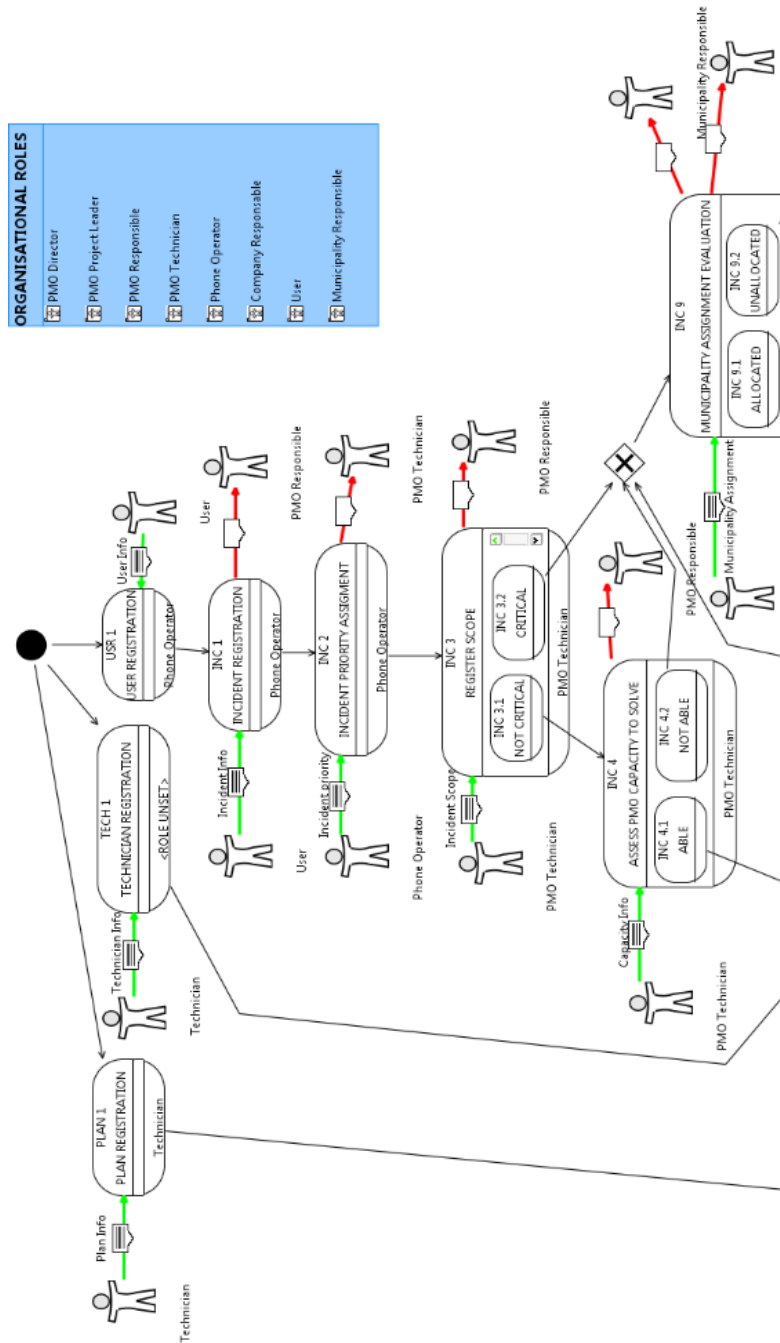


Figure B.2. Event Diagram using Communication Analysis

B.2.1 Test Model

This phase of CoSTest involves a model-to-model transformation that is carried out according to the model-driven strategy implemented in CoSTest (see Section 6.3.1). Figure B.3 shows the test model for IM case study.

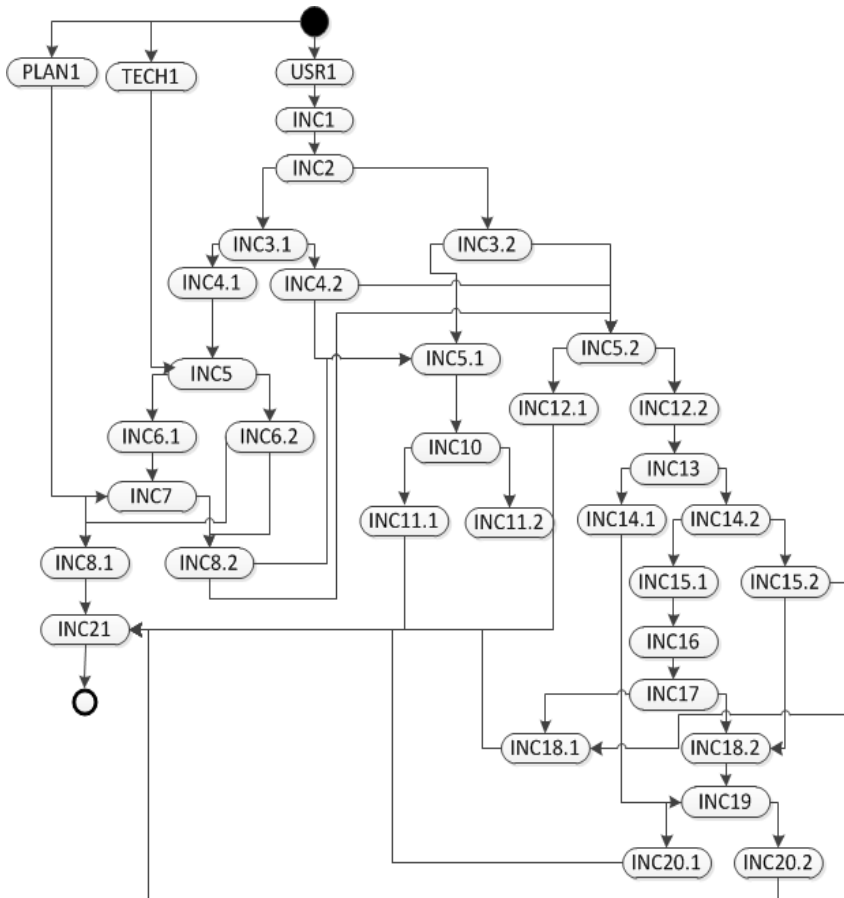


Figure B.3. Test Model for IM case study

B.2.2 Test Scenario Model

The model-driven generation for the test scenario model is implemented in CoSTest using a classic pathfinder or graph traversal algorithm to traverse from parent root to child node (see Section 6.3.2). The test scenarios are summarized in the following list.

APPENDIX B

1. PLAN_REGISTRATION:TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:EXISTS_PLAN:PMO_INCIDENT_RESOLUTION:UNSOLVED_BY_PMO:ALLOCATED_IN_MUNICIPALITY:MUNICIPALITY_INcident_RESOLUTION:UNSOLVED_BY_MUNICIPALITY:ASSIGNED_TO_COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:IMPROVEMENT:NEW_DEVELOPMENT:IMPACT_ANALYSIS:FUNCTIONAL_AND_TECHNICAL_DESIGN_DOCUMENTS_REVISION:APPROVAL:COMPANY_INCIDENT_RESOLUTION:SOLVED_BY_COMPANY:INCIDENCE_CLOSURE
2. USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:CRITICAL:UNALLOCATED_IN_MUNICIPALITY:NOT_ASSIGNED_TO_COMPANY:INCIDENCE_CLOSURE
3. USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_NOT_ABLE:UNALLOCATED_IN_MUNICIPALITY:ASSIGNED_TO_COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:IMPROVEMENT:NEW_DEVELOPMENT:IMPACT_ANALYSIS:FUNCTIONAL_AND_TECHNICAL_DESIGN_DOCUMENTS_REVISION:APPROVAL:COMPANY_INCIDENT_RESOLUTION:SOLVED_BY_COMPANY:INCIDENCE_CLOSURE
4. TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:NOT_EXISTS:UNSOLVED_BY_PMO:ALLOCATED_IN_MUNICIPALITY:MUNICIPALITY_INCIDENT_RESOLUTION:UNSOLVED_BY_MUNICIPALITY:ASSIGNED_TO_COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:IMPROVEMENT:NEW_DEVELOPMENT:IMPACT_ANALYSIS:FUNCTIONAL_AND_TECHNICAL_DESIGN_DOCUMENTS_REVISION:APPROVAL:COMPANY_INCIDENT_RESOLUTION:SOLVED_BY_COMPANY:INCIDENCE_CLOSURE
5. PLAN_REGISTRATION:TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:EXISTS_PLAN:PMO_INCIDENT_RESOLUTION:SOLVED_BY_PMO:INCIDENCE_CLOSURE
6. PLAN_REGISTRATION:TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:EXISTS_PLAN:PMO_INCIDENT_RESOLUTION:UNSOLVED_BY_PMO:UNALLOCATED_IN_MUNICIPALITY:ASSIGNED_TO_COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:IMPROVEMENT:NEW_DEVELOPMENT:IMPACT_ANALYSIS:FUNCTIONAL_AND_TECHNICAL_DESIGN_DOCUMENTS_REVISION:APPROVAL:COMPANY_INCIDENT_RESOLUTION:SOLVED_BY_COMPANY:INCIDENCE_CLOSURE
7. PLAN_REGISTRATION:TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:EXISTS_PLAN:PMO_INCIDENT_RESOLUTION:UNSOLVED_BY_PMO:ALLOCATED_IN_MUNICIPALITY:MUNICIPALITY_INcident_RESOLUTION:SOLVED_BY_MUNICIPALITY:INCIDENCE_CLOSURE
8. PLAN_REGISTRATION:TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:EXISTS_PLAN:PMO_INCIDENT_RESOLUTION:UNSOLVED_BY_PMO:ALLOCATED_IN_MUNICIPALITY:MUNICIPALITY_INcident_RESOLUTION:UNSOLVED_BY_MUNICIPALITY:NOT_ASSIGNED_TO_COMPANY:INCIDENCE_CLOSURE
9. PLAN_REGISTRATION:TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:EXISTS_PLAN:PMO_INCIDENT_RESOLUTION:UNSOLVED_BY_PMO:ALLOCATED_IN_MUNICIPALITY:MUNICIPALITY_INcident_RESOLUTION:UNSOLVED_BY_MUNICIPALITY:ASSIGNED_TO_COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:INCIDENT:COMPANY_INCIDENT_RESOLUTION:SOLVED_BY_COMPANY:INCIDENCE_CLOSURE

-
10. PLAN_REGISTRATION:TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:EXISTS_PLAN:PMO_INCIDENT_RESOLUTION:UNSOLVED_BY_PMO:ALLOCATED_IN_MUNICIPALITY:MUNICIPALITY_INcident_RESOLUTION:UNSOLVED_BY_MUNICIPALITY:ASSIGNED_TO_COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:IMPROVEMENT:IMPROVE:APPROVAL:COMPANY_INCIDENT_RESOLUTION:SOLVED_BY_COMPANY:INCIDENCE_CLOSURE
 11. PLAN_REGISTRATION:TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:EXISTS_PLAN:PMO_INCIDENT_RESOLUTION:UNSOLVED_BY_PMO:ALLOCATED_IN_MUNICIPALITY:MUNICIPALITY_INcident_RESOLUTION:UNSOLVED_BY_MUNICIPALITY:ASSIGNED_TO_COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:IMPROVEMENT:NEW_DEVELOPMENT:IMPACT_ANALYSIS:FUNCTIONAL_AND_TECHNICAL_DESIGN_DOCUMENTS_REVISION:NOT_ALLOWED:INCIDENCE_CLOSURE
 12. PLAN_REGISTRATION:TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:EXISTS_PLAN:PMO_INCIDENT_RESOLUTION:UNSOLVED_BY_PMO:ALLOCATED_IN_MUNICIPALITY:MUNICIPALITY_INcident_RESOLUTION:UNSOLVED_BY_MUNICIPALITY:ASSIGNED_TO_COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:IMPROVEMENT:NEW_DEVELOPMENT:IMPACT_ANALYSIS:FUNCTIONAL_AND_TECHNICAL_DESIGN_DOCUMENTS_REVISION:APPROVAL:COMPANY_INCIDENT_RESOLUTION:UNSOLVED_BY_COMPANY:INCIDENCE_CLOSURE
 13. USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_NOT_ABLE:UNALLOCATED_IN_MUNICIPALITY:NOT_ASSIGNED_TO_COMPANY:INCIDENCE_CLOSURE
 14. USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_NOT_ABLE:UNALLOCATED_IN_MUNICIPALITY:ASSIGNED_TO_COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:INCIDENT:COMPANY_INCIDENT_RESOLUTION:SOLVED_BY_COMPANY:INCIDENCE_CLOSURE
 15. USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_NOT_ABLE:UNALLOCATED_IN_MUNICIPALITY:ASSIGNED_TO_COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:IMPROVEMENT:IMPROVE:APPROVAL:COMPANY_INCIDENT_RESOLUTION:SOLVED_BY_COMPANY:INCIDENCE_CLOSURE
 16. USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_NOT_ABLE:UNALLOCATED_IN_MUNICIPALITY:ASSIGNED_TO_COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:IMPROVEMENT:NEW_DEVELOPMENT:IMPACT_ANALYSIS:FUNCTIONAL_AND_TECHNICAL_DESIGN_DOCUMENTS_REVISION:NOT_ALLOWED:INCIDENCE_CLOSURE
 17. USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_NOT_ABLE:UNALLOCATED_IN_MUNICIPALITY:ASSIGNED_TO_COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:IMPROVEMENT:NEW_DEVELOPMENT:IMPACT_ANALYSIS:FUNCTIONAL_AND_TECHNICAL_DESIGN_DOCUMENTS_REVISION:APPROVAL:COMPANY_INCIDENT_RESOLUTION:UNSOLVED_BY_COMPANY:INCIDENCE_CLOSURE
 18. TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:NOT_EXISTS:SOLVED_BY_PMO:INCIDENCE_CLOSURE
 19. TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:NOT_EXISTS:UNSOLVED_BY_PMO:UNALLOCATED_IN_MUNICIPALITY:ASSIGNED_TO_COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY
-

APPENDIX B

- ANALYSIS:IMPROVEMENT:NEW_DEVELOPMENT:IMPACT_ANALYSIS:FUNCTIONAL AND TECHNICAL DESIGN DOCUMENTS REVISION:APPROVAL:COMPANY_INCIDENT_RESOLUTION:SOLVED_BY_COMPANY:INCIDENCE_CLOSURE
20. TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:NOT_EXISTS:UNSOLVED_BY_PMO:ALLOCATED_IN_MUNICIPALITY:MUNICIPALITY_INCIDENT_RESOLUTION:SOLVED_BY_MUNICIPALITY:INCIDENCE_CLOSURE
 21. TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:NOT_EXISTS:UNSOLVED_BY_PMO:ALLOCATED_IN_MUNICIPALITY:MUNICIPALITY_INCIDENT_RESOLUTION:UNSOLVED_BY_MUNICIPALITY:NOT_ASSIGNED_TO_COMPANY:INCIDENCE_CLOSURE
 22. TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:NOT_EXISTS:UNSOLVED_BY_PMO:ALLOCATED_IN_MUNICIPALITY:MUNICIPALITY_INCIDENT_RESOLUTION:UNSOLVED_BY_MUNICIPALITY:ASSIGNED_TO_COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:INCIDENT:COMPANY_INCIDENT_RESOLUTION:SOLVED_BY_COMPANY:INCIDENCE_CLOSURE
 23. TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:NOT_EXISTS:UNSOLVED_BY_PMO:ALLOCATED_IN_MUNICIPALITY:MUNICIPALITY_INCIDENT_RESOLUTION:UNSOLVED_BY_MUNICIPALITY:ASSIGNED_TO_COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:IMPROVEMENT:IMPROVE:APPROVAL:COMPANY_INCIDENT_RESOLUTION:SOLVED_BY_COMPANY:INCIDENCE_CLOSURE
 24. TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:NOT_EXISTS:UNSOLVED_BY_PMO:ALLOCATED_IN_MUNICIPALITY:MUNICIPALITY_INCIDENT_RESOLUTION:UNSOLVED_BY_MUNICIPALITY:ASSIGNED_TO_COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:IMPROVEMENT:NEW_DEVELOPMENT:IMPACT_ANALYSIS:FUNCTIONAL_AND_TECHNICAL_DESIGN_DOCUMENTS_REVISION:NOT_ALLOWED:INCIDENCE_CLOSURE
 25. TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:NOT_EXISTS:UNSOLVED_BY_PMO:ALLOCATED_IN_MUNICIPALITY:MUNICIPALITY_INCIDENT_RESOLUTION:UNSOLVED_BY_MUNICIPALITY:ASSIGNED_TO_COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:IMPROVEMENT:NEW_DEVELOPMENT:IMPACT_ANALYSIS:FUNCTIONAL_AND_TECHNICAL_DESIGN_DOCUMENTS_REVISION:APPROVAL:COMPANY_INCIDENT_RESOLUTION:UNSOLVED_BY_COMPANY:INCIDENCE_CLOSURE
 26. PLAN_REGISTRATION:TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:EXISTS_PLAN:PMO_INCIDENT_RESOLUTION:UNSOLVED_BY_PMO:UNALLOCATED_IN_MUNICIPALITY:NOT_ASSIGNED_TO_COMPANY:INCIDENCE_CLOSURE
 27. PLAN_REGISTRATION:TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:EXISTS_PLAN:PMO_INCIDENT_RESOLUTION:UNSOLVED_BY_PMO:UNALLOCATED_IN_MUNICIPALITY:ASSIGNED_TO_COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:INCIDENT:COMPANY_INCIDENT_RESOLUTION:SOLVED_BY_COMPANY:INCIDENCE_CLOSURE
 28. PLAN_REGISTRATION:TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:EXISTS_PLAN:PMO_INCIDENT_RESOLUTION:UNSOLVED_BY_PMO:UNALLOCATED_IN_MUNICIPALITY:ASSIGNED_TO_COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:INCIDENT:COMPANY_INCIDENT_RESOLUTION:SOLVED_BY_COMPANY:INCIDENCE_CLOSURE
-

- MO_IS_ABLE:RESOURCE_ALLOCATION:EXISTS_PLAN:PMO_INCIDENT_RESOLUTION:UNSOLVED BY PMO:UNALLOCATED IN MUNICIPALITY:ASSIGNED TO COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:IMPROVEMENT:APPROVAL:COMPANY_INCIDENT_RESOLUTION:SOLVED_BY_COMPANY:INCIDENCE_CLOSURE
29. PLAN_REGISTRATION:TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:EXISTS_PLAN:PMO_INCIDENT_RESOLUTION:UNSOLVED BY PMO:UNALLOCATED IN MUNICIPALITY:ASSIGNED TO COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:IMPROVEMENT:NEW_DEVELOPMENT:IMPACT_ANALYSIS:FUNCTIONAL_AND_TECHNICAL_DESIGN_DOCUMENTS_REVISION:NOT_ALLOWED:INCIDENCE_CLOSURE
 30. PLAN_REGISTRATION:TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:EXISTS_PLAN:PMO_INCIDENT_RESOLUTION:UNSOLVED BY PMO:UNALLOCATED IN MUNICIPALITY:ASSIGNED TO COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:IMPROVEMENT:NEW_DEVELOPMENT:IMPACT_ANALYSIS:FUNCTIONAL_AND_TECHNICAL_DESIGN_DOCUMENTS_REVISION:APPROVAL:COMPANY_INCIDENT_RESOLUTION:UNSOLVED_BY_COMPANY:INCIDENCE_CLOSURE
 31. PLAN_REGISTRATION:TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:EXISTS_PLAN:PMO_INCIDENT_RESOLUTION:UNSOLVED BY PMO:ALLOCATED IN MUNICIPALITY:MUNICIPALITY_INcident_RESOLUTION:UNSOLVED BY MUNICIPALITY:ASSIGNED TO COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:INCIDENT_COMPANY_INCIDENT_RESOLUTION:UNSOLVED_BY_COMPANY:INCIDENCE_CLOSURE
 32. PLAN_REGISTRATION:TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:EXISTS_PLAN:PMO_INCIDENT_RESOLUTION:UNSOLVED BY PMO:ALLOCATED IN MUNICIPALITY:MUNICIPALITY_INcident_RESOLUTION:UNSOLVED BY MUNICIPALITY:ASSIGNED TO COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:IMPROVEMENT:IMPROVE:NOT_ALLOWED:INCIDENCE_CLOSURE
 33. PLAN_REGISTRATION:TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:EXISTS_PLAN:PMO_INCIDENT_RESOLUTION:UNSOLVED BY PMO:ALLOCATED IN MUNICIPALITY:MUNICIPALITY_INcident_RESOLUTION:UNSOLVED BY MUNICIPALITY:ASSIGNED TO COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:IMPROVEMENT:IMPROVE:APPROVAL:COMPANY_INCIDENT_RESOLUTION:UNSOLVED_BY_COMPANY:INCIDENCE_CLOSURE
 34. USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_NOT_ABLE:UNALLOCATED IN MUNICIPALITY:ASSIGNED TO COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:INCIDENT_COMPANY_INCIDENT_RESOLUTION:UNSOLVED_BY_COMPANY:INCIDENCE_CLOSURE
 35. USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_NOT_ABLE:UNALLOCATED IN MUNICIPALITY:ASSIGNED TO COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:IMPROVEMENT:IMPROVE:NOT_ALLOWED:INCIDENCE_CLOSURE
 36. USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_NOT_ABLE:UNALLOCATED IN MUNICIPALITY:ASSIGNED TO COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:IMPROVEMENT:IMPROVE:APPROVAL:COMPANY_INCIDENT_RESOLUTION:UNSOLVED_BY_COMPANY:INCIDENCE_CLOSURE

APPENDIX B

37. TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:NOT_EXISTS:UNSOLVED_BY_PMO:UNALLOCATED_IN_MUNICIPALITY:NOT_ASSIGNED_TO_COMPANY:INCIDENCE_CLOSURE
38. TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:NOT_EXISTS:UNSOLVED_BY_PMO:UNALLOCATED_IN_MUNICIPALITY:ASSIGNED_TO_COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:INCIDENT:COMPANY_INCIDENT_RESOLUTION:SOLVED_BY_COMPANY:INCIDENCE_CLOSURE
39. TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:NOT_EXISTS:UNSOLVED_BY_PMO:UNALLOCATED_IN_MUNICIPALITY:ASSIGNED_TO_COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:IMPROVEMENT:IMPROVE:APPROVAL:COMPANY_INCIDENT_RESOLUTION:SOLVED_BY_COMPANY:INCIDENCE_CLOSURE
40. TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:NOT_EXISTS:UNSOLVED_BY_PMO:UNALLOCATED_IN_MUNICIPALITY:ASSIGNED_TO_COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:IMPROVEMENT:NEW_DEVELOPMENT:IMPACT_ANALYSIS:FUNCTIONAL_AND_TECHNICAL_DESIGN_DOCUMENTS_REVISION:NOT_ALLOWED:INCIDENCE_CLOSURE
41. TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:NOT_EXISTS:UNSOLVED_BY_PMO:UNALLOCATED_IN_MUNICIPALITY:ASSIGNED_TO_COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:IMPROVEMENT:NEW_DEVELOPMENT:IMPACT_ANALYSIS:FUNCTIONAL_AND_TECHNICAL_DESIGN_DOCUMENTS_REVISION:APPROVAL:COMPANY_INCIDENT_RESOLUTION:UNSOLVED_BY_COMPANY:INCIDENCE_CLOSURE
42. TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:NOT_EXISTS:UNSOLVED_BY_PMO:ALLOCATED_IN_MUNICIPALITY:MUNICIPALITY_INCIDENT_RESOLUTION:UNSOLVED_BY_MUNICIPALITY:ASSIGNED_TO_COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:INCIDENT:COMPANY_INCIDENT_RESOLUTION:UNSOLVED_BY_COMPANY:INCIDENCE_CLOSURE
43. TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:NOT_EXISTS:UNSOLVED_BY_PMO:ALLOCATED_IN_MUNICIPALITY:MUNICIPALITY_INCIDENT_RESOLUTION:UNSOLVED_BY_MUNICIPALITY:ASSIGNED_TO_COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:IMPROVEMENT:IMPROVE:NOT_ALLOWED:INCIDENCE_CLOSURE
44. TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:NOT_EXISTS:UNSOLVED_BY_PMO:ALLOCATED_IN_MUNICIPALITY:MUNICIPALITY_INCIDENT_RESOLUTION:UNSOLVED_BY_MUNICIPALITY:ASSIGNED_TO_COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:IMPROVEMENT:IMPROVE:APPROVAL:COMPANY_INCIDENT_RESOLUTION:UNSOLVED_BY_COMPANY:INCIDENCE_CLOSURE
45. PLAN_REGISTRATION:TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:EXISTS_PLAN:PMO_INCIDENT_RESOLUTION:UNSOLVED_BY_PMO:UNALLOCATED_IN_MUNICIPALITY:ASSIGNED_TO_COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:INCIDENT:COMPANY_INCIDENT_RESOLUTION:UNSOLVED_BY_COMPANY:INCIDENCE_CLOSURE

-
46. PLAN_REGISTRATION:TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:EXISTS_PLAN:PMO_INCIDENT_RESOLUTION:UNSOLVED_BY_PMO:UNALLOCATED_IN_MUNICIPALITY:ASSIGNED_TO_COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:IMPROVEMENT:IMPROVE:NOT_ALLOWED:INCIDENCE_CLOSURE
 47. PLAN_REGISTRATION:TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:EXISTS_PLAN:PMO_INCIDENT_RESOLUTION:UNSOLVED_BY_PMO:UNALLOCATED_IN_MUNICIPALITY:ASSIGNED_TO_COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:IMPROVEMENT:IMPROVE:APPROVAL:COMPANY_INCIDENT_RESOLUTION:UNSOLVED_BY_COMPANY:INCIDENCE_CLOSURE
 48. TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:NOT_EXISTS:UNSOLVED_BY_PMO:UNALLOCATED_IN_MUNICIPALITY:ASSIGNED_TO_COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:INCIDENT_RESOLUTION:UNSOLVED_BY_COMPANY:INCIDENCE_CLOSURE
 49. TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:NOT_EXISTS:UNSOLVED_BY_PMO:UNALLOCATED_IN_MUNICIPALITY:ASSIGNED_TO_COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:IMPROVEMENT:IMPROVE:NOT_ALLOWED:INCIDENCE_CLOSURE
 50. TECHNICIAN_REGISTRATION:USER_REGISTRATION:INCIDENT_REGISTRATION:INCIDENT_PRIORITY_ASSIGNMENT:NOT_CRITICAL:PMO_IS_ABLE:RESOURCE_ALLOCATION:NOT_EXISTS:UNSOLVED_BY_PMO:UNALLOCATED_IN_MUNICIPALITY:ASSIGNED_TO_COMPANY:INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS:IMPROVEMENT:IMPROVE:APPROVAL:COMPANY_INCIDENT_RESOLUTION:UNSOLVED_BY_COMPANY:INCIDENCE_CLOSURE

B.2.3 Test Data

For specification of test values, data was extracted from Test Model and stored in a data base (see Section 5.3.1).

Table B.25 shows the test values for Incident Manager case study. These values are the example values passed to the test model from the requirements model.

B.3 Test Case Generation

As one can observe in the list of test scenarios (Section B.2.2), the test cases are grouped into 50 possible test scenarios, all of which were defined from the requirements model shown in Figure B.2.

APPENDIX B

Table B.25. Values for variables of test model for Incident Management

Test case	Test Item	Variable	Data Type	Concrete Values
USER_REGISTRATION	USER	id_user	number	100
USER_REGISTRATION	USER	name	text	Pepe Pérez
TECHNICIAN_REGISTRATION	TECHNICIAN	id_technician	number	200
TECHNICIAN_REGISTRATION	TECHNICIAN	name	text	Juan Valverde
PLAN_REGISTRATION	RESOLUTION_PLAN	id_plan	number	200
PLAN_REGISTRATION	RESOLUTION_PLAN	name	text	Enable activity register
PLAN_REGISTRATION	RESOLUTION_PLAN	step_sequence	text	1- Log in as administrator 2- Select option "Enable activity"
INCIDENT_REGISTRATION	INCIDENT	id_incident	number	501
INCIDENT_REGISTRATION	INCIDENT	request_type	text	Request
INCIDENT_REGISTRATION	INCIDENT	component	text	Activities
INCIDENT_REGISTRATION	INCIDENT	contact_information	text	RRHH secretary
INCIDENT_REGISTRATION	INCIDENT	Initial_scope	text	no critical
INCIDENT_REGISTRATION	INCIDENT	subject	text	Enable activity
INCIDENT_REGISTRATION	INCIDENT	Description	text	Activity cannot be created
INCIDENT_REGISTRATION	INCIDENT	step_sequence	text	1- Log in as administrator 2- Select option "Enable activity"
INCIDENT_PRIORITY_ASSIGNMENT	Initial_scope	state	text	Pending Review
INCIDENT_PRIORITY_ASSIGNMENT	Initial_scope	progress	text	INCIDENT PRIORITY ASSIGNMENT
INCIDENT_PRIORITY_ASSIGNMENT	Initial_scope	progress	text	Bug
NOT_CRITICAL	Estimated_scope	State	text	Incident Revision
NOT_CRITICAL	Estimated_scope	progress	text	REGISTER SCOPE
NOT_CRITICAL	Estimated_scope	Estimated_scope	text	NOT CRITICAL
CRITICAL	Estimated_scope	State	text	Incident Revision
CRITICAL	Estimated_scope	progress	text	REGISTER

	scope			SCOPE
CRITICAL	Estimated_scope	Estimated_scope	text	CRITICAL
PMO_IS_ABLE	progress	State	text	PMO IS ABLE
PMO_IS_ABLE	progress	progress	text	ASSESS PMO CAPACITY TO SOLVE
PMO_IS_NOT_ABLE	progress	state	text	PMO IS NOT ABLE
PMO_IS_NOT_ABLE	progress	progress	text	ASSESS PMO CAPACITY TO SOLVE
RESOURCE_ALLOCATION	RESOURCE_ALLOCATION	id_allocation	text	500
EXISTS_PLAN	plan_response	plan_response	text	EXISTS_PLAN
NOT_EXISTS	plan_response	plan_response	text	NOT EXISTS
UNSOLVED_BY_PMO	progress	state	text	UNSOLVED BY PMO
UNSOLVED_BY_PMO	progress	Progress	text	PMO_RESOLUTION
SOLVER_BY_PMO	progress	state	text	SOLVED BY PMO
SOLVER_BY_PMO	progress	progress	text	PMO_RESOLUTION
INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS	EXTERNAL_COMPANY_ANALYSIS	id_analysis	Number	800
INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS	EXTERNAL_COMPANY_ANALYSIS	subject	text	analysis of company XYZ
INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS	EXTERNAL_COMPANY_ANALYSIS	description	text	analysis of incident
INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS	EXTERNAL_COMPANY_ANALYSIS	analysis	text	This is a software improvement
INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS	EXTERNAL_COMPANY_ANALYSIS	implications	text	Access to the database must be checked
INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS	EXTERNAL_COMPANY_ANALYSIS	possible_solutions	text	Login as Administrator
INCIDENT_AND_IMPACT_EXTERNAL_COMPANY_ANALYSIS	EXTERNAL_COMPANY_ANALYSIS	time_estimation	text	2 days
IMPROVEMENT	progress	state	text	IMPROVEMENT
IMPROVEMENT	progress	progress	text	ACTION PLAN DEFINITION
INCIDENT	progress	state	text	INCIDENT
INCIDENT	progress	progress	text	ACTION PLAN DEFINITION
NEW_DEVELOPMENT	progress	state	text	NEW DEVELOPMENT

APPENDIX B

NEW_DEVELOPMENT	progress	progress	text	IMPROVEMENT EVALUATION
IMPROVE	progress	state	text	IMPROVE
IMPROVE	progress	progress	text	IMPROVEMENT EVALUATION
IMPACT_ANALYSIS	cost_estimation	state	text	Solution analysis
IMPACT_ANALYSIS	cost_estimation	progress	text	None
IMPACT_ANALYSIS	cost_estimation	time_estimation	text	1 week
IMPACT_ANALYSIS	cost_estimation	cost_estimation	text	500.00
PMO_INCIDENT_RESOLUTION	step_sequence	state	text	PMO
PMO_INCIDENT_RESOLUTION	step_sequence	progress	text	none
PMO_INCIDENT_RESOLUTION	step_sequence	step_sequence	text	PMO resolution
ALLOCATED_IN_MUNICIPALITY	progress	state	text	ALLOCATED IN MUNICIPALITY
ALLOCATED_IN_MUNICIPALITY	progress	progress	text	MUNICIPALITY ASSIGNMENT EVALUATION
UNALLOCATED_IN_MUNICIPALITY	progress	state	text	UNALLOCATED IN MUNICIPALITY
UNALLOCATED_IN_MUNICIPALITY	progress	progress	text	MUNICIPALITY ASSIGNMENT EVALUATION
MUNICIPALITY_INCIDENT_RESOLUTION	step_sequence	state	text	Municipality resolution
MUNICIPALITY_INCIDENT_RESOLUTION	step_sequence	progress	text	Municipality
MUNICIPALITY_INCIDENT_RESOLUTION	step_sequence	step_sequence	text	Login as Administrator
UNSOLVED_BY_MUNICIPALITY	progress	state	text	UNSOLVED BY MUNICIPALITY
UNSOLVED_BY_MUNICIPALITY	progress	progress	text	MUNICIPALITY RESOLUTION VALIDATION
SOLVED_BY_MUNICIPALITY	progress	state	text	SOLVED BY MUNICIPALITY
SOLVED_BY_MUNICIPALITY	progress	progress	text	MUNICIPALITY RESOLUTION VALIDATION
ASSIGNED_TO_COMPANY	progress	state	text	ASSIGNED TO COMPANY
ASSIGNED_TO_COMPANY	progress	progress	text	COMPANY ASSIGNMENT EVALUATION
NOT_ASSIGNED_TO_	progress	state	text	NOT ASSIGNED

COMPANY				TO COMPANY
NOT_ASSIGNED_TO_COMPANY	progress	progress	text	COMPANY ASSIGNMENT EVALUATION
FUNCTIONAL_AND_TECHNICAL_DESIGN_DOCUMENTS_REVISION	progress	state	text	DOCUMENTS REVISION
FUNCTIONAL_AND_TECHNICAL_DESIGN_DOCUMENTS_REVISION	progress	progress	text	FUNCTIONAL AND TECHNICAL DESIGN DOCUMENTS REVISION
APPROVAL	progress	state	text	APPROVAL
APPROVAL	progress	progress	text	NEED AND VIABILITY EVALUATION
NOT_ALLOWED	progress	state	text	NOT ALLOWED
NOT_ALLOWED	progress	progress	text	NEED AND VIABILITY EVALUATION
COMPANY_INCIDENT_RESOLUTION	step_sequence	state	text	company resolution
COMPANY_INCIDENT_RESOLUTION	step_sequence	progress	text	Progress
COMPANY_INCIDENT_RESOLUTION	step_sequence	step_sequence	text	Select the new option
SOLVED_BY_COMPANY	progress	state	text	SOLVED BY COMPANY
SOLVED_BY_COMPANY	progress	progress	text	COMPANY RESOLUTION VALIDATION
UNSOLVED_BY_COMPANY	progress	state	text	UNSOLVED BY COMPANY
UNSOLVED_BY_COMPANY	progress	progress	text	COMPANY RESOLUTION VALIDATION
INCIDENCE_CLOSURE	state	state	text	CLOSED

In order to illustrate the test case generation phase of the case study, we selected all test cases to be generated (see Section 5.4.3), which include some negative conditions such as out of range values, based on variable partitions that can be derived from CS information, constraint violations, minimum cardinality violation, and, unique value violation for class variables.

The result of this phase is 115 different test cases to test the 50 test scenarios. For instance, test scenario number 2 with test case number 51 belongs to the set of test items shown in Figure B.4.

These test items represent the report of a critical incident that is not solved by the PMO and is not assigned to either the municipality or a company, therefore its final status is closed.

```

1 private import PMO::*;
2 public import Alf::Library::BasicTypes::*;
3 public import Alf::Library::Asserts::*;
4 // Conceptual Schema under Test : PMO
5 // Goal: Validate the scenario N° 2 with valid value (test case positive)
6 // The Script consists of 50 Test Scenarios
7 activity AbsTScenario_2_PMO () {
8   // Test Case: USER_REGISTRATION
9   // Services
10  user = new USER(p_atrid_user=100,p_atrname= "Pepe Pérez");
11  // Test Case: INCIDENT_REGISTRATION
12  // Services
13  incident = new INCIDENT(p_atrid_incident=501,p_atrrequest_type= "request",
14  p_atrcomponent= "activities",p_atrcontact_information= "secretaria RRHH",
15  p_atrinitial_scope= "no critical",p_atrsubject= "Registro de actividad",
16  p_atrdescription= "no puede crearse la actividad",
17  p_atrstep_sequence= "1- está deshabilitada la opcion",p_agruser=user_);
18  // Links
19 //user incident.createLink(user ,incident );
20 // Test Case: INCIDENT_PRIORITY_ASSIGNMENT
21 // Triggers
22 incident._set_initial_scope(pt_state= "Pending Review",pt_progress= "INCIDENT_PRIORITY_ASSIGNMENT",
23 pt_initial_scope= "bug",p_thisINCIDENT=incident_);
24 // Test Case: CRITICAL
25 // Triggers
26 incident._set_estimated_scope(pt_state= "Incident Revision",pt_progress= "REGISTER SCOPE",
27 pt_estimated_scope= "CRITICAL",p_thisINCIDENT=incident_);
28 // Test Case: UNALLOCATED_IN_MUNICIPALITY
29 // Triggers
30 incident._set_progress(pt_state= "UNALLOCATED IN MUNICIPALITY",
31 pt_progress= "MUNICIPALITY_ASSIGNMENT_EVALUATION",p_thisINCIDENT=incident_);
32 // Test Case: NOT_ASSIGNED_TO_COMPANY
33 // Triggers
34 incident._set_progress(pt_state= "NOT ASSIGNED TO COMPANY",
35 pt_progress= "COMPANY_ASSIGNMENT_EVALUATION",p_thisINCIDENT=incident_);
36 // Test Case: INCIDENCE_CLOSURE
37 // Triggers
38 incident._set_state(pt_state= "closed",p_thisINCIDENT=incident_);
39 }

```

Figure B.4. Test cases of the test scenario #2

B.4 Mutant Generation

This step was performed automatically by means of the CoSTest Mutant Generator (see Sections 5.5 and 7.7). Figure B.5. shows an excerpt of the UML class diagram used as CS for IM case study as an example of the result obtained in the mutation step. Table B.26 shows eight mutants that were generated from the IM conceptual schema after applying the mutant generation process to the case study.

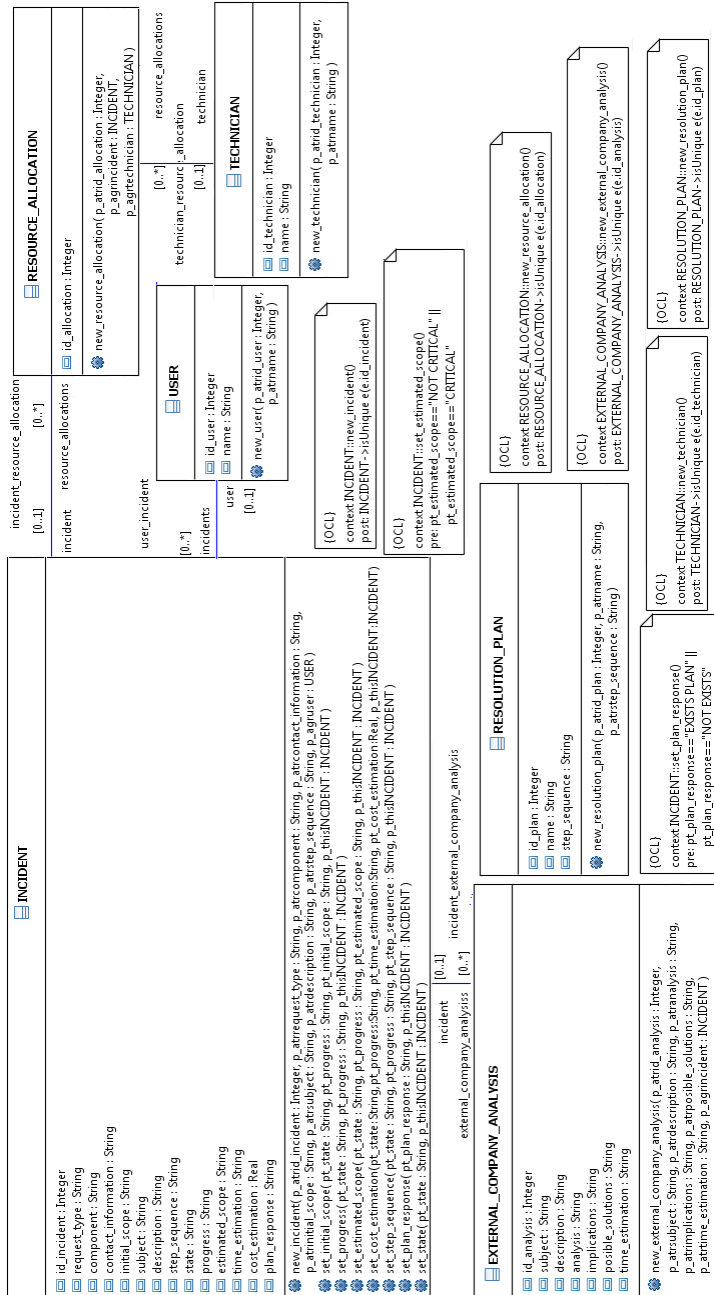


Figure B.5. Excerpt of the Conceptual Schema for Incident Management System

Table B.26. List of First Order Mutants generated for the case study

No	Mutation Operator	Mutation Operator Rule	Mutated elements
1	MAS_2	Deletes an Association	user_incident
2	MCO_5	Deletes a constraint)	context INCIDENTset_estimated_scope() precondition pt_estimated_scope=="NOT CRITICAL" pt_estimated_scope=="CRITICAL"
3	MPA_4	Deletes a Parameter from a Class Operation)	p_thisINCIDENT(set_cost_estimation-INCIDENT)
4	UPA2_2	Adds a Parameter to a Class Operation)	set_initial_scope-INCIDENT
5	WAS1_1	Changes the member ends)	incident_external_company_analysis
6	WCL1_2	Changes Class visibility property)	EXTERNAL_COMPANY_ANALYSIS
7	WCOS_6	Changes the 12 operator (==) with (!=))	context INCIDENTset_progress() precondition pt_state=="PMO IS ABLE" pt_state=="PMO IS NOT ABLE" pt_state=="UNALLOCATED IN MUNICIPALITY" pt_state=="ALLOCATED IN MUNICIPALITY" pt_state=="IMPROVEMENT" pt_state=="NEW DEVELOPMENT" pt_state=="APPROVAL" pt_state=="NOT ALLOWED" pt_state=="IMPROVE" pt_state=="INCIDENT" pt_state=="NOT ASSIGNED TO COMPANY" pt_state=="ASSIGNED TO COMPANY" pt_state=="SOLVED BY PMO" pt_state=="UNSOLVED BY PMO" pt_state=="SOLVED BY MUNICIPALITY" pt_state=="UNSOLVED BY MUNICIPALITY" pt_state=="SOLVED BY COMPANY" pt_state=="UNSOLVED BY COMPANY" pt_state=="DOCUMENTS REVISION"
8	WOP2_7	Changes the operation visibility property)	set_state-INCIDENT

B.5 Test Execution

This phase of our approach is automatic; we only had to select the mutated CSs and then run the test cases against them. We divide this section into two main subsections (B.5.1 and B.5.2), each of which focuses on a specific tasks of the framework.

B.5.1 Generation of the Executable Conceptual Schema Under Test

This step was performed automatically by means of the UML2ALF transformation implemented in the CSUT Processor that is provided by CoSTest (see Section 6.3.6). As an example of the result obtained in the Executable Conceptual Schema Generation step, Figure B.5 shows the UML CD-based CS for Incident Management System and the Figures B.6-B.16 show the ALF units that were generated after applying the UML2ALF transformation to mutated IM case study.

```

1 package PMO {
2     public class INCIDENT;
3     public class EXTERNAL_COMPANY_ANALYSIS;
4     public class RESOLUTION_PLAN;
5     public class USER;
6     public class RESOURCE_ALLOCATION;
7     public class TECHNICIAN;
8     public assoc incident_external_company_analysis;
9     public assoc user_incident;
10    public assoc incident_resource_allocation;
11    public assoc technician_resource_allocation;
12 }

```

Figure B.6. ALF unit for PMO class

```

1 namespace PMO;
2 assoc incident_external_company_analysis{
3     public 'incident':INCIDENT[1];
4     public 'external_company_analysiss':EXTERNAL_COMPANY_ANALYSIS[*];
5 }

```

Figure B.7. ALF unit for Incident_external_company association

```

1 namespace PMO;
2 public import Alf::Library::PrimitiveBehaviors::StringFunctions::*;
3 public import Alf::Library::BasicTypes::*;
4 class EXTERNAL_COMPANY_ANALYSIS{
5     public id_analysis: Integer;
6     public subject: String;
7     public description: String;
8     public analysis: String;
9     public implications: String;
10    public possible_solutions: String;
11    public time_estimation: String;
12 @Create EXTERNAL_COMPANY_ANALYSIS(in p_atrid_analysis:Integer,in p_atrsubject:String,
13     in p_atrdescription:String,
14     in p_atranalysis:String,
15     in p_atrimPLICATIONS:String,
16     in p_atrpossible_solutions:String,
17     in p_atrtime_estimation:String,
18     in p_agrincident:INCIDENT ) {
19     this.id_analysis=p_atrid_analysis;
20     this.subject=p_atrsubject;
21     this.description=p_atrdescription;
22     this.analysis=p_atranalysis;
23     this.implications=p_atrimPLICATIONS;
24     this.possible_solutions=p_atrpossible_solutions;
25     this.time_estimation=p_atrtime_estimation;
26     incident_external_company_analysis.createLink(p_agrincident,this);
27     if ( EXTERNAL_COMPANY_ANALYSIS->isUnique e(e.id_analysis){} else
28     {WriteLine("Error in PostCondition 'context EXTERNAL_COMPANY_ANALYSIS::
29     new_external_company_analysis() post: EXTERNAL_COMPANY_ANALYSIS->isUnique
30     e(e.id_analysis)");
31     this.destroy(); }
32 } }

```

Figure B.8. ALF unit for EXTERNAL_COMPANY_ANALYSYS class

```

1 namespace PMO;
2 assoc incident_resource_allocation{
3     public 'incident':INCIDENT[1];
4     public 'resource_allocations':RESOURCE_ALLOCATION[*];
5 }

```

Figure B.9. ALF unit for incident_resource_allocation association

```

1 namespace PMO;
2 assoc technician_resource_allocation{
3     public 'technician':TECHNICIAN[1];
4     public 'resource_allocations':RESOURCE_ALLOCATION[*];
5 }

```

Figure B.10. ALF unit for technician_resource_allocation association

```

1 namespace PMO;
2 public import Alf::Library::PrimitiveBehaviors::StringFunctions::*;
3 public import Alf::Library::BasicTypes::*;
4 class INCIDENT {
5     public id_incident: Integer;
6     public request_type: String;
7     public component: String;
8     public contact_information: String;
9     public initial_scope: String;
10    public subject: String;
11    public description: String;
12    public step_sequence: String;
13    public state: String;
14    public progress: String;
15    public estimated_scope: String;
16    public time_estimation: String;
17    public cost_estimation: Real;
18    public plan_response: String;
19    @Create INCIDENT( in p_atrid_incident:Integer, in p_atrrequest_type:String, in p_atrocomponent:String,
20                    in p_atrcontact_informtion:String,
21                    in p_atrinitial_scope:String,
22                    in p_atrsubject:String,
23                    in p_atrdescription:String,
24                    in p_atrstep_sequence:String,
25                    in p_agruser:USER ) {
26        this.id_incident=p_atrid_incident;
27        this.request_type=p_atrrequest_type;
28        this.component=p_atrocomponent;
29        this.contact_information=p_atrcontact_informtion;
30        this.initial_scope=p_atrinitial_scope;
31        this.subject=p_atrsubject;
32        this.description=p_atrdescription;
33        this.step_sequence=p_atrstep_sequence;
34    user_incident.createLink(p_agruser,this);
35        if ( INCIDENT->isUnique e(e.id_incident) ) {} else
36        { WriteLine("Error in PostCondition 'context INCIDENT::new_incident()
37    post: INCIDENT->isUnique e(e.id_incident)");
38    this.destroy(); }
39    }
40    public set_initial_scope( in pt_state:String, in pt_progress:String, in pt_initial_scope:String,
41                            in p_thisINCIDENT:INCIDENT ) {
42        this.state=pt_state;
43        this.progress=pt_progress;
44        this.initial_scope=pt_initial_scope;
45    }
46    public set_progress( in pt_state:String, in pt_progress:String, in p_thisINCIDENT:INCIDENT ) {
47        if ( pt_state=="PMO IS ABLE" || pt_state=="PMO IS NOT ABLE" || pt_state=="UNALLOCATED IN MUNICIPALITY"
48        else
49        { WriteLine("Error in Precondition 'context INCIDENT::set_progress()
50    pre: pt_state=='PMO IS ABLE' || pt_state=='PMO IS NOT ABLE' || pt_state=='UNALLOCATED IN MUNICIPALITY' ||
51    }
52    }
53    this.state=pt_state;
54    this.progress=pt_progress;
55    }
56    public set_estimated_scope( in pt_state:String, in pt_progress:String, in pt_estimated_scope:String,
57                              in p_thisINCIDENT:INCIDENT ) {
58        if ( pt_estimated_scope=="NOT CRITICAL" || pt_estimated_scope=="CRITICAL" ) {} else
59        { WriteLine("Error in Precondition 'context INCIDENT::set_estimated_scope()
60    pre: pt_estimated_scope=='NOT CRITICAL' || pt_estimated_scope=='CRITICAL'"); }
61    this.state=pt_state;
62    this.progress=pt_progress;
63    this.estimated_scope=pt_estimated_scope;
64    }
65    public set_cost_estimation( in pt_state:String, in pt_progress:String, in pt_time_estimation:String,
66                              in pt_Cost_estimation:Real,
67                              in p_thisINCIDENT:INCIDENT ) {
68        this.state=pt_state;
69        this.progress=pt_progress;
70        this.time_estimation=pt_time_estimation;
71        this.cost_estimation=pt_cost_estimation;
72    }
73    public set_step_sequence( in pt_state:String, in pt_progress:String, in pt_step_sequence:String,
74                             in p_thisINCIDENT:INCIDENT ) {
75        this.state=pt_state;
76        this.progress=pt_progress;
77        this.step_sequence=pt_step_sequence;
78    }
79    public set_plan_response( in pt_plan_response:String, in p_thisINCIDENT:INCIDENT ) {
80        if ( pt_plan_response=="EXISTS PLAN" || pt_plan_response=="NOT EXISTS" ) {} else
81        { WriteLine("Error in Precondition 'context INCIDENT::set_plan_response()
82    pre: pt_plan_response=='EXISTS PLAN' || pt_plan_response=='NOT EXISTS'"); }
83    this.plan_response=pt_plan_response;
84    }
85    public set_state( in pt_state:String, in p_thisINCIDENT:INCIDENT ) {
86        this.state=pt_state; }

```

Figure B.11. ALF unit for INCIDENT class

```

1 namespace PMO;
2 public import Alf::Library::PrimitiveBehaviors::StringFunctions::*;
3 public import Alf::Library::BasicTypes::*;
4 class RESOLUTION_PLAN{
5     public id_plan: Integer;
6     public name: String;
7     public step_sequence: String;
8     @Create RESOLUTION_PLAN( in p_atrid_plan:Integer, in p_atrname:String,
9         in p_atrstep_sequence:String ) {
10         this.id_plan=p_atrid_plan;
11         this.name=p_atrname;
12         this.step_sequence=p_atrstep_sequence;
13         // postcondition
14         if ( RESOLUTION_PLAN->isUnique e(e.id_plan){} else
15         { WriteLine("Error in PostCondition 'context RESOLUTION_PLAN::new_resolution_plan()
16         post: RESOLUTION_PLAN->isUnique e(e.id_plan)");
17         this.destroy(); }
18 } }

```

Figure B.12. ALF unit for RESOLUTION_PLAN class

```

1 namespace PMO;
2 public import Alf::Library::PrimitiveBehaviors::StringFunctions::*;
3 public import Alf::Library::BasicTypes::*;
4 class RESOURCE_ALLOCATION{
5     public id_allocation: Integer;
6     @Create RESOURCE_ALLOCATION( in p_atrid_allocation:Integer, in p_agrincident:INCIDENT,
7         in p_agrtechnician:TECHNICIAN ) {
8         this.id_allocation=p_atrid_allocation;
9         incident_resource_allocation.createLink(p_agrincident,this);
10        technician_resource_allocation.createLink(p_agrtechnician,this);
11        if ( RESOURCE_ALLOCATION->isUnique e(e.id_allocation){} else
12        { WriteLine("Error in PostCondition 'context RESOURCE_ALLOCATION::new_resource_allocation()
13        post: RESOURCE_ALLOCATION->isUnique e(e.id_allocation)");
14        this.destroy(); }
15 } }

```

Figure B.13. ALF Unit for RESOURCE_ALLOCATION class

```

1 namespace PMO;
2 public import Alf::Library::PrimitiveBehaviors::StringFunctions::*;
3 public import Alf::Library::BasicTypes::*;
4 class TECHNICIAN{
5     public id_technician: Integer;
6     public name: String;
7     @Create TECHNICIAN(in p_atrid_technician:Integer,in p_atrname:String) {
8         this.id_technician=p_atrid_technician;
9         this.name=p_atrname;
10        if ( TECHNICIAN->isUnique e(e.id_technician){} else
11        { WriteLine("Error in PostCondition 'context TECHNICIAN::new_technician()
12        post: TECHNICIAN->isUnique e(e.id_technician)");
13        this.destroy(); }
14 } }

```

Figure B.14. ALF unit for TECHNICIAN class

```

1 namespace PMO;
2 public import Alf::Library::PrimitiveBehaviors::StringFunctions::*;
3 public import Alf::Library::BasicTypes::*;
4 class USER{
5     public id_user: Integer;
6     public name: String;
7     @Create USER( in p_atrid_user:Integer,in p_atrname:String) {
8         this.id_user=p_atrid_user;
9         this.name=p_atrname;
10 } }

```

Figure B.15. ALF unit for USER class


```

1 namespace PMO;
2 assoc user_incident{
3     public 'user':USER[1];
4     public 'incidents':INCIDENT[*];
5 }

```

Figure B.16. ALF unit for user_incident association

B.5.2 Generation of the Execution Trace

The testing execution phase of the mutants was performed by means of the Test Processor, which is integrated in CoSTest (see Section 7.6). As an example of the execution trace, Figure B.17 shows the execution trace of the test cases against the MAS_2 mutant.

```

-----Test Case :001-----
-----Test Case :002-----
Error in PostCondition 'context RESOLUTION_PLAN::new_resolution_plan()
post: RESOLUTION_PLAN->isUnique e(e.id_plan)'
-----Test Case :003-----
-----Test Case :004-----
Error in PostCondition 'context TECHNICIAN::new_technician()
post: TECHNICIAN->isUnique e(e.id_technician)'
-----Test Case :005-----
-----Test Case :006-----
-----Test Case :007-----
Error in PostCondition 'context INCIDENT::new_incident()
post: INCIDENT->isUnique e(e.id_incident)'
-----Test Case :008-----
Constraint violations:
  propertyAccessExpressionFeatureResolution          in          C:\Users\PC-
Mafer\workspace\COSTest\ExecutableTestCases\UML-ALF\PMO\PMO_TS_001_TC_008.alf
at line 22, column 62

```

Figure B.17. Example of Execution Trace for the MAS_2 mutant

From these results, we can see that test cases numbers 2, 4 and 7 are negative test cases to validate constraints; therefore, the expected result is an error in postcondition because the constraints are violated in order to test their existence as we can see in Figure B.17. The result for test case number 8 shows that line 22 produces the fault *property Access Expression Feature Resolution*, and then the testing process is stopped.

B.6 Test Evaluation

This phase of our approach is automatic; therefore, we did not have to perform any work at this point. The result of the test evaluation phase was (1) a defect report that provides feedback to the tester of the CS and also (2) a coverage report comparing the elements included in the conceptual schema and those executed in the test cases. Table B.27 shows the verdicts for each CS mutant, the found defects and the CS elements affected by the defect

Table B.27. Testing results for the mutants of Table B.26

CSUT	Final Verdict	Found Defects	Localized Element
MAS_2	Failed	Missing or private Association	Association=user_incident;
MCO_5	Failed	Missing Constraint	contextincident::set_plan_response() pre:pt_plan_response=='exist splan'
MPA_4	Inconclusive	Incorrect Parameter Data Type	Class=INCIDENT; Operation=set_cost_estimation()
WAS1_1	Failed	Missing or private Association	Association=incident_external_co mpany_analysis;
WCL1_2	Inconclusive	Missing Class (or private)	Class=EXTERNAL_COMPANY_ ANALYSIS; Operation=new EXTERNAL_COMPANY_ANALY SIS()
WCO5_6	Failed	Missing Constraint	contextincident::set_progress() pre:pt_state=='improve'
WOP2_7	Failed	Missing Operation (or private)	Class=INCIDENT; Operation=set_state()

An example of a CoSTest report is shown in Figure B.18, in which seven of the eight test cases were successfully passed in the testing process.

The eighth test case returns the verdict Fail. Then, the execution trace is analysed by using the information shown in Figure B.17 and the defect missing (or private) Association is reported.

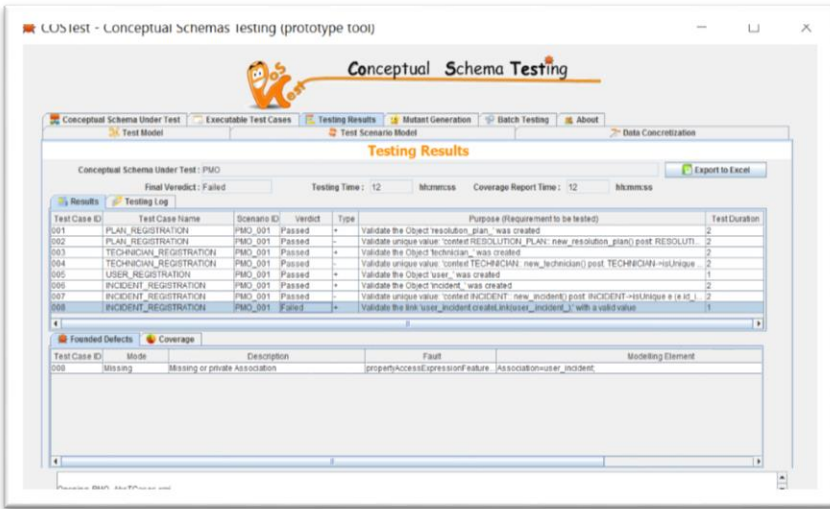


Figure B.18. Defect report obtained in the testing process for MAS_2 CS

Thus, Figure B.19 shows the coverage report generated for the MAS_2 CS mutant by comparing the elements included in the conceptual schema and those executed in the test cases.

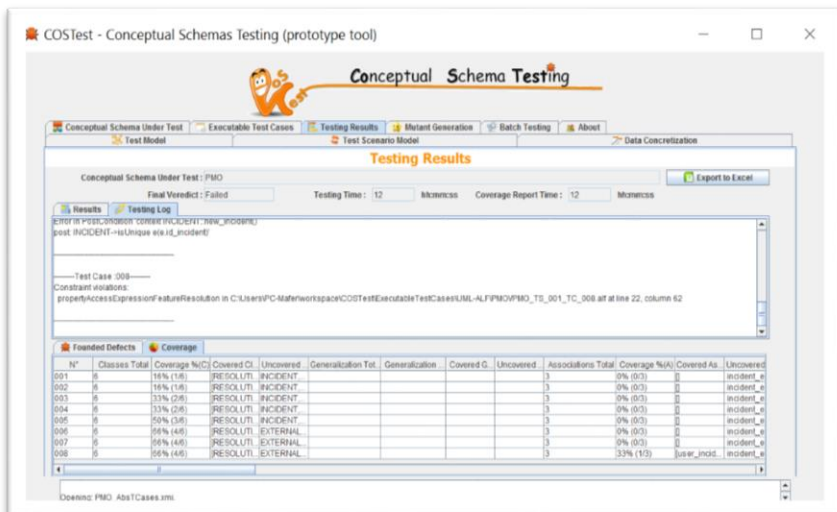


Figure B.19. Coverage report obtained in the testing process for MAS_2 CS

B.7 Conclusions

This appendix presents a case study that exemplifies the CoSTest framework described in this thesis. To this end, the appendix applies the validation framework to eight CS mutants, which represent the conceptual schema for the Incident Manager System that was defined for the everis company. The application of the approach to an industrial case study allowed us to identify some of CoSTest's limitations (such as highlighting the failed test cases in red). However, it also allowed us to be optimistic since CoSTest successfully supported the design, execution, and evaluation of the test cases for detecting defects in the mutants generated from the CS of the case study.

Appendix C

Supplementary Material on the Evaluation Study

This appendix includes material that was used during the evaluation study that is presented in Chapter 8. First, the appendix presents several instruments that were employed during the execution phase of the study. These instruments are the characterization form, the CoSTest tool installation guide, the guideline with the task template, and the interview questions, which are given in Sections C.1, C.2, C.3 and C.4, respectively.

C1. Characterization Form

This section presents the characterization form. As Section 8.5.9 describes, the characterization form is divided in two parts. The first part requests demographic data, such as gender, age, and work status. This part of the form is shown in Figure C.1. The second part includes twenty-two questions concerning the subjects' level of experience of the topics covered by the study (e.g. modelling activities and testing). This part of the form is shown in Figures C.2 - C.4.

Characterization Form		
Please state your Age :		
<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
<24	25-29	30-34
<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
35-39	40-44	45-49
<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
>50		
Please state your Gender:		
<input type="radio"/>	<input type="radio"/>	
Male	Female	
What is the highest level of education that you have completed?		
<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
High school (or equivalent)	Technical school	Bachelor degree
<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Master degree	Doctoral degree	Other (specify)

Figure C.1. Characterization form: Demographic data

Profile and Demographics of Practitioners

Q1. What is your current job position?

- Project manager
- Test manager
- Software tester
- Modeller/Analyst
- Developer
- Other _____

Q2. Do you have any previous experience in UML modelling?

No Yes

Q3. If you have answered “yes”, how many years of experience do you have in UML modelling?

- < 2 years
- 2 - 5 years
- 6 - 10 years
- 10 years

Q4. If you have answered “yes”, how do you qualify your experience regarding to use Modelling tools to do modelling activities in your job? (e.g UML2 diagrams, Papyrus)

Tool	Poor	Fair	Average	Good	Excellent
Tool 1: UML2 diagrams	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Tool 2: Papyrus	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Others (specify)					
Tool 3:	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Tool 4:	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Q5. Do you have any previous experience regarding Software Testing?

No Yes

Q6. If you have answered “yes”, how many years of experience do you have in software testing?

- < 2 years
- 2 - 5 years
- 6 - 10 years
- 10 years

Q7. Do you have any formal training in software testing or quality assurance?

No Yes

Q8. If you answer was Yes in Q3. What formal training was taken?

Q9. What type of application is developed by your company?

- Mobile applications
- Desktop applications
- Embedded applications
- Web based applications
- Other _____

Figure C.2. Characterization form: Experience (1)

Q10. How many employee work in your company?

- less than 50
- 50 – 250
- More than 250

Q11. Which one of the following software testing methods are commonly used in your company?

- White box testing
- Black box testing
- Gray box testing
- Agile testing
- Ad hoc testing
- Others _____

Q12. Which one of the following software testing type is performed in your company?

- Unit testing
- Security testing
- Usability testing
- Smoke testing
- Functional testing
- Conceptual Schema testing
- Integration testing
- System testing
- Acceptance testing
- Automated testing
- Regression testing
- Other testing

Q13. In the most recent projects in your company, did you use any test automation tools to perform Software Functional Testing?

- No
- Yes

Q14. If the answer was Yes in Q13, which tool did you use and which was it purpose?

Q15. In the most recent projects in your company, do you perform Software Functional Testing at conceptual schema level?

- No
- Yes

Q16. If the answer was Yes in Q15, which testing type did you use?

- Manual
- Automatic

Q17. If the answer was Automatic in Q16, which tool did you use?

Q18. Please describe why you like to use Manual Functional Software Testing?

Q19. In your most recent project, which software test design techniques do you use?

- Boundary value analysis
- Equivalent partitioning
- Decision table
- State transition
- Use case testing
- Error Guessing
- Exploratory testing
- Other

Figure C.3. Characterization form: Experience (2)

Q20. To what extent do you agree with listed statement why your company do not use Functional Software Test Automation Tools?

	Strongly Disagree	Disagree	Neither Agree nor Disagree	Agree
Setting up automated tools is very difficult and costly				
Require scripting skills, which some testers lack				
Time-consuming to use				
Costly to use				
Difficult to use				
Lack of expertise				
Lack of supporting tools				
Maintenance difficulty and Cost				
Lack of information about the available tools				
Automation was not needed				

Q21. From your experience in your recent projects, please describe the main disadvantages or drawbacks of manual functional software testing?

Q22. Are you planning to change the way how you currently implement Software Functional Testing or do you have idea how to improve it, please describe your answer.

Figure C.4. Characterization form: Experience (3)

C2. CoSTest Tool Installation Guide

The CoSTest tool can be downloaded as a compressed bundle (*.zip/*.rar) from: <https://staq.dsic.upv.es/webstaq/costest/costest.zip>

- Double click on the filename from left list (e.g. VideoClub_mutant.umlclass)

To create the graphical view (if it does not exist)

- Click with right button on UML diagram (e.g. VideoClub_mutant.uml) and select the option “Initialize Class Diagram”
 - To select the parent folder “ConceptualSchemas” and enter the filename e.g. “VideoClub_mutant.umlclass”
 - Click on button Finish to generate the graphical view.

C3. Guideline with Task Template for VideoClub Case

This section presents the guideline with the tasks template for the VideoClub case (see Figures C.5 and C.6).

C4. User Acceptance Form

This section presents the user acceptance form. As Section 8.5.6 describes, we developed the user acceptance form following the Post-study System Usability Questionnaire [151], which suggests measuring perceived usefulness and perceived ease-of-use by means of two scales of 7-point Likert items, ranging from “strongly agree” (1) to “neutral” (4) to “strongly disagree” (7). The first of these two scales, which evaluates perceived usefulness, is graphically depicted in Figure C.7. The second scale, which evaluates perceived ease-of-use, is graphically depicted in Figure C.8.

GUIDELINE FOR VIDEOCLUB CASE	
To use the CoSTest tool execute the runServer.bat file, then execute the CoSTest.bat file.	
1. Convert the Requirement Model in a Test Model	<ul style="list-style-type: none"> a) Click on the Tab "Test Model" b) Click on the button "Select File" to select the Requirement Model file (e.g. VideoClub.cametamodel) from the folder "ReqModels" c) Click on the button "Save as" to choose the target folder "TestModels" and to write the filename as "VideoClub_TestModel.xml". d) Click on button "Test Model Generation" to generate the Test Model. e) Verify that the model graph is generated.
2. Convert the Test Model in a Test Scenarios Model with the abstract test cases	<ul style="list-style-type: none"> a) Click on the Tab "Abstract Test Cases" b) Click on the button "Select File" to select the Test Model file. The filename is the similar to the filename entered in the previous step but it ends with an underscore (e.g. VideoClub_TestModel_.xml) from the folder "TestModels". c) Click on the button "Save as" to choose the target folder "AbstractTestCases" and to write the filename as "VideoClub_Scenarios.xml" d) Click on button "Abstract Test Cases Generation" to generate the Test Scenario Model. The file VideoClub_Scenarios.xml is generated. e) Verify that the model tree is generated.
3. Concretize the variables of the Test Model with the values taken from Test model.	<ul style="list-style-type: none"> a) Click on the Tab "Data Concretization" b) Click on the button "Select File" to select the Test Model file. The filename is the similar to the filename entered in the first step (e.g. VideoClub_TestModel.xml). c) Click on the button "Generate from Model" to generate the data d) Verify that the variables list is generated.
4. Convert the Conceptual Schema Under Test (CSUT) in an Executable CSUT.	<ul style="list-style-type: none"> a) Click on the Tab "Conceptual Schema Under Test" b) Click on the button "Select File" to select the CSUT file (e.g. VideoClub_mutant.uml) from the folder "ConceptualSchemas" c) Click on the button "Select Folder" to choose the target folder "ExecutableTestCases\VideoClub" d) Click on button "CSUT Transformation" to generate the executable CSUT. The file "VideoClub.alf" is added in the left file list. e) Select the filename "VideoClub.alf" from the files list (left). f) Click on button "CSUT Parser" in order to verify the syntax. g) Verify that the parsing is OK
5. Execution of the test scripts on the Executable CSUT	<ul style="list-style-type: none"> a) Click on the Tab "Executable Test Cases" b) Click on the button "Select File" to choose the Abstract Test Cases file (e.g. VideoClub_Scenarios.xml) from the folder "AbstractTestCases" c) Click on the button "Select CSUT" to choose the CSUT (e.g. VideoClub_mutant.uml) from the folder "ConceptualSchemas" d) Click on the button "Select Folder" to choose the target folder "ExecutableTestCases\VideoClub"

Figure C.5. Guideline for VideoClub case (1)

e) Click on the button “Code Generation” to generate the abstract test cases
 f) Click on the button “Test Case Concretization” to generate the concrete and executable test cases
 g) Select from left file list the filenames that begin with “VideoClub_TS_1_TC_” (total 36 files).
 h) Click on the button “Testing”

6. Analyse the testing report generated by the tool and identify the faults. If the CSUT has no faults, the process has finished.

a) Click on the Tab “Testing Results”
 b) Review the test cases executed in the Results Table and searching if there is one test case with a “Failed” Verdict.
 c) Review the Founded Defects Table to identify the defect type and CSUT element (e. g. Class-Operation-Parameter, Association, Class-Constraint)
 d) Write the following information about the founded defects otherwise write “no defects”

Iteration	Test Scenario ID	Test Case ID	Verdict	Detected Fault	CSUT Element
1					
2					
3					
4					
5					

7. If the CSUT has detected faults, correct them on CSUT.

a) Go to Eclipse UML2 Diagrams
 b) Double click on the CSUT filename to open it
 c) Correct the defect according the identified defects
 d) Save the changes.
 e) Return to the CoSTest tool for the next iteration

Figure C.6. Guideline for VideoClub case (2)

The Post-Study Perceived Usefulness Questionnaire

		Strongly agree				Strongly disagree			
		1	2	3	4	5	6	7	NA
1	Using CoSTest in my job would enable me to accomplish tasks more quickly.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	
2	Using CoSTest would improve my job performance.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	
3	Using CoSTest in my job would increase my productivity.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	
4	Using CoSTest would enhance my effectiveness on the job.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	
5	Using CoSTest would make it easier to do my job.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	
6	I would find CoSTest useful in my job.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	

Figure C.7. User Acceptance Form: Perceived Usefulness

The Post-Study Ease to Use Questionnaire		Strongly agree					Strongly disagree		
		1	2	3	4	5	6	7	NA
1	Learning to operate CoSTest would be easy for me	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	
2	I would find it easy to get CoSTest to do what I want it to do.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	
3	My interaction with CoSTest would be clear and understandable.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	
4	I would find CoSTest to be flexible to interact with.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	
5	It would be easy for me to become skilful at using CoSTest.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	
6	I would find CoSTest easy to use.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	

Figure C.8. User Acceptance Form: Perceived Ease-of-Use

