

Functional Size Measurement and Model Verification for Software Model-Driven Developments

A COSMIC-based Approach

Beatriz Marín Campusano



Advisors

Dr. Oscar Pastor López | Dr. Alain Abran



UNIVERSIDAD
POLITECNICA
DE VALENCIA



Doctoral Thesis

Functional Size Measurement and Model Verification for Software Model-Driven Developments: A *COSMIC-based Approach*

Beatriz Marín Campusano

Advisors: Dr. Oscar Pastor
Dr. Alain Abran

July 2011

Functional Size Measurement and Model Verification for Software Model-Driven Developments: A *COSMIC-based Approach*

This report was prepared by

Beatriz Marín Campusano

Advisors

Oscar Pastor López, Universitat Politècnica de València

Alan Abran, Université du Québec à Montréal

Members of the Thesis Committee:

Prof. Vicente Pelechano Ferragud, Universitat Politècnica de València

Prof. Juan José Cuadrado Gallego, Universidad Carlos III de Madrid

Prof. Antonio Vallecillo Moreno, Universidad de Málaga

Prof. Natalia Juristo Juzgado, Universidad Politécnica de Madrid

Prof. Pedro Valderas Aranda, Universitat Politècnica de València

Centro de Investigación en Métodos de Producción de Software (PROS)

Universitat Politècnica de València

Camino de Vera s/n, 46022 Valencia, Spain

Tel: +34-963877007 Ext. 83530

Fax: +34-963877359

Web: www.pros.upv.es

To Giovanni, Bianca, and Caterina

Acknowledgement

The development of this thesis has been an exciting challenge, with several discussions, meetings, and years of work. During this academic journey, I have been encouraged and accompanied by many people that I would like to thank.

First of all, I would like to thank my advisor and friend Oscar Pastor for his confidence, affection, support, and encouragement. He gave me the possibility to start researching, and throughout these years, he has always supported my ideas, giving me valuable assistance in every step done to face the challenging life of a researcher.

I would also like to thank my advisor Alain Abran for his assistance, confidence, and encouragement. All our meetings and e-mails have been very helpful to develop, improve, and communicate this work.

A special gratitude is to Vicente Pelechano for show me that the hard work had always the best reward and for his extremely quick feedback that I always have received.

I extend many thanks to Tanja Vos for her responsibility and enthusiasm in every work that we had performed. Tanja, you are an example of the work well done.

A special mention is for Juan Sánchez for his affection and generous advice.

Also, I would like to thank Vicky, Ana, Fani, Paqui, Nata, Marce, Mariajo, Pedro, J-Lu, Paco, Pau, Carlos, Ignacio, Sergio E., Arthur, and

Sergio S. for all the great moments that we have had inside and outside of the lab. Dear friends, thanks for your affection, friendly conversations, coffees, and dinners, which have been essential for the happy ending of this thesis.

I extend these thanks to all my colleagues at the PROS research center, and the administrative personnel and colleagues at DSIC for their friendliness and good vibrations at work, which make a really delight to work in this department.

I would like to thank my parents, my sisters and brothers, and my nephews and nieces for all their unconditional love and support throughout my life.

Finally, my special thanks to my beloved husband and colleague Giovanni for his love, understanding, and encouragement. Without him, this work would not have been possible. I would also thank to my babies Bianca and Caterina for their love, happiness, and for give me time to carry out this work.

Abstract

Historically, software production methods and tools have a unique goal: *to produce high quality software*. Since the goal of Model-Driven Development (MDD) methods is no different, MDD methods have emerged to take advantage of the benefits of using conceptual models to produce high quality software.

In such MDD contexts, conceptual models are used as input to automatically generate final applications. Thus, we advocate that there is a relation between the quality of the final software product and the quality of the models used to generate it. The quality of conceptual models can be influenced by many factors. In this thesis, we focus on the accuracy of the techniques used to predict the characteristics of the development process and the generated products.

In terms of the prediction techniques for software development processes, it is widely accepted that knowing the functional size of applications in order to successfully apply effort models and budget models is essential. In order to evaluate the quality of generated applications, defect detection is considered to be the most suitable technique.

The research goal of this thesis is to provide an accurate measurement procedure based on COSMIC for the automatic sizing of object-oriented OO-Method MDD applications. To achieve this research goal, it is necessary to accurately measure the conceptual models used in the generation of object-oriented applications. It is also very important for these models not to

have defects so that the applications to be measured are correctly represented.

In this thesis, we present the OOmCFP (OO-Method COSMIC Function Points) measurement procedure. This procedure makes a twofold contribution: the accurate measurement of object-oriented applications generated in MDD environments from the conceptual models involved, and the verification of conceptual models to allow the complete generation of correct final applications from the conceptual models involved.

The OOmCFP procedure has been systematically designed, applied, and automated. This measurement procedure has been validated to conform to the ISO 14143 standard, the metrology concepts defined in the ISO VIM, and the accuracy of the measurements obtained according to ISO 5725. This procedure has also been validated by performing empirical studies.

The results of the empirical studies demonstrate that OOmCFP can obtain accurate measures of the functional size of applications generated in MDD environments from the corresponding conceptual models. They also demonstrate that OOmCFP is useful in finding defects in conceptual models that are related to the consistency and the correctness of the conceptual model.

Resumen

Históricamente, los métodos y herramientas desarrollados para producir software han tenido un único objetivo: *producir software de alta calidad*. Dado que el objetivo de los métodos de Desarrollo de Software Dirigidos por Modelos (DSDM) no es diferente, estos métodos han surgido para tomar ventaja de los beneficios de usar modelos conceptuales para producir software de alta calidad.

En el contexto del DSDM, los modelos conceptuales son usados como entrada para generar automáticamente las aplicaciones finales. Por esto, creemos que existe una relación entre la calidad del producto de software generado y la calidad de los modelos utilizados para generarlo. La calidad de modelos conceptuales puede ser influenciada por muchos factores. En esta tesis, nos enfocamos en la exactitud de las técnicas utilizadas para predecir características del proceso de desarrollo y de los productos generados.

Respecto a las técnicas predictivas para los procesos de desarrollo de software, es ampliamente aceptado que el conocimiento del tamaño funcional de las aplicaciones es esencial para aplicar satisfactoriamente modelos de esfuerzo y presupuesto. Respecto a la evaluación de la calidad de las aplicaciones generadas, la detección de defectos es considerada como la técnica más apropiada.

El objetivo de investigación de esta tesis es proveer un procedimiento de medición exacto basado en COSMIC para la medición automática de

aplicaciones orientadas a objeto del método DSDM OO-Method. Para lograr este objetivo de investigación, es necesario medir exactamente los modelos conceptuales usados en la generación de esas aplicaciones orientadas a objeto. También es muy importante que esos modelos no tengan defectos para que las aplicaciones medidas sean correctamente representadas.

En esta tesis, presentamos el procedimiento de medición OOmCFP (Puntos de Función COSMIC para OO-Method). Este procedimiento hace una contribución doble: la medición exacta de aplicaciones orientadas a objeto generadas en entornos DSDM a partir de los modelos conceptuales involucrados, y la verificación de los modelos conceptuales para permitir la generación de aplicaciones finales correctas a partir de los modelos conceptuales involucrados.

El procedimiento OOmCFP ha sido sistemáticamente diseñado, aplicado y automatizado. Este procedimiento de medición ha sido validado respecto a su conformidad con el estándar ISO 14143, con los conceptos de metrología definidos en el estándar ISO VIM, y respecto a la exactitud de las medidas obtenidas de acuerdo al estándar ISO 5725. El procedimiento OOmCFP también ha sido validado mediante la realización de estudios empíricos.

Los resultados de los estudios empíricos demuestran que OOmCFP puede obtener medidas exactas del tamaño funcional de aplicaciones generadas en entornos DSDM a partir de los modelos conceptuales involucrados. Además, los resultados demuestran que OOmCFP es útil encontrando defectos en modelos conceptuales, los cuales están relacionados a la consistencia y la correctitud de los modelos conceptuales.

Resum

Històricament, els mètodes i eines desenvolupades per tal de produir programari han tingut un únic objectiu: *produir programari d'alta qualitat*. Atès que l'objectiu dels mètodes de Desenvolupament de Programari Dirigit per Models (DSDM) no és diferent, aquests mètodes han sorgit per aprofitar els beneficis d'usar models conceptuals per i així produir programari d'alta qualitat.

En el context del DSDM, els models conceptuals són usats com entrada per a generar automàticament les aplicacions finals. Així, nosaltres creiem que existeix una relació entre la qualitat del producte de programari generat i la qualitat dels models utilitzats per generar-lo. La qualitat dels models conceptuals pot estar influenciada per molts factors. En aquesta tesi, ens centrem en l'exactitud de les tècniques utilitzades per predir característiques del procés de desenvolupament i dels productes generats.

Respecte a les tècniques predictives per als processos de desenvolupament de programari, és àmpliament acceptat que el coneixement de la grandària funcional de les aplicacions és essencial per aplicar satisfactòriament models d'esforç i pressupost. Respecte a l'avaluació de la qualitat de les aplicacions generades, la detecció de defectes és considerada com la tècnica més apropiada.

L'objectiu d'investigació d'aquesta tesi és obtenir un procediment de mesura exacte basat en COSMIC per mesurar automàticament aplicacions

orientades a objecte del mètode DSDM OO-Method. Per aconseguir aquest objectiu d'investigació, és necessari mesurar exactament els models conceptuals usats en la generació de les aplicacions orientades a objecte. També és molt important per a eixos models no tindre defectes perquè les aplicacions mesurades estiguen correctament representades.

En esta tesi, presentem el procediment de mesura OOmCFP (Punts de Funció COSMIC per a OO-Method). Este procediment fa una contribució doble: la mesura exacta d'aplicacions orientades a objecte generades en entorns DSDM a partir dels models conceptuals involucrats, i la verificació dels models conceptuals per permetre la generació d'aplicacions finals correctes a partir dels models conceptuals involucrats.

El procediment OOmCFP ha estat sistemàticament dissenyat, aplicat i automatitzat. Aquest procediment de mesura ha estat validat respecte a la seua conformitat amb l'estàndard ISO 14143, amb els conceptes de metodologia definits en l'estàndard ISO VIM, i respecte a l'exactitud de les mesures obtingudes d'acord a l'estàndard ISO 5725. El procediment OOmCFP també ha estat validat mitjançant la realització d'estudis empírics.

Els resultats dels estudis empírics demostren que OOmCFP pot obtindre mesures exactes de la grandària funcional d'aplicacions generades en entorns DSDM a partir dels models conceptuals involucrats. Els resultats també demostren que OOmCFP és útil trobant defectes en models conceptuals, els quals estan relacionats a la consistència i la correctitud dels models conceptuals.

Table of Contents

Chapter 1 Introduction	1
1.1 Motivation	3
1.2 Problem Statement	5
1.3 Research Goal.....	9
1.4 Research Methodology.....	11
1.5 Context of the Thesis.....	15
1.6 Thesis Structure.....	16
Chapter 2 Foundations	21
2.1 The COSMIC Measurement Method	21
2.2 The OO-Method Approach: A MDD Method.....	24
2.2.1 The OO-Method Object Model	26
2.2.2 The OO-Method Dynamic Model	27
2.2.3 The OO-Method Functional Model.....	28
2.2.4 The OO-Method Presentation Model	29
2.3 A Software Measurement Process Model	32
2.4 Conclusions	33
Chapter 3 State of the Art	35
3.1 Functional Size Measurement Procedures based on COSMIC	36
3.1.1 Proposal of Bévo et al. (1999).....	36
3.1.2 Proposal of Jenner (2001)	37

3.1.3	Proposal of Diab et al. (2001)	38
3.1.4	Proposal of Poels (2002)	39
3.1.5	Proposal of Nagano et al. (2003).....	41
3.1.6	Proposal of Azzouz et al. (2004).....	42
3.1.7	Proposal of Condori-Fernández et al. (2004)	43
3.1.8	Proposal of Habela et al. (2005).....	44
3.1.9	Proposal of Grau et al. (2007)	45
3.1.10	Proposal of Levesque et al. (2008).....	46
3.1.11	General Analysis of Measurement Procedures	47
3.2	Approaches to Detect Defects in Conceptual Models	50
3.2.1	Proposal of Travassos et al. (1999)	51
3.2.2	Proposal of Laitenberger et al. (2000).....	52
3.2.3	Proposal of Conradi et al. (2003)	53
3.2.4	Proposal of Gomma et al. (2003)	54
3.2.5	Proposal of Kuzniarz et al. (2003)	55
3.2.6	Proposal of Berenbach (2004).....	56
3.2.7	Proposal of Lange et al. (2004)	57
3.2.8	Proposal of Leung et al. (2005).....	59
3.2.9	Proposal of Bellur et al. (2006)	61
3.2.10	Proposal of Egyed (2006).....	61
3.2.11	General Analysis of Defect Detection Proposals	63
3.3	Conclusions	64
Chapter 4 Design of a FSM Procedure.....		67
4.1	Definition of the Objectives	70
4.1.1	Purpose of the Measurement	71
4.1.2	Scope of the Measurement	72
4.1.3	Granularity Level of the Software	74
4.2	Characterization of the Concept.....	74
4.3	Selection of the Metamodel.....	75
4.3.1	Functional Users and Boundaries	77

4.3.2	Functional Processes	80
4.3.3	Data Groups.....	88
4.3.4	Data Attributes	89
4.3.5	Data Movements.....	89
4.4	Definition of the Numerical Assignment Rules	122
4.4.1	Measurement Function	122
4.4.2	Aggregation of Results.....	123
4.5	Conclusions	125
Chapter 5 Evaluation of the Design of OOmCFP		127
5.1	Conformity Evaluation of OOmCFP.....	129
5.2	Metrology Evaluation of the Design of OOmCFP	131
5.2.1	Measurement Foundation	133
5.2.2	Quantities and Units	135
5.2.3	Measurement Standards - Etalons	139
5.3	Precision Evaluation of OOmCFP.....	140
5.3.1	A Method for the Evaluation of Precision.....	143
5.3.2	A Pilot Study to Evaluate the Precision of OOmCFP	153
5.3.3	Lessons Learned from the Pilot Study.....	156
5.4	Conclusions	159
Chapter 6 Application of OOmCFP		161
6.1	Manual Application of OOmCFP	162
6.1.1	Software Documentation Gathering	163
6.1.2	Construction of the COSMIC Software Model	167
6.1.3	The Measurement Phase: Assignment of Numerical Rules ...	181
6.2	Automatic Application of OOmCFP	182
6.2.1	Architecture of OOmCFP Tool	182
6.2.2	Efficiency in the Measurement.....	184
6.2.3	Using the OOmCFP Tool	186
6.2.4	Verification of the OOmCFP Tool	191

6.3	Conclusions	193
Chapter 7 Evaluation of the Application of OOmCFP..... 195		
7.1	Metrology Evaluation of the Application of OOmCFP.....	196
7.1.1	Measurement Procedure	197
7.1.2	Devices for Measurement.....	199
7.1.3	Operations with Devices.....	200
7.1.4	Properties of Measuring Devices.....	202
7.2	Precision Evaluation of the Application of OOmCFP.....	202
7.2.1	The Definition Phase of the Experiment	204
7.2.2	The Measurement Phase of the Experiment.....	205
7.2.3	The Evaluation Phase of the Experiment	206
7.2.4	Validity of Experimental Results	209
7.3	Accuracy Evaluation of the Application of OOmCFP	211
7.4	Conclusions	213
Chapter 8 Defect Detection in MDD Conceptual Models..... 215		
8.1	A Metamodel for Defect Detection.....	218
8.2	Using OOmCFP to Detect Defects.....	223
8.3	Formalization of Defect Detection Rules.....	229
8.4	A Defect Detection Tool	234
8.4.1	The OOmCFP Tool	235
8.5	Conclusions	237
Chapter 9 Defect Detection Case Study..... 239		
9.1	Design of the Case Study	240
9.1.1	Case and Subjects Selection.....	241
9.1.2	Data Collection Procedure	244
9.1.3	Analysis Procedure.....	245
9.1.4	Validity Procedure	247
9.2	Results.....	250

9.2.1	Execution Description.....	251
9.2.2	Analysis and Interpretation Issues.....	252
9.3	Conclusions.....	265
Chapter 10	Conclusions.....	267
10.1	Contributions.....	268
10.2	Publications.....	270
10.3	Future Work.....	275
References	279
Appendix A	OOMCFP Measurement Guide.....	297
Appendix B	Conformity Evaluation Checklist.....	321
Appendix C	Number of Defects in the Case Study.....	327

List of Figures

Figure 1.1 Design Research Methodology.	11
Figure 1.2 First cycle of the methodology applied to this thesis.	12
Figure 1.3 Second cycle of the methodology applied to this thesis.	14
Figure 2.1 Phases and activities in COSMIC version 3.0	22
Figure 2.2 Models and transformations of the OO-Method approach.	24
Figure 2.3 The Jacquet and Abran software measurement process model... 33	
Figure 3.1 Main methods related to this thesis.	35
Figure 4.1 Design Process of the OOmCFP Measurement Procedure.	69
Figure 4.2 Pieces of software and layers of an OO-Method application.....	74
Figure 4.3 Metamodel of COSMIC version 3.0	76
Figure 4.4 Functional users and boundaries of an OO-Method application. 78	
Figure 4.5 Data movements between users and layers of an OO-Method application.	90
Figure 4.6 Data movements in a Display Set without derived attributes.	91
Figure 4.7 Data movements in a Display Set with derived attributes.	93
Figure 4.8 Data movements in a Filter.	95
Figure 4.9 Data movements in a Filter with variables that have default values specified.	97

Figure 4.10 Data movements in a Service with preconditions that are fulfilled.....	100
Figure 4.11 Data movements in a Service with preconditions that are not fulfilled.....	101
Figure 4.12 Data movements in a Service.....	103
Figure 4.13 Data movements in a Service with default values for its entry arguments.	107
Figure 4.14 Data movements in a Service of a class with integrity constraints that are fulfilled.	110
Figure 4.15 Data movements in a Service of a class with an integrity constraint that is not fulfilled.....	111
Figure 4.16 Data movements in a Service with a control condition.....	113
Figure 4.17 Data movements in a Service that is contained in a class with triggers specified.	115
Figure 4.18 Data movements in a Service with arguments that have defined dependency rules.	116
Figure 4.19 Data movements in a Service with a conditional navigation fulfilled and an initialization of an argument.	118
Figure 4.20 Data movements in a Service with a conditional navigation fulfilled and a navigational filtering.....	121
Figure 5.1 Conformity evaluation process for OOmCFP.....	130
Figure 5.2 High-level model of categories of metrology terms [Abran and Sellami 2002].	132
Figure 5.3 Levels of the Measurement Foundation.....	134
Figure 5.4 Measurement Foundations for OOmCFP.	135
Figure 5.5 High-level model of Quantity category.	135
Figure 5.6 High-level model of Measurement Standards - Etalons category.	139
Figure 5.7 Measurement of accuracy under ISO 5725.....	141
Figure 5.8 Method for evaluation of precision of software measures.	145

Figure 6.1 Application of OOmCFP Measurement Procedure.....	162
Figure 6.2 Object model of a rent-a-car system.	164
Figure 6.3 Example functional model of the Vehicle class.....	165
Figure 6.4 Dynamic model of the Client class of the rent-a-car system. ...	165
Figure 6.5 Menu (HAT) specified for the rent-a-car system.....	166
Figure 6.6. Attributes for the PIU_RentsDetails of the rent-a-car system..	166
Figure 6.7. Functional users, Pieces of software, layers, and boundaries of the rent-a-car system.	168
Figure 6.8. Data movements that occur in the rent-a-car system.	174
Figure 6.9. Analysis process in the OOmCFP tool.	183
Figure 6.10 Schema of the solution to avoid overflow problems.....	185
Figure 6.11 First Interface of OOmCFP tool.....	186
Figure 6.12 Second Interface of OOmCFP tool.	188
Figure 6.13 Third Interface of OOmCFP tool.....	188
Figure 6.14 Main page of the measurement report for the rent-a-car application.	189
Figure 6.15 Data movements in the elements contained in the functional processes of the rent-a-car application.	190
Figure 6.16 Data groups related to the data movements identified in the rent-a-car application.	190
Figure 7.1 High-level model of categories of metrology terms [Abran and Sellami 2002].	196
Figure 7.2 Topology of measurement procedure.	197
Figure 7.3 Topology of measurement instrument.	199
Figure 8.1 A metamodel for defects detection in conceptual models	222
Figure 8.2 Process of the defect detection tool.....	235
Figure 8.3 Screenshot of the tool and HTML defects report.....	237
Figure 9.1 Defects found by each inspector in Model2.....	253

Figure 9.2 Distinct defects found in Model4.....	254
Figure 9.3 Defect Types found by the inspection team in Model1	256
Figure 9.4 Defect Types found by the inspection teams	259
Figure 9.5 Defect Types found by the OOmCFP tool.....	262

List of Tables

Table 1. Defect Types presented by Laitenberger et al. (2000).....	53
Table 2. Defect Types presented by Gomma et al. (2003).	55
Table 3. Defect Types presented by Kuzniarz et al. (2003).....	56
Table 4. Defect Types presented by Berenbach (2004).....	58
Table 5. Defect Types presented by Lange et al. (2004 and 2006).	59
Table 6. Defect Types presented by Leung et al. (2005).....	60
Table 7. Defect Types presented by Bellur et al. (2006).	62
Table 8. Presentation elements contained in the interaction units.	83
Table 9. Quality criteria for the design of a measurement method.....	133
Table 10. Instantiation of ISO 5725 with software engineering concepts.	144
Table 11. Table to collect the results of the measures (adapted from ISO 5725).....	149
Table 12. Table to collect the arithmetic means of cells (adapted from ISO 5725).....	150
Table 13. Table for recording the spread of cells.	151
Table 14. Contained elements in PIU_Rental.....	170
Table 15. Contained elements in PIU_Vehicle.....	171
Table 16. Contained elements in PIU_Office.....	172
Table 17. Functional Processes, data groups, and data attributes of the rent-a-car application.	173
Table 18. Data movements that occur in PIU_Rental.....	176
Table 19. Data movements that occur in PIU_Vehicle.....	179

Table 20. Data movements that occur in PIU_Office.....	181
Table 21. Results obtained by the OOmCFP tool for 6 conceptual models of real projects.	192
Table 22. Models used to test the performance of the OOmCFP tool.....	193
Table 23. Quality criteria for the application of a measurement method according to metrology concepts.....	197
Table 24. Results of the measurement exercise.....	206
Table 25. Arithmetic means of cells.....	207
Table 26. Spread of cells.....	207
Table 27. Repeatability variance and reproducibility variance of each level.....	208
Table 28. Results obtained by the OOmCFP tool.....	212
Table 29. Defects related to mapping rules of OOmCFP.....	224
Table 30. Defects related to display and filter patterns OOmCFP rules....	225
Table 31. Rules to identify the data movements of OOmCFP.....	226
Table 32. Rules to identify the data movements of OOmCFP.....	227
Table 33. Rules to identify the data movements of OOmCFP.....	228
Table 34. 8 Defect Types of Conceptual Models found using OOmCFP and OCL Rules.....	230
Table 35. 9 Defect Types of Conceptual Models found using OOmCFP and OCL Rules.....	231
Table 36. 6 Defect Types of Conceptual Models found using OOmCFP and OCL Rules.....	232
Table 37. 5 Defect Types of Conceptual Models found in the literature and OCL Rules.....	233
Table 38. Distribution of subjects in the inspection teams for the five models.....	251
Table 39. Analysis of Defects found in Model4.....	255
Table 40. Description of Defect types related to the correctness and consistency found by the inspection teams.....	257

Table 41. Description of Defect types related to the completeness found by the inspection teams. 258
Table 42. Description of Defect types found by the OOmCFP tool. 260
Table 43. Defects found by the inspection teams (IT) and OOmCFP. 262
Table 44. Summary of publications related to this thesis. 275

Chapter 1

Introduction

Model-Driven Development (MDD) technologies attempt to separate the business logic from the platform technology in order to allow for the automatic generation of software through well-defined model transformations. In a software production process based on MDD technology, the conceptual models are the key artifacts that are used as input in the process of code generation. For this reason, the conceptual models must provide a holistic view of all the components of the final application (including the structure of the system, the behavior of the system, the interaction between the users and the system, and the interaction among the components of the system) in order to be able to automatically generate the final application.

In this context, we advocate that there is a relation between the quality of the final software product and the quality of the models used to generate it. Therefore, it would be possible to improve the quality of the software products that are generated using MDD technologies by evaluating and improving the quality of the conceptual models involved.

The quality of conceptual models is influenced by many factors, for instance, the suitability of the modeling language for the problem domain, the compliance of the tools used for modeling and performing the transformations over the modeling language, the convenience of the

modeling process with respect to the language and the tools used, the accuracy of the techniques applied to predict the characteristics of the development process and the generated products, etc. Thus, in order to evaluate the quality of conceptual models, it is necessary to select adequate techniques to control these factors. In this thesis, we focus on the accuracy of the techniques used to predict the characteristics of the development process and the generated products.

In terms of the prediction techniques for software development processes, it is widely accepted that knowing the functional size of applications is essential in order to successfully apply effort models (to predict the efforts that will be required in the project) and budget models (to predict the money that will be spent on the project) [Meli *et al.* 2000]. This knowledge can be used to generate indicators that facilitate the management of development projects.

To evaluate the quality of generated applications, defect detection is considered to be a suitable technique since it offers a high level of empirical validity that is provided by the variety of applications that are observed.

In a MDD context, where models are representing the different perspectives of the intended software applications, the functional size of the generated applications can be directly measured from the corresponding conceptual models. Thus, relevant indicators can be obtained in early stages of the development cycle.

Furthermore, since a functional size measurement procedure analyzes all the elements of the conceptual model that participate in the system functionality, the functional size procedure can be used to identify defects in the conceptual models. Thus, the aim of this work is to present a functional size measurement procedure that (1) allows the accurate measurement of final applications generated in MDD environments from their conceptual models, and (2) uses the analysis performed by the functional size measurement procedure to automatically detect defects in the conceptual models of the generated applications.

The rest of this chapter is organized as follows: Section 1.1 presents the reasons that inspire this thesis, and Section 1.2 presents the problem tackled in this work. Section 1.3 presents the goals of this research, and Section 1.4 presents the research methodology followed to achieve these goals. Section 1.5 presents the development context of this thesis, and finally, Section 1.6 presents the structure of the rest of the document.

1.1 Motivation

According to the 2011 Gartner study [Gartner 2011], global IT spending totaled \$3.4 trillion, up 5.4 percent from 2009 levels. However, although there is a continual increase in IT investments, the success of IT projects is still low. The latest CHAOS study published in 2010 [Standish_Group 2010] shows a decrease in project success rates, with only 32% of all projects being delivered on time, on budget, and with the required features and functions.

To achieve success in IT projects, the management and control of these kind of products is crucial. However, as Tom de Marco says [DeMarco 1982]: “you can not control what you can not measure”, and as Norman Fenton says [Fenton and Pfleeger 1996]: “you can not predict what you can not measure”. Taking into account these celebrated phrases, it is possible to state that the measurement of software products is crucial for the management and control of software products.

Software measurement can be performed from several different points of view, which can be grouped into two different approaches: *a priori* and *a posteriori* [Tran-Cao *et al.* 2002]. Since the *a posteriori* approaches (for instance, kilobytes of the product, code lines [Conte86], etc.) have a high dependency on the technological platform, these approaches have the disadvantage of providing measures too late and with little relevance for management. In contrast, since the *a priori* approaches (for instance,

functional size) are independent of the technological platform, these approaches are gaining more and more attention in the software measurement community because they allow the generation of indicators at early stages of the software development cycle.

Functional Size Measurement (FSM) was defined in the late 1970's mainly through the Function Point Analysis (FPA) proposal [Albrecht 1979]. Later, FPA was adapted by other proposals, from which five proposals have been recognized as standard measurement methods: IFPUG FPA [ISO/IEC 2003b], MK II FPA [ISO/IEC 2002], NESMA FPA [ISO/IEC 2005], COSMIC FFP [ISO/IEC 2003a], and FISMA [ISO/IEC 2008]. In general terms, these standards define a set of concepts and steps that must be taken into account to obtain the functional size of applications. Thus, the result of the application of a functional size measurement standard is a number that represents the amount of functionality of an application.

The functional size can be used to control the development of software products by calculation of indicators and comparison with the indicators of other software projects. Also, the functional size can be used in effort or budget models to predict these issues of the projects. In addition, since the process used to obtain the functional size takes into account all the concepts related to the functionality of an application, it is possible to exploit the measurement process in order to find defects that are related to these concepts.

In a MDD context [Mellor *et al.* 2003] [Selic 2003], where conceptual models are used as inputs in the generation process of the final application, the use of functional size measurement procedures can improve the quality of the generated applications by means of the improvement of the quality of the corresponding conceptual models. Thus, functional size measurement procedures can be used as suitable techniques to generate indicators of the conceptual models and to find defects in the conceptual models.

1.2 Problem Statement

The ISO/IEC 14143-1 [ISO 1998] standard defines *functional size* as the size of the software derived by quantifying the functional user requirements. This standard also defines a Functional Size Measurement (FSM) as the process of measuring the functional size. In addition, this standard defines a FSM method as the implementation of a FSM that is defined by a set of rules, which is defined in accordance with the mandatory features defined in the ISO/IEC 14143-1.

The measurement methods that fulfill the characteristics defined in the ISO/IEC 14143-1 [ISO 1998] and that have been verified with the ISO/IEC 14143-2 [ISO 2002] have been recognized as standards. These measurement methods are: IFPUG FPA [ISO/IEC 2003b], MK II FPA [ISO/IEC 2002], NESMA FPA [ISO/IEC 2005], COSMIC [ISO/IEC 2011], and FISMA [ISO/IEC 2008]. The first three methods are based on the Function Point Analysis proposal [Albrecht 1979], which takes into account only the functionality of the system that the human user observes.

Later, FPA has been adapted to object-oriented models aligned with the UML standard [Lehne 1997] [Tavares *et al.* 2002] [Uemura *et al.* 1999]. These FPA-based methods have several limitations for the correct measurement of systems: for instance, they only allow the measurement of the functionality that the human user sees, ignoring all the functionality that is needed for the correct operation of the application (which must be built by the developer, even though it is not seen by the human user); they only allow the measurement of Management Information Systems, which excludes the measurement of other types of software (such as real time software); and they do not consider the functionality that allows communication between layers in systems with a layer-based architecture.

Another significant limitation of FPA-based methods is that the size of any elementary process within a model is limited to only three intervals of classifications for DET (Data Element Types), RET (Record Element

Types), or FTR (File Types Referenced). These intervals assign a specific functional size for the elementary process in the complexity tables of the FSM method. Therefore, the functional size of a model will not vary even if some elementary processes have a very large number of DET, RET, or FTR [Giachetti *et al.* 2007; Kitchenham 1997]. Even though we found approaches that measure the functional size in MDD environments in the literature [Abrahão *et al.* 2006] [Abrahão *et al.* 2007], these approaches are FPA-based and also have the FPA limitations.

To overcome the limitations of the FPA-based measurement methods, the COSMIC measurement method [Abran *et al.* 2001] was defined in the late 1990's as the second generation of the functional size measurement method. It has been adopted as an international standard: ISO 19761 [ISO/IEC 2003a]. One of the advantages of COSMIC on FPA is that it allows the measurement of the functional size from different points of view, for instance, the functionality that the users see and the functionality that the developer has to build. Another advantage is that COSMIC uses a mathematical function that is not limited by maximum values to aggregate the functional size of the functional processes specified in the conceptual models. This helps to better distinguish the size of large conceptual models. Another advantage of COSMIC is that it allows the measurement of applications that are generated in layers, which allows the measurement of the whole application or the measurement of every layer of the application.

Currently, there are some measurement procedures that apply COSMIC to estimate the functional size of future software applications from models. A measurement procedure is defined by the International Vocabulary of Basic and General Terms in Metrology (VIM) [ISO 2004] as a “*set of operations, described specifically, used in the performance of particular measurements according to a given method of measurement*”. Some measurement procedures apply COSMIC to requirement models [Bévo *et al.* 1999] [Jenner 2001] [Condori-Fernández 2004] [Grau and Franch 2007b]. These proposals use scenarios, use-case diagrams, sequence diagrams, and i*

models to estimate the functional size. However, these models do not have enough semantic expressiveness to specify all the functionality of the involved systems (for instance, in these models it is not possible to specify the way in which the values of the attributes of a class changes); therefore, the functional size obtained by these proposals is not the accurate functional size of the final application.

There are other proposals that are designed to measure the functional size of conceptual models used in the automatic generation of final applications. These conceptual models provide more functional expressiveness than requirement models. This is the case of Diab's proposal [Diab *et al.* 2001] and Poels' proposal [Poels 2002].

Diab's proposal presents a measurement procedure to measure real-time applications modeled with the ROOM language [Selic *et al.* 1994]. Poels' proposal presents a measurement procedure to object-oriented applications of the domain of Management Information Systems (MIS) that are modeled with an event-based method named MERODE [Dedene and Snoeck 1994]. The main disadvantage of these two proposals is that the conceptual model does not allow the specification of all the functionality of the final application; for instance, that conceptual model does not allow the specification of the presentation of the application. Also, Poels' proposal is restricted to a specific technology because it uses the AndroMDA tool to specify the presentation of the application and to generate the final application. This generates a traceability problem between the generated application and the conceptual model. In addition, both proposals were defined using an old version of the COSMIC measurement method. Therefore, these proposals do not take into account the improvements made to the COSMIC measurement method, for instance, the capability to measure the functional size of a piece of software depending on the functionality that another piece of software requires. Other FSM procedures (based on COSMIC) for measuring the functional size of conceptual models can be found in the survey presented in [Marín *et al.* 2008].

In summary, none of the proposals for measurement procedures allows an accurate measurement of the functional size of MIS applications from the related conceptual model. Thus, the prediction of productivity, effort, budget, and other indicators using these proposals is very far from the real values. Moreover, none of them take advantage of the procedure used to obtain the functional size or the automation of the procedure in order to find defects in the models. In some cases, measurement procedures have been automated to obtain the functional size, but they do not take advantage of automation to improve the quality of the conceptual models. The main limitation of the approaches presented above comes from the lack of expressiveness of the conceptual models that are involved in the generation of the final application.

If the conceptual model has enough expressiveness to specify all the functionality of the final application, then a measurement procedure can accurately measure the functional size of the final application from its conceptual model. In addition, since a functional size measurement procedure analyzes the conceptual models in order to identify and count the elements that represent the functionality of the final applications, the measurement procedure can be used to detect the defects that the models may have.

In order to solve the problem statement, this thesis proposes a functional size measurement procedure based on COSMIC for the measurement of the conceptual models of a specific MDD environment called OO-Method.

The OO-Method approach is an object-oriented method that puts the MDD technology into practice [Pastor *et al.* 2001], separating the business logic from the platform technology, allowing the automatic generation of final applications by means of well-defined model transformations [Pastor and Molina 2007]. It provides the semantic formalization needed to define complete and unambiguous conceptual models, thereby allowing the specification of all the functionality of the final application at the conceptual level. This method has been implemented in an industrial tool [CARE-

Technologies 2011] that allows the automatic generation of fully working applications. The applications generated can be desktop or web MIS applications and can be generated in several technologies (for instance, java, C#, visual basic, etc.).

Therefore, the measurement procedure proposed in this thesis performs an accurate measurement of the functional size of final applications from the corresponding conceptual models and is also used to detect defects that the conceptual models may have.

1.3 Research Goal

The research goal of this thesis is **to provide an accurate measurement procedure based on COSMIC for the automatic sizing of object-oriented OO-Method MDD applications**. To reach this research goal, it is necessary to accurately size the conceptual models used in the generation of object-oriented applications, and it is also very important for these models not to have defects in order to correctly represent the applications to be measured. Thus, our research goal will be satisfied by dealing with and achieving the following sub-goals:

- To provide an accurate measurement procedure based on COSMIC for the automatic sizing of OO-Method conceptual models.
- To ensure that OO-Method conceptual models are defect-free models to allow for the generation and the measurement of object-oriented applications.

The first sub-goal will be reached by satisfying the following research objectives:

- To design a functional size measurement procedure based on COSMIC for the measurement of OO-Method applications from

their conceptual models. This design includes a set of mapping rules between the concepts of COSMIC and OO-Method, and the definition of a set of rules that allow the quantification of the functional size of OO-Method conceptual models.

- To apply the measurement procedure. This application includes the definition of a procedure that guides the application of the measurement procedure and the definition of an example to apply the measurement procedure.
- To verify the accuracy of the measurement procedure. This verification will be performed using standards, metrology concepts, and controlled experiments.
- To automate the measurement of the functional size of OO-Method conceptual models. This automation includes an analysis of the existing tools, the implementation of a measurement tool, and the verification of the implemented tool.

To reach the second sub-goal, the following research objectives must be satisfied:

- To study the defects types that can be identified using the functional size measurement procedure proposed. This study includes the definition of a list of defect types and their classification.
- To verify the use of the functional size measurement procedure to identify defects in the conceptual models. This verification includes the measurement of conceptual models that have defects and conceptual models that do not have defects.
- To automate the detection of defects in conceptual models. This automation includes an analysis of the existing tools, the implementation of a defect detection tool, and the verification of the implemented tool.

1.4 Research Methodology

To perform the work of this thesis, the design methodology for performing research in information systems as described by [March and Smith 1995] and [Vaishnavi and Kuechler 2007] has been selected. *Design Research* involves the analysis and performance of designed artifacts to understand, explain, and, very frequently, to improve the behavior of aspects of Information Systems. The *Design Research Methodology* is comprised of five phases (see Figure 1.1): Awareness of Problem, Suggestion, Development, Evaluation, and Conclusion.

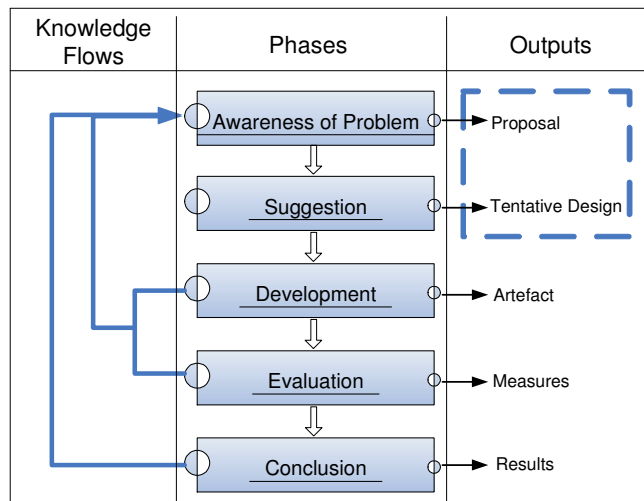


Figure 1.1 Design Research Methodology.

The awareness of an interesting problem may come from academia or industry. The output of this first phase is a proposal for a new research effort. The suggestion phase is a creative step that has a tentative design for the proposal as result. The tentative design is implemented in the

development phase. The output of this phase corresponds to an artefact, which can vary depending on the tentative design to be constructed. In the evaluation phase, the artefact is evaluated according to the criteria that are in the proposal. The measurements of this phase, which are joined to additional information gained in the construction of the artefact, can deliver feedback to another design suggestion. Finally, the conclusion phase is the final step of the research effort. The output of this phase is the result of the research effort and the knowledge gained.

Following the cycle defined in the Design Research Methodology (see Figure 1.2), we started with the *Awareness of Problem*. By performing systematic reviews of the literature, we have found that none of the proposals for functional size measurement procedures allows the accurate measurement of the functional size of MIS applications in the conceptual models. Therefore, in this thesis, we propose the definition of a measurement procedure that can obtain accurate functional size measures of the final software products from the corresponding conceptual models.

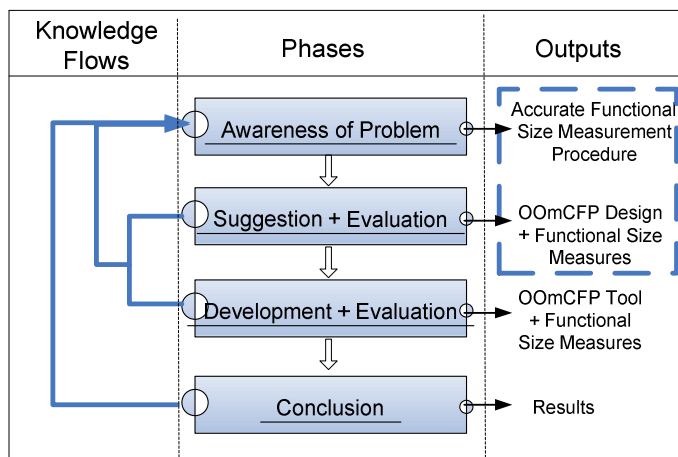


Figure 1.2 First cycle of the methodology applied to this thesis.

Next, we performed the second phase of the methodology which corresponds to the *Suggestion*. In this phase, the design of the functional size measurement procedure has been performed. This procedure is called OOmCFP. The OOmCFP procedure has been systematically designed to obtain accurate measurement results of the application that is generated from the conceptual models. In this phase, the application of the OOmCFP procedure to different conceptual models has also been performed.

At this point, we decided that it is very important to evaluate the design of the measurement procedure before the implementation in order to prevent the propagation of design defects. Therefore, we proceeded with the *Evaluation* phase. To verify the accuracy of the functional size measurement results obtained by the procedure designed (OOmCFP), we performed the evaluation phase in accordance with the concepts defined for functional size methods, the definition of metrology concepts, the accuracy of the measurement results obtained, and some empirical studies. The results of the evaluation have been used to correct the design of the OOmCFP measurement procedure.

Once the solution to the problem was designed, we performed the *Development* phase. In this phase, we developed a tool that automates the application of the OOmCFP measurement procedure. Then, we performed the *Evaluation* phase again. In this phase, we verified the OOmCFP tool empirically. To do this, we used several conceptual models (student conceptual models and industrial conceptual models). The results of this phase have been used to correct the OOmCFP tool.

Finally, we performed the *Conclusion* phase. In this phase, we analyzed the results of our research work. As the main result, we have designed a measurement procedure that can obtain accurate measures of the functional size of object-oriented applications from the corresponding conceptual models, and we have also automated the application of this procedure.

Even though the measurement procedure is essential to be able to obtain accurate measures, other factors can also influence the measurement: for

instance, the use of models that do not completely specify the final application. Thus, given that a functional size measurement procedure analyzes all the elements of the conceptual model that fulfill the functional user requirements, the measurement procedure can be used as a very valuable tool to identify defects in the conceptual models. Therefore, in this thesis we initiated a second cycle of the design research methodology (see Figure 1.3).

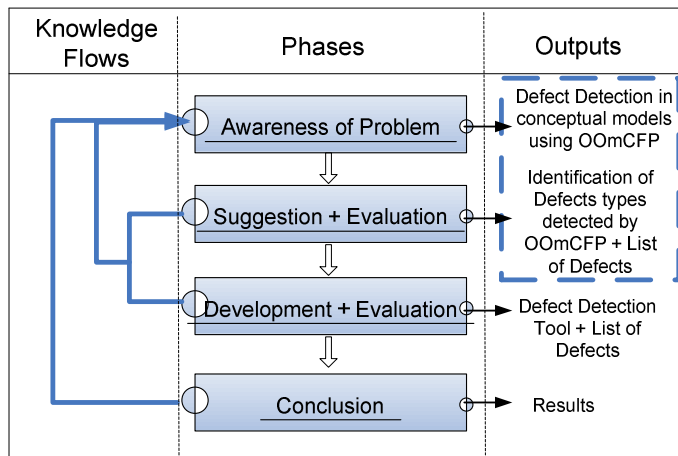


Figure 1.3 Second cycle of the methodology applied to this thesis.

In the *Awareness of Problem* phase of the second cycle of the methodology, we found that none of the measurement procedures take advantage of their design and automation to identify defects. Therefore, in this thesis, we also propose that a functional size measurement procedure can be used to detect defects in the conceptual models.

In the *Suggestion* phase, we identified what kind of defects can be detected in the conceptual models using the OOmCFP functional size measurement procedure. As in the first cycle of the application of the Design Research Methodology, we performed the *Evaluation* phase to verify that the OOmCFP procedure found defects in erroneous conceptual models.

Then, we performed the *Development* phase. In this phase, we updated the OOmCFP tool in order to deliver (1) a set of defects when the conceptual models have defects, or (2) the functional size when the conceptual models have a high quality (complete and correct). Next, we performed the *Evaluation* phase of this second cycle of the Design Research Methodology. In this phase, we verified the OOmCFP tool with erroneous conceptual models as well as with high-quality conceptual models.

At the end of the second cycle of the Design Research Methodology, we performed the *Conclusion* phase. In this phase, we concluded the research effort of this thesis. However, taking into account the evolving nature of the design research methodology, many areas for further research can emerge, such as testing of conceptual models. Therefore, we have also delimited the areas for further research in this phase.

1.5 Context of the Thesis

This thesis was developed at the Software Production Methods Research Center (ProS) at the Universidad Politécnica de Valencia.

The current working lines in which the ProS Research Center is involved are mainly focused on the following aspects:

- Model Driven Development and Code Generation
- Organizational Modelling and Requirements Engineering
- Development of Ambient Intelligence Systems
- Human-Computer Interaction and Usability
- Web Engineering
- Conceptual Modelling and Development of Services and Web Applications
- Empirical Software Engineering
- Software Quality and Automated Testing

- Genome Conceptual Modelling

The work presented in this thesis has been developed combining the following research lines: Model-Driven Development and Code Generation, Empirical Software Engineering, and Software Quality and Automated Testing.

This work has been made possible thanks to the following enterprises and R&D Spain and European projects:

- “CONCOM: Construcción de Compiladores” CARE-Technologies Project referenced as UPV 20060745.
- “SESAMO: Construcción de Servicios de Software a partir de Modelos” CICYT Project referenced as TIN2007-62894.
- “OSAMI Commons: Open Source Ambient Intelligence Commons”. ITEA 2 project referenced as TSI-020400-2008-114.
- “PISA: Producción de Software en Ambientes MDA” CARE-Technologies Project referenced as UPV 20080250.
- “MYMOBILEWEB: Tecnologías Avanzadas para el Acceso Movil, Independiente de Dispositivo e Inteligente (Guiado por Semántica) a Aplicaciones, Servicios y Portales de Información” ITEA 2 Project referenced as TSI-020400-2010-118
- “Verificación de Proceso y Validación de Producto Software de Equipos Digitales Simples relacionados con la Seguridad para Centrales Nucleares” IBERDROLA Project referenced as UPV 20100652.
- “FITTEST: Future Internet Testing”. FP7 project referenced as FP7-ICT-2009-5 / INFOS-ICT-257574.

1.6 Thesis Structure

The remainder of the thesis is organized as follows:

Chapter 2: Foundations

This chapter presents the methods that we use as the foundations of this thesis. Initially, we present the COSMIC Functional Size Measurement Method. Then, we introduce the OO-Method approach. Since this thesis uses the conceptual model of the OO-Method approach, we explain in detail all the conceptual constructs that make up this model. Next, we present a process model that we follow for the systematic specification of the measurement procedure.

Chapter 3: State of the Art

This chapter presents an analysis of the works related to this thesis. First of all, we analyze several functional size measurement procedures that have been defined to measure conceptual models in accordance with COSMIC. Next, we present a set of proposals that have been defined to detect defects in conceptual models.

Chapter 4: Design of a FSM Procedure

This chapter presents the design of the OOmCFP (OO-Method COSMIC Function Points) measurement procedure. The design of OOmCFP has been systematically carried out in order to obtain accurate measurement results, which is the most important contribution of this thesis. The design of the OOmCFP procedure contains a set of rules to identify the conceptual constructs that contribute to the functional size of software applications. Also, the design contains a set of rules to calculate the functional size of software applications from their conceptual models.

Chapter 5: Evaluation of the Design of OOmCFP

This chapter presents the evaluation of the design of the OOmCFP FSM procedure. This evaluation has been carried out by using the ISO/IEC 14143-2 standard for the conformity evaluation of software size measurement methods, the International vocabulary of basic and general

terms in metrology (VIM), and the precision of the measurement results obtained by OOmCFP. To evaluate the precision, a method based on the ISO 5725-2 for the Accuracy (trueness and precision) of measurement methods and results has been defined, and a pilot study has been carried out. This method is also an important contribution of this thesis.

Chapter 6: Application of OOmCFP

This chapter presents a process for the application of the OOmCFP measurement procedure and it also presents an example of the manual application of OOmCFP. In addition, this chapter presents a strategy for automatically applying OOmCFP to OO-Method conceptual schemas. This strategy has been implemented in a tool to perform the measurement of the functional size efficiently. This is another important contribution of this thesis. Finally, the application of the OOmCFP tool to real projects is presented.

Chapter 7: Evaluation of the Application of OOmCFP

This chapter presents the evaluation of the application of OOmCFP. This validation has been carried out using the International Vocabulary of Basic and General Terms in Metrology (VIM), performing a laboratory experiment to evaluate the precision using the method defined in Chapter 5, and performing a comparison of the results obtained by the OOmCFP tool with results obtained by experts.

Chapter 8: Defect Detection in MDD Conceptual Models

This chapter introduces how a functional size measurement procedure can be used to find defects in conceptual models. A set of defect types that can be identified in the conceptual models using OOmCFP is defined. Then, the conceptual constructs involved in the defect types are formalized by means of a metamodel and a set of OCL rules. Next, a strategy to automatically apply OOmCFP to detect defects is presented.

Chapter 9: Defect Detection Case Study

This chapter presents a case study that has been carried out to evaluate the usefulness of the application of a functional size measurement procedure to detect defects. The results indicate that the FSM is useful since it finds all the defects related to a specific defect type, it finds different defect types than an inspection team; and it finds defects related to the correctness and the consistency of the models.

Chapter 10: Conclusions

This chapter concludes this work by presenting the contributions of this thesis, the publications that this work has originated, and future works.

1. Introduction

Chapter 2

Foundations

The COSMIC [ISO/IEC 2003a] measurement method allows the measurement of functional size by means of the identification of data movements that occur between functional users and functional processes (and vice versa). Since the objective of this thesis is to present a measurement procedure based on the COSMIC measurement method, this chapter presents a brief explanation of the COSMIC measurement method, a brief explanation of the OO-Method MDD approach, and a process model for the definition of software measurement procedures.

2.1 The COSMIC Measurement Method

In 1997, St-Pierre et al. [St-Pierre *et al.* 1997] define the Full Function Points (FFP) method for the measurement of control systems, real-time systems, and embedded systems. This measurement procedure obtains the functional size of an application by means of the identification of functional processes and data movements that occur in these processes. The data movements can be *Entry* to the functional process, *Exit* from the functional process, *Read* data from persistent storage, and *Write* to persistent storage.

When all the data movements have been identified, they are aggregated to obtain the functional size of each functional process.

Later, in 1999 the Common Software Measurement International Consortium (COSMIC) publishes the 2.0 version of the COSMIC-FFP measurement method [Abran *et al.* 1999]. Since some modifications were performed to the 2.0 version of COSMIC-FFP, the 2.1 version was published in 2001 [Abran *et al.* 2001] and the 2.2 version was published in 2003 [Abran *et al.* 2003]. This last version was recognized as the ISO/IEC 19761 standard [ISO/IEC 2003a]. In 2007, a new version of COSMIC was published [Abran *et al.* 2007]. This last version aggregates a phase for the definition of the measurement strategy. Therefore, the last version of the COSMIC measurement method [Abran *et al.* 2007] includes three phases: the measurement strategy, the mapping of concepts, and the measurements of the identified concepts (see Figure 2.1).

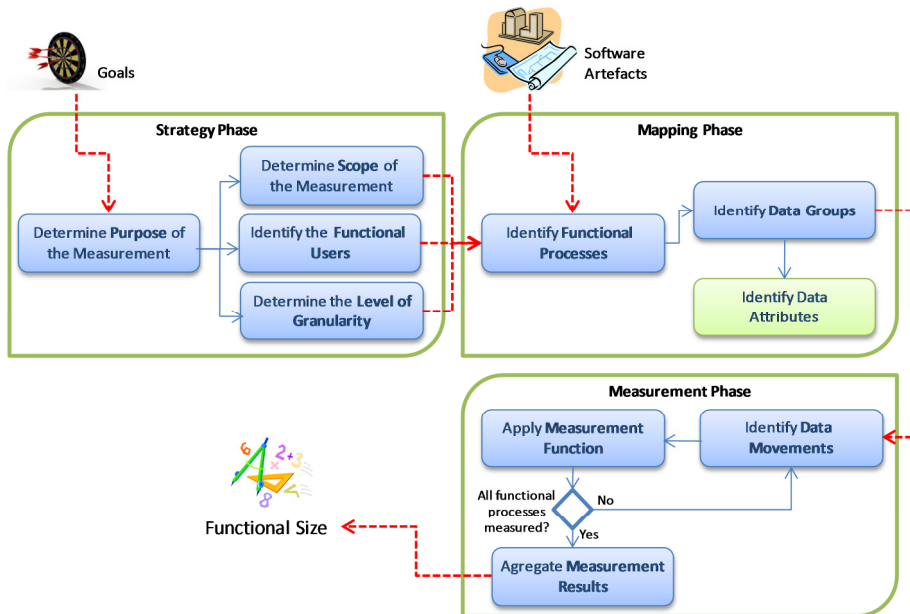


Figure 2.1 Phases and activities in COSMIC version 3.0

In the *strategy phase*, the purpose of the measurement exercise must be defined to explain why it is necessary to measure and what the measurement result will be used for. Next, the scope of the measurement must be defined in order to select the set of user functional requirements that will be included in the measurement task. Then, the functional users of the application to be measured must be identified. The functional users are the types of users that send (or receive) data to (from) the functional processes of a piece of software. This phase also includes the identification of the boundary, which is a conceptual interface between the functional user and the piece of software that will be measured. Finally, the level of granularity of the description of a piece of software to be measured is identified.

In the *mapping phase*, the functional processes must be identified (i.e., the elementary components of a set of functional user requirements). Every functional process is triggered by a data movement from the functional user, and the functional process is completed when it has executed all the data movements required for the triggering event. It should be kept in mind that a triggering event is an event that causes a functional user of the piece of software to initiate one or more functional processes. Next, the data groups must be identified. A data group is a set of data attributes that are distinct, non empty, non ordered, non redundant, and that participates in a functional process. Finally, the identification of the data attributes, which comprise the smallest part of information of a data group, is optional.

In the *measurement phase*, the data movements (Entry, Exit, Read and Write) for every functional process must be identified. When all the data movements of the functional process are identified, the measurement function for the functional process must be applied. This is a mathematical function that assigns 1 CFP (Cosmic Function Point) to each data movement of the functional process. Then, after all the functional processes are measured, the measurement results are aggregated to obtain the functional size of the piece of software that has been measured.

2.2 The OO-Method Approach: A MDD Method

The OO-Method approach is an object-oriented method that separates the business logic from the platform technology in order to allow the automatic generation of final applications by means of well-defined model transformations [Pastor *et al.* 2001]. This MDD approach allows the generation of applications that correspond to the domain of Management Information System (MIS).

The OO-Method approach puts the MDD technology into practice [Pastor and Molina 2007]. Its software production process is comprised of four models (see Figure 2.2): the *Requirements Model*, the *Conceptual Model*, the *Execution Model*, and the *Implementation Model*. These models have a direct correspondence with the models of the MDA architecture: the Computation-Independent Model (CIM), the Platform-Independent Model (PIM), the Platform-Specific Model (PSM), and the Implementation Model (IM).

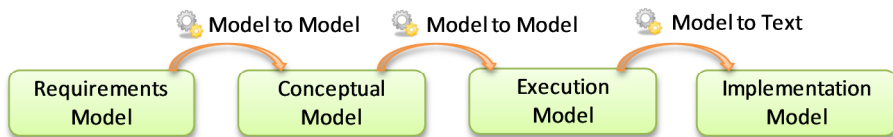


Figure 2.2 Models and transformations of the OO-Method approach.

In the Requirements Model [Diaz *et al.* 2005] [Insfrán *et al.* 2002], the systems analyst specifies the requirements of the system describing the mission statement, a function refinement tree, and a set of corresponding use-case diagrams. Then, the use-case diagrams are used to semi-automatically generate sequence diagrams.

From the Requirements Model, it is possible to semi-automatically generate the basis of the Conceptual Model [Pastor and Molina 2007]. The Conceptual Model captures the static and the dynamic part of a system by

means of an object model, a dynamic model, and a functional model. The Conceptual Model also captures the presentation of a system and the interaction with the system, allowing the specification of the graphical user interface in an abstract way by means of a presentation model.

From the Conceptual Model, it is possible to automatically generate the corresponding software product by applying a predefined Execution Model [Gómez *et al.* 1998]. The Execution model is based on the transformation of the basic building units of the Conceptual Model (which are defined in a formal language named OASIS [Pastor *et al.* 1992]) into their associated software representations. In fact, a Conceptual Model Compiler is responsible for executing that task. The OASIS language [Pastor *et al.* 1992] is used to specify the basic conceptual constructs used for building conceptual schemas (i.e. the preconditions of services, the derivation formula for the derived attributes, the body of the services, the integrity constraints for classes, the valid state transitions, the filters formula for the presentation, etc.).

Finally, the OO-Method Model Compiler generates applications in a three-tier architecture: one tier for the client component, which contains the graphical user interface; one tier for the server component, which contains the business rules and the connections to the database; and one tier for the database component, which contains the persistence aspects of the applications. An industrial implementation has been developed by the enterprise CARE Technologies [CARE-Technologies 2011]. This has also guided our selection of tools in order to apply our ideas in an industrial Model Compiler. The OO-Method Model Compiler allows the generation of applications in several technologies, such as JSP, ASP, C#, EJB, VB, SQL, ORACLE, DB2, etc., depending on the software architecture selected by the user when performing the model compilation process.

In this work, we need only focus on the Conceptual Model, which has the artefacts that are required to measure the functional size of the final application through the corresponding measurement process. For a better

understanding of the Conceptual Model, the OO-Method constructs specified in the object model, the dynamic model, the functional model, and the presentation model are briefly described in the following paragraphs.

2.2.1 The OO-Method Object Model

The object model of the OO-Method approach describes the static part of the system. This model allows the specification of classes, attributes, derived attributes, events, transactions, operations, preconditions, integrity constraints, agents, and relationships between classes. The main concepts of the object model are well-known because they are the same as those used in the UML class diagram [OMG 2010].

The main conceptual construct of the object model is the class. A class describes a set of objects that share the same specifications of characteristics, constraints, and semantics. A class can have attributes, services, integrity constraints and relationships with other classes.

The attributes of a class represents characteristics of this class. The attributes of a class can also be derived attributes, which obtain their value from the values of other attributes or constants.

The services of a class are basic components that are associated with the specification of the behavior of a class. The services can be events, transactions or operations. The events are atomic services, indivisible, which can assign a value to an attribute. The transactions are a sequence of events or other transactions that have two ways to end the execution: either all involved services are correctly executed, or none of the services are executed. Finally, the operations are a sequence of events, transitions or other operations, which are executed sequentially independently of whether or not the involved services have been executed correctly. The services can have preconditions that limit its execution because the preconditions are conditions that must be true for the execution of a service, which can be an event, a transaction or an operation

The classes also can have integrity constraints, which are expressions of a semantic condition that must be preserved in every valid state of an object.

The agents are active classes that can access specific attributes of the classes of the model and that can execute specific services of the classes of the model.

Finally, the relationships between classes can be agent relationships (that represent which object can activate which services); association, aggregation, and composition relationships; and specialization relationships.

2.2.2 The OO-Method Dynamic Model

The OO-Method dynamic model is comprised of two diagrams: the state transition diagram and the object interaction diagram.

The state transition diagram defines the valid lives of the objects that belong to a class. This diagram has the following conceptual constructs: initial state, final state, intermediate states, and transitions. Most of the concepts of this diagram are the same as those used in the UML state transition diagram [OMG 2010].

The initial state represents the state that objects are in immediately before they are created. The final state represents the state that objects are in immediately after they are destroyed. And, the intermediate states represent different situations that an object of a class may find itself in at any point during its life. The intermediate states have incoming and outgoing transitions, which represent a change of the state of an object. The transitions are activated by an agent that executes a service and can also have a condition to execute the service when it is required.

The object interaction diagram defines the interactions among the objects of the system. To do this, the triggers of the classes of the system and the global transactions or operations of the system are defined. The triggers are defined in a specific class. Each trigger is composed by a trigger condition and a service to be executed after a successful execution of a service that

activates the trigger. The triggers can be executed for: (1) the same object that has been used to activate the trigger, (2) a particular object of the class, (3) several objects of the class, or (4) all the objects of the class. Each trigger service is executed in the background in a transparent way for the user, who does not know the result of the execution (either success or failure).

The global transactions and operations are sequences of services, like the transactions and operations of the object model. The global services can involve services of any class of the system. Usually, these services are defined when it is necessary to execute services of objects that are not related.

2.2.3 The OO-Method Functional Model

The functional model of the OO-Method approach allows the specification of the effects that the execution of an event has over the value of the attributes of the class that owns the event.

The functional model uses *valuations* to assign values to the corresponding attributes. The valuations can have preconditions. These preconditions and the effect of the valuation must be specified by means of well-formed, first-order logic formulae that are defined using the OASIS language.

The change that a valuation produces in the value of an attribute is classified into three different categories: *state*, *cardinal*, and *situation*. The state category implies that the change of the value of an attribute depends only on the effect specified in the valuation for the event, and it does not depend on the value in the previous state. The cardinal category increases, decreases or initializes the numeric-type attributes. The situation category implies that the valuation effect is applied only if the value of the attribute is equal to a predefined value specified as *current value* of the attribute.

2.2.4 The OO-Method Presentation Model

In order to specify the interaction between the users of an application and the system, the OO-Method approach allows the specification of views in the object model. A *view* corresponds to a set of interfaces, which are the communication point between agents and classes of the OO-Method object model. When the views of a system have been defined, the interaction model of each view must be specified.

The presentation model allows the specification of the graphical user interface of an application in an abstract way [Molina 2003]. To do this, the presentation model has a set of abstract presentation patterns that are organized hierarchically in three levels: access structure, interaction units, and auxiliary patterns.

The first level allows the specification of the system access structure. In this level, the set of entry options that each user of the application will have available is specified by means of a *Hierarchy Action Tree* (HAT).

Based on the menu-like view provided by the first level, the second level allows the specification of the interaction units of the system. The interaction units are groups of functionality that allow the users of the application to interact with the system. Thus, the interaction units of the presentation model represent entry-points for the application, and they can be:

- A *Service Interaction Unit* (SIU). This interaction unit represents the interaction between a user of the application and the execution of a system service. In other words, the SIUs allow the users of the application to enter the values for the arguments of a service, to execute the service, and to offer to the users the feedback of the results of the execution of the service.
- A *Population Interaction Unit* (PIU). This interaction unit represents the interaction with the system that deals with the presentation of a set of instances of a class. In a PIU, an instance can be selected, and the

corresponding set of actions and/or navigations for the selected instance are offered to the user.

- An *Instance Interaction Unit* (IIU). This interaction unit represents the interaction with an object of the system. In an IIU, as well as in a PIU, the corresponding set of actions and/or navigations for the instance are offered to the user.
- A *Master Detail Interaction Unit* (MDIU). This interaction unit represents the interaction with the system through a composite interaction unit. A MDIU corresponds to the joining of a master interaction unit (which can be an IIU or a PIU) and a detail interaction unit (which can be a set of IIUs, PIUs, or SIUs).

The third level of the presentation model allows the specification of the auxiliary patterns that characterize lower level details about the behaviour of the interaction units. These auxiliary patterns are:

- The *entry* pattern, which is used to indicate that the user can enter values for the arguments of the SIUs.
- The *defined selection* pattern, which is used to specify a list of specific values to be selected by the user.
- The *introduction pattern*, which is used to define masks for the introduction of values by the user.
- The *arguments grouping* pattern, which is used to group a set of arguments of the SIUs in order to facilitate the user interaction with the system.
- The *arguments dependency* pattern, which is used to define dependencies among the values of the arguments of a service. To do this, Event-Condition-Action (ECA) rules are defined for each argument of the service. The ECA rules have the following semantics: when an interface event occurs in an argument of a service (i.e., the user enters a value), an action is performed if a given condition is satisfied.

- The *arguments pre-charge* pattern, which is used to define a set of objects that can be selected as arguments of a SIU.
- The *order criteria* pattern, which allows the objects of a PIU to be ordered. This pattern consists of the ascendant/descendant order over the values of the attributes of the objects presented in the PIU.
- The *display set* pattern, which is used to specify which attributes of a class or its related classes will be shown to the user in a PIU or an IIU.
- The *navigation* pattern, which allows the navigation from an interaction unit to another interaction unit.
- The *action* pattern, which allows the execution of services by joining and activating the corresponding SIUs by means of actions.
- The *filter pattern*, which allows a restricted search of objects for a population interaction unit. A filter can have data-valued variables and object-valued variables. These variables can have a default value defined, a PIU related to select the value of the object-valued variables, and pre-charge capabilities for the values of the object-valued arguments.
- The *navigational filtering* pattern, which allows the navigation to related objects to be restricted using a filter condition.
- The *initialization of arguments* pattern, which allows the argument of a SIU to be initialized when it is accessed directly from another SIU.
- The *conditional navigation* pattern, which allows the navigation from a SIU to other interaction units, depending on the execution result (success or failure) of the SIU.

Each auxiliary pattern has its own scope that states the context in which it can be applied. Taking into account the characteristics of the presentation model of the OO-Method approach, it is possible to completely specify the abstract graphical user interface of applications that correspond to the OO-

Method domain (MIS applications). Then, the Model Compiler transforms the presentation model into the corresponding concrete user interface to characterize those parts of the final software product that represent the user interaction.

2.3 A Software Measurement Process Model

The process model for software measurement proposed by Jacquet *et al.* [Jacquet and Abran 1997] has been used in the development of other functional size measurement procedures, such as [Abrahão *et al.* 2006], [Abrahão *et al.* 2007], [Condori-Fernández *et al.* 2007], [Grau and Franch 2007b]. This process model is comprised of four steps (see Figure 2.3): design of a measurement method, application of the measurement method rules, measurement results analysis, and exploitation of the measurement results.

In the first step, the concept to be measured and the rules to measure this concept must be defined. This step is divided into four sub-steps to complete the design of the measurement method: definition of the objectives, characterization of the concept to be measured, selection of the metamodel, and definition of the numerical assignment rules.

In the second step, the designed measurement method must be applied to the software or a piece of the software. This step is divided into three sub-steps: software documentation gathering, construction of the software model, and assignment of the numerical rules.

In the third step, the measurement result obtained in the second step must be documented and audited. This step is divided into two sub-steps: presentation of the measurement results and the audit of the results. In the fourth step, the results obtained from the application of the method are used in different ways, for instance in quality models, in estimation processes, in budget models, in productivity models, etc.

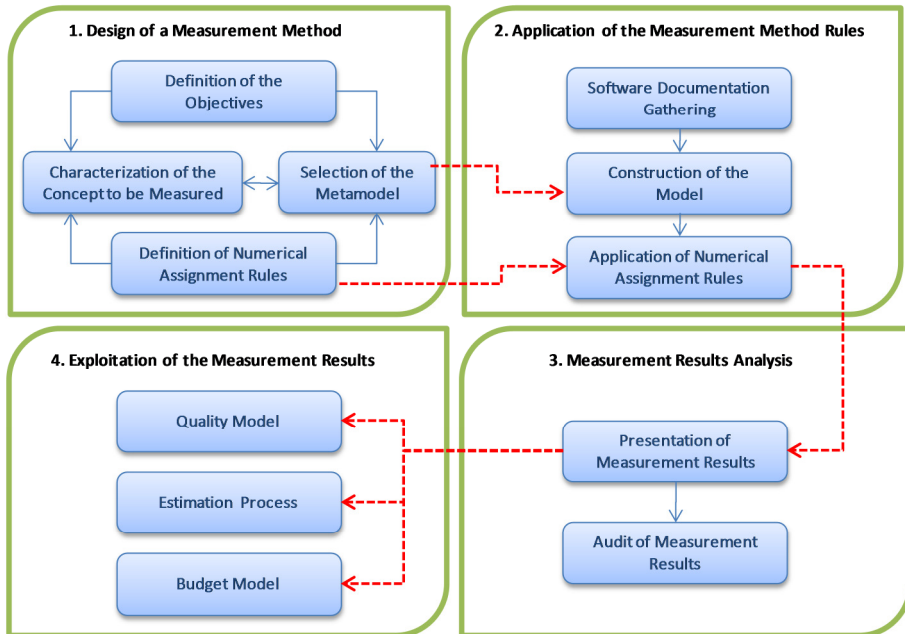


Figure 2.3 The Jacquet and Abran software measurement process model.

2.4 Conclusions

In this chapter, we have presented the methods and process that were selected to perform the thesis work: the COSMIC Functional Size measurement method, the OO-Method Model-Driven Development method, and the process defined by Jacquet *et al.* to develop measurement procedures. These methods are the basis for the development of the measurement procedure that is presented in the following chapters.

Chapter 3

State of the Art

This work deals with Functional Size Measurement (FSM) and defect detection of models used to generate applications in Model-Driven Development environments (see Figure 3.1). Thus, we have systematically reviewed the state of the art focusing on (1) functional size measurement procedures based on COSMIC for conceptual models, and (2) proposals for defect detection in conceptual models. Therefore, Section 3.1 presents FSM procedures; Section 3.2 presents approaches to detect defects; and Section 3.3 presents some conclusions.

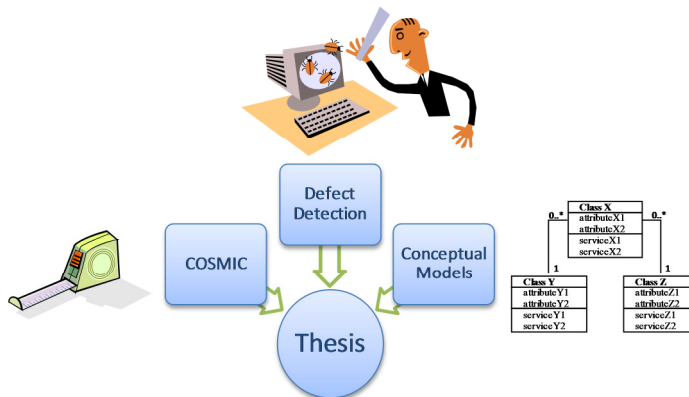


Figure 3.1 Main methods related to this thesis.

3.1 Functional Size Measurement Procedures based on COSMIC

Currently, many functional size measurement procedures have been developed for applying the COSMIC measurement method to the conceptual models of particular software production methods. We summarize these measurement procedures according to the following criteria [Lothar and Dumke 2001]: the version of the measurement method, the context of the proposal, the functional domain (i.e., real time systems, management information systems), the input artifact (i.e., a requirements model, an analysis model, and a design model), the rules to apply the procedure, the instrument to apply the procedure, and the verification of the procedure.

Thus, we present ten proposals of functional size measurement procedures based on COSMIC, and then, we present an overall analysis of the proposals using the criteria presented above. It is important to note that only the proposals by Nagano *et al.* [Nagano and Ajisaka 2003], and Condori-Fernández *et al.* [Condori-Fernández *et al.* 2007] were correctly defined as measurement procedures. Even though the rest of the proposals presented in this analysis were not originally defined as measurement procedures, they do correspond to measurement procedures according to the definition of the International Vocabulary of Basic and General Terms of Metrology [ISO 2004], which defines a measurement procedure as: *a detailed description of a measurement according to one or more measurement principles and to a given measurement method.*

3.1.1 Proposal of Bévo et al. (1999)

Bévo et al. [Bévo *et al.* 1999] perform a mapping between concepts of UML diagrams (use cases, scenarios, and classes) and concepts of COSMIC. A general description of this proposal is presented below:

- *Version of the Measurement Method.* Cosmic-FFP version 2.0 [Abran *et al.* 1999]
- *Context of the Proposal.* Unified Modelling Language (UML) version 1.0
- *Functional Domain.* Management information systems.
- *Input Artifact.* Diagrams of use cases, scenarios, and classes.
- *Rules to Apply the Procedure.* The boundary of the system to be measured is included in the use case diagram. Each use case corresponds to a functional process. The data movements are represented in scenarios, which are sequences of interactions that occur within a use case. Each class of the class diagram corresponds to a data group, and the attributes of those classes correspond to the data attributes. Each actor corresponds to a functional user. The triggering events and the layers are not represented with concepts of UML diagrams.
- *Instrument to Apply the Procedure.* A tool named *Metric Xpert* [Bevo 2005].
- *Verification of the Procedure.* The accuracy of the proposal was verified [Bevo 2005]. To perform this verification, five case studies were measured with the *Metric Xpert* tool. Then, the results were compared with the measures obtained by experts, obtaining differences that fluctuated between 11% and 33%.

3.1.2 Proposal of Jenner (2001)

Jenner [Jenner 2001] discusses the granularity aspect of the use cases in the proposal by Bévo *et al.* presented above. For this reason, the general characteristics of the Jenner proposal are very similar to the characteristics of the Bévo *et al.* proposal.

- *Version of the Measurement Method.* Cosmic-FFP version 2.0 [Abran *et al.* 1999]

- *Context of the Proposal.* UML version 1.0
- *Functional Domain.* Management information systems.
- *Input Artifact.* Diagrams of use cases, sequences, and classes.
- *Rules to Apply the Procedure.* Each functional process is represented by a sequence diagram because Jenner considers that sequence diagrams represent an adequate abstraction level of the use cases. The data movements are represented by the interaction messages of the sequence diagrams. This proposal also uses swimlanes to represent the layers of a system.
- *Instrument to Apply the Procedure.* This procedure has a tool [Jenner 2002].
- *Verification of the Procedure.* The proposal has been verified using case studies.

3.1.3 Proposal of Diab et al. (2001)

Diab et al. [Diab *et al.* 2001] present a set of formal rules that allow the measurement of the functional size of real time applications that are specified with Real-Time Object Oriented Modelling (ROOM) [Selic *et al.* 1994]. The ROOM specifications are used by the Rational Rose Real Time (RRRT) tool for the design and specification of real-time systems. The general characteristics of this proposal are the following:

- *Version of the Measurement Method.* Cosmic-FFP version 2.0 [Abran *et al.* 1999]
- *Context of the Proposal.* The design of an RRRT model might be observed through two different view points: structure and behavior. The structure of an RRRT model is based on three kinds of entities: actors, protocols, and data objects. An actor is an active object that has restricted visibility of and by other actors. A protocol represents a set of messages that can be exchanged among the actors. A data

object is the basic unit of the system data. On the other hand, the dynamic part of an RRRT model is specified with a finite state machine for each actor. Each state machine can be defined with states, sub-states, actions, and transitions between the states.

- *Functional Domain.* Real-time systems.
- *Input Artifact.* RRRT model (static and dynamic part).
- *Rules to Apply the Procedure.* The boundary of the system to be measured is represented by a set of actors. The layers correspond to a set of actors with the same level of abstraction, which must be selected by the practitioners using their human judgment. Each transition corresponds to a functional process. The data movements are represented by actions and messages. Actors and protocol classes correspond to data groups, and the attributes and variables of these classes correspond to the data attributes.
- *Instrument to Apply the Procedure.* A tool named *μRose* [Diab *et al.* 2005]. This tool implements the measurement procedure that is updated to version 2.2 of Cosmic-FFP [ISO/IEC 2003a].
- *Verification of the Procedure.* The rules of the proposal have been verified by experts of COSMIC. In addition, this proposal has been applied to case studies, and the results have been compared with the measures obtained by experts. Finally, the tool assures the repeatability and consistency of the proposal.

3.1.4 Proposal of Poels (2002)

Poels [Poels 2002] presents a mapping between concepts of COSMIC and the concepts of the business and services models of MERODE [Dedene and Snoeck 1994]. Later, this proposal was extended to allow the measurement of multilayer applications [Poels 2003b], specifying that the business model corresponds to a layer, and the services model corresponds to another layer. The general characteristics of this proposal are the following:

- *Version of the Measurement Method.* Cosmic-FFP version 2.1 [Abran *et al.* 2001]
- *Context of the Proposal.* The MERODE development method. This method is based on the MERODE conceptual model, which is comprised of a business model and a services model. The business model is composed by a class diagram, an object-event table, and a state transition diagram. The services model specifies the generation of events by the user and their transmission to the business model.
- *Functional Domain.* Management information systems.
- *Input Artifact.* MERODE model (business and services models).
- *Rules to Apply the Procedure.* Poels defines the rules separately for each model of MERODE. The users of the business model correspond to the services model. The boundary of the business model corresponds to the boundary between the business model and the users. Each functional process of the business model corresponds to a set of class methods over all of the enterprise objects, which are invoked by the occurrence of a type of business event. Each data movement corresponds to each class method that composes a functional process. In the business model, the exit data movements are not represented. The data groups correspond to the classes of the business model. On the other hand, the users of the services model correspond to the user interface model (this model is not specified in the MERODE model). The boundary of the services model corresponds to the boundary between the services model and the users. Each functional process of the services model corresponds to a non-persistent service object that is invoked by an input, output or control service request message or by a business event occurrence (for output object only). Again, each data movement corresponds to each class method that composes a functional process, and all the types of data movements are represented in the services model.

- *Instrument to Apply the Procedure.* Manual application of the procedure.
- *Verification of the Procedure.* This proposal has been validated theoretically [Poels 2003a].

3.1.5 Proposal of Nagano et al. (2003)

Nagano et al. [Nagano and Ajisaka 2003] present a measurement procedure to measure the functional size of real-time applications specified using xUML [Mellor and Balcer 2002]. The general characteristics of this proposal are:

- *Version of the Measurement Method.* Cosmic-FFP version 2.0 [Abran et al. 1999]
- *Context of the Proposal.* The Shlaer-Mellor development method [Shlaer and Mellor 1992]. This method is an object-oriented method that uses xUML to specify systems.
- *Functional Domain.* Real-time systems.
- *Input Artifact.* Classes, state-transition, and collaboration diagrams.
- *Rules to Apply the Procedure.* The candidate data groups are attributes and relationships between objects of the class diagram. Also, the parameters of messages and control signals are candidate data groups. The triggering events are identified in the collaboration diagrams, which include the relationship between the external entity and the objects of the system. The functional processes correspond to a sequence of data movements. Finally, the data movements correspond to the actions that an object performs to move it from one state to the next state according to the collaboration diagram.
- *Instrument to Apply the Procedure.* Manual application of the procedure.

- *Verification of the Procedure.* This proposal has been applied to the Rice Cooker case study [COSMIC_Group 2003], and the results were compared with the results obtained by experts, obtaining a difference of 53%.

3.1.6 Proposal of Azzouz et al. (2004)

Azzouz et al. [Azzouz and Abran 2004] present a tool that automates the measurement of the functional size of applications developed with the Rational Unified Process (RUP) [Kruchten 2000]. The general characteristics of this proposal are:

- *Version of the Measurement Method.* Cosmic-FFP version 2.2 [Abran et al. 2003]
- *Context of the Proposal.* Rational Unified Process. This method uses UML to specify the systems.
- *Functional Domain.* Management information systems.
- *Input Artifact.* Use case diagrams, scenarios, and detailed scenarios.
- *Rules to Apply the Procedure.* Azzouz et al. base their proposal on the rules described by Bévo [Bévo et al. 1999] and Jenner [Jenner 2001]. However, Azzouz considers that the layer cannot be represented in the UML diagrams. Therefore, the user of the tool must manually identify the layers of the system. Also, this proposal adds a stereotype to identify the triggering events in the use case diagrams. The measurement is performed in three phases of RUP: in the business modeling and requirement analysis phase, the artifact used is the use case diagram; in the analysis phase, the artifact used is the scenario; and in the design phase, the artifact used is the detailed scenario.
- *Instrument to Apply the Procedure.* A tool integrated in the Rational Rose tool.

- *Verification of the Procedure.* The tool was verified using the Rice Cooker case study [COSMIC_Group 2003].

3.1.7 Proposal of Condori-Fernández et al. (2004)

Condori-Fernández et al. [Condori-Fernández *et al.* 2007] present a measurement procedure to estimate the functional size of object-oriented systems from the requirements specifications that are defined using the OO-Method approach [Pastor *et al.* 2001]. The general characteristics of this proposal are:

- *Version of the Measurement Method.* Cosmic-FFP version 2.2 [Abran *et al.* 2003]
- *Context of the Proposal.* The development method OO-Method. This method is based on a formal language. It is an object-oriented method that allows the automatic generation of final applications by means of model transformations [Pastor and Molina 2007]. The software production process in OO-Method is represented by four models: the requirements model, the conceptual model, the execution model, and the implementation model. The requirement model specifies the system requirements using a set of techniques such as the mission statement, the functions refinement tree, and the use case diagram. To establish the traceability between the requirements model and the conceptual model, the requirements model uses sequence diagrams. The conceptual model captures the static and dynamic properties of the functional requirements of the system (object, dynamic, and functional models). The conceptual model also allows the specification of the user interfaces in an abstract way through the presentation model. The execution model allows the transition from the problem space to the solution space. The implementation model corresponds to the final application.

Thus, the software product can be generated in a systematic and automatic way for different platforms.

- *Functional Domain.* Management information systems.
- *Input Artifact.* OO-Method requirements model (functions refinement tree, use case diagrams, and sequence diagrams).
- *Rules to Apply the Procedure.* The boundary of the system to be measured corresponds to the border between the set of use cases and the actors of the use case diagram. Each functional process corresponds to each elementary function of the functions refinement tree (primary use case). Also, each secondary use case corresponds to a functional process. The data groups are identified in the sequence diagram. Each different actor, control class, or entity class of the sequence diagram corresponds to a data group. The data movements correspond to the messages of the sequence diagrams. In this proposal a single layer is identified because there is not a functional partition at the requirements level. The triggering events are not represented.
- *Instrument to Apply the Procedure.* Manual application of the procedure.
- *Verification of the Procedure.* This proposal has been rigorously verified in several ways [Condori-Fernández 2007]: according to measurement theory; in conformity with COSMIC; using the formal framework DISTANCE; evaluating the repeatability and reproducibility of the measures obtained [Condori-Fernández and Pastor 2006], and evaluating its adoption in practice.

3.1.8 Proposal of Habela et al. (2005)

Habela et al. [Habela *et al.* 2005] present an extension of the use case model, which allows the measurement of the functional size using COSMIC. The general characteristics of this proposal are:

- *Version of the Measurement Method.* Cosmic-FFP version 2.2 [Abran *et al.* 2003].
- *Context of the Proposal.* UML version 1.5
- *Functional Domain.* Management information systems.
- *Input Artifact.* Use case diagrams, and detailed use cases using a template that includes references to business rules, pre-conditions, post-conditions, and a description in steps of the main and alternatives scenarios.
- *Rules to Apply the Procedure.* Each use case corresponds to one or more functional processes. The data movements are identified in each step described in the scenarios. Each step specifies the movement of a set of data attributes. The uses, extends, and generalizations between use cases are taken into account to avoid redundancies in the measurement.
- *Instrument to apply the Procedure.* Manual application of the procedure.
- *Verification of the Procedure.* We did not find studies of validation, verification, or application of this proposal.

3.1.9 Proposal of Grau et al. (2007)

Grau et al. [Grau and Franch 2007b] present a set of mapping rules to measure the functional size of i* models generated by means of reengineering of systems using PRiM [Grau and Franch 2007a]. The general characteristics of the Grau et al. proposal are the following:

- *Version of the Measurement Method.* Cosmic-FFP version 2.2 [Abran *et al.* 2003]
- *Context of the Proposal.* The PRiM method, which is a process reengineering i* method that addresses the specification, analysis and design of information systems from a reengineering point of

view. In PRiM, the i^* model is comprised of two models: an operational i^* model (that contains the functionality of the system), and an intentional i^* model (that contains the non-functional requirements). To generate the operational i^* model, scenario-based templates named *Detailed Interaction Scripts* are used. These templates describe the information of each activity of the current process by means of pre-conditions, post-conditions, triggering events, and a list of actions undertaken in the activity.

- *Functional Domain*. Management information systems.
- *Input Artifact*. Detailed interaction scripts, and an operational i^* model.
- *Rules to Apply the Procedure*. The boundary of the system to be measured corresponds to the actor of the operational i^* model that represents the different pieces of the system. The users are actors of the operational i^* model that represent one or more human roles. The data movements are identified in the operational i^* model and correspond to any dependency where the *dependum* is a resource. Each functional process corresponds to an activity of the detailed interaction scripts. The triggering events are part of the conditions associated to the activity. Finally, the data groups correspond to the resources of the detailed interaction script.
- *Instrument to Apply the Procedure*. A tool named *J-PRiM*.
- *Verification of the Procedure*. This proposal has been applied to the C-Registration case study [Khelifi *et al.* 2003], and the results have been compared with the results obtained by experts, obtaining a difference of 53%.

3.1.10 Proposal of Levesque et al. (2008)

Levesque et al. [Levesque *et al.* 2008] apply COSMIC to measure the functional size of systems from use case diagrams and sequence diagrams.

This proposal classifies the functional processes in two groups: data movement types and data manipulation types. The general characteristics of the Levesque et al. proposal are the following:

- *Version of the Measurement Method.* Cosmic-FFP version 2.1 [Abran et al. 2007]
- *Context of the Proposal.* UML version 1.4, and UML version 2.0
- *Functional Domain.* Management information systems.
- *Input Artifact.* Use cases and sequence diagrams.
- *Rules to Apply the Procedure.* For the functional processes corresponding to the data movement type, each use case is a functional process. The actors of the use case are the users. The entities of the sequence diagram are the data groups. The data movements correspond to the messages among the entities of the sequence diagram. On the other hand, the data manipulations correspond to the conditions associated to the error messages of the sequence diagrams. Finally, this proposal obtains the functional size aggregating the messages between the actors and objects of the sequence diagrams.
- *Instrument to Apply the Procedure.* Manual application of the procedure.
- *Verification of the Procedure.* This proposal has been applied to the Rice Cooker case study [COSMIC_Group 2003], and the results have been compared with the results obtained by experts, obtaining a difference of 8%.

3.1.11 General Analysis of Measurement Procedures

In this section, a general analysis of the COSMIC measurement procedures found in the literature review is presented according to the following criteria: version of the COSMIC method, context of the proposal, functional domain,

input artifact, rules to apply the procedure, instrument to apply the procedure, and verification of the procedure.

With respect to the version of the COSMIC measurement method, we observed that four proposals (Bévo, Jenner, Diab, and Nagano) use the 2.0 version, two proposals (Poels and Levesque) use the 2.1 version, and four proposals (Azzouz, Condori-Fernández, Habela, and Grau) use the 2.2 version. It is important to note that the proposal by Nagano (which was defined in 2003) uses the 2.0 version in spite of newer versions of COSMIC already existing in 2003. It is also important to note that the proposal by Levesque (which was defined in 2008) uses the 2.1 version in spite of the version 3.0 of COSMIC already existing in 2008. Our opinion is that newer versions of COSMIC provide improvements and clarifications that help to better understand the measurement method and to obtain accurate measures. Therefore, we consider that the use of the last version of the method is very important for the correct development of measurement procedures.

With regard to the context of the procedure, five proposals (Bévo, Jenner, Azzouz, Habela, and Levesque) allow the measurement of the functional size of Management Information Systems (MIS) using UML models. As is well known, the UML models don't allow the specification of all the functionality of the final application and also can have consistency and ambiguity problems [Berkenkötter 2008] [France *et al.* 2006] [Opdahl and Henderson-Sellers 2005]. One proposal (Diab) measures RRRT models, one proposal (Poels) measures MERODE models, one proposal (Nagano) measures xUML specifications, one proposal (Condori-Fernández) measure OO-Method requirement models, and one proposal (Grau) measures *i** models.

The UML, MERODE, and *i** models do not have enough expressivity to specify all the functional requirements of the applications (for instance, these models do not allow the specification of the values assigned to the attributes of the classes, the interaction units, etc.). The same situation occurs with the OO-Method requirement model. In addition, the Poels' proposal is restricted

to a specific technology because it uses the AndroMDA tool to specify the presentation of the application and to generate the final application. Therefore, the proposals based on these models incorrectly measure the functional size of applications. The proposals based on the RRRT model and the xUML specification have enough semantic formalization to specify all the functional requirements, allowing the measurement of the functional size of the applications.

With respect to the functional domain, we observed that only two proposals (Diab, Nagano) have been developed for the domain of real time systems. The remaining nine proposals have been developed for the domain of management information systems. We did not find any measurement procedure proposal for other domains (such as algorithmic systems, geographical systems, or ubiquitous systems), in spite of the COSMIC measurement method can be also applied to other software system domains.

With regards to the input artifact, all the proposals use more than one input artifact. Seven proposals (Bévo, Jenner, Azzouz, Condori-Fernández, Habela, Grau, and Levesque) use input artifacts obtained in the requirements phase, and three proposals (Diab, Poels, Nagano) use input artifacts obtained in the analysis phase. However, the requirements and analysis models don't have enough expressiveness to specify all the functionality of final applications.

With respect to the rules to apply the procedure, only one proposal (Condori-Fernández) perform the design of the measurement procedure, defining the objectives of the procedure, the characterization of the concept to be measured, the mapping with the concepts of COSMIC, and the measurement rules. The remaining nine proposals only define some mappings between the concepts of COSMIC and the concepts of the conceptual models to be measured.

The design of a measurement procedure is a key stage in the development of a measurement procedure (correctly abstracting the elements that will be measured), since, otherwise, the procedure may not measure

what should be measured according to the specifications of the base measurement method selected. It is also important to keep in mind the direct influence that the design of a measurement procedure has on the application and possible automations of the procedure.

With regard to the instrument to apply the procedure, five proposals (Bévo, Jenner, Diab, Azzouz, Grau) have been automated, and five proposals (Poels, Nagano, Condori-Fernández, Habela, and Levesque) must be applied manually. The manual measurement of functional size is generally a very time-consuming and error-prone task. Therefore, it is very important to automate the measurement procedures in order to obtain a solution that can be efficiently applied in academic and industrial environments. Also, a tool that automates the measurement procedures reduces the measurement costs and the measurement training, and ensures repeatability of the measures.

Finally, with respect to the verification of the procedure, we observed that only one proposal (Habela) has not been verified in some way. The remaining nine proposals have been verified using different techniques: using case studies, performing theoretical validations, performing conformity validations, using controlled experiments, etc. Thus, it is important to keep in mind that a high quality design of a functional size measurement procedure is not enough to assure the quality of the measures obtained by this procedure. To ensure the quality of the results obtained, it is also essential to verify the developed procedure.

3.2 Approaches to Detect Defects in Conceptual Models

Defect detection refers to anomalies found in software products in order to correct them and, therefore, obtain software products of better quality. The IEEE 1012 standard for software verification and validation [IEEE 2004]

define an anomaly as *anything observed in the documentation or operation of software that deviates from expectations based on previously verified software products or reference documents*. This definition is very broad, so that different people can find different anomalies in the same software artifact, and even the anomalies that one person found could have been not perceived as anomalies by another person. This situation has caused many researchers to redefine the concepts of error, defect, failure, fault, etc.; and many times these concepts have been used indistinctly [Fenton and Neil 1999].

In order to avoid the proliferation of concepts related to software anomalies, the latest version of the IEEE 1044 standard [IEEE 2009] defines the concepts of error, defect and fault. We use these concepts to apply them to MDD processes as follows:

- Error: is a human action that produces an incorrect result in a model.
- Defect: is a deficiency in a model, meaning that the model does not meet its requirements or specifications and needs to be either repaired or replaced.
- Fault: is a manifestation of a defect in software.

Currently, there are several proposals to detect defects in models that can be used in a MDD context. These approaches usually apply reading techniques or rules defined from the experience of some researchers. We analyze the literature according to the models inspected, the defect types found, and the instrument applied in each proposal.

3.2.1 Proposal of Travassos et al. (1999)

Guilherme Travassos et al. [Travassos *et al.* 1999] use reading techniques to perform software inspections in high-level, object-oriented designs. They use UML diagrams that are focused on data structure and behavior. These authors advocate that the design artifacts (a set of well-related diagrams)

should be read in order to determine whether they are consistent with each other and whether they adequately capture the requirements. Design defects occur when these conditions are not met.

These authors use a defect taxonomy that is borrowed from requirements defects [Basili *et al.* 1996], which classifies the defects as *Omission*, *Incorrect Fact*, *Inconsistency*, *Ambiguity*, and *Extraneous Information*. These authors perform an empirical study to evaluate the defect detection in design models. The general characteristics of this proposal are the following:

- *Models*. UML Class, UML State-transition, and UML Sequence.
- *Defect Types*. They present one example defect type for each classification. However, these authors do not present the types of defects found in the empirical study.
- *Instrument to Apply the Proposal*. Manual application.

3.2.2 Proposal of Laitenberger et al. (2000)

Laitenberger et al. [Laitenberger *et al.* 2000] present a controlled experiment that compares the checklist-based reading (CBR) technique with the perspective-based reading (PBR) technique to detect defects in UML models.

These authors define three inspection scenarios in the UML models in order to detect defects from different viewpoints (designers, testers, and programmers). A brief description of this proposal is the following:

- *Models*. UML Class, UML Collaboration, and Fusion Operation Model.
- *Defect Types*. These authors do not explicitly identify the types of defects found in UML models. However, they present a set of concepts that must be checked in the UML models from different viewpoints, which are used to infer the types of defects from these concepts. Table 1 presents the defect types inferred.
- *Instrument to Apply the Proposal*. Manual application.

Table 1. Defect Types presented by Laitenberger et al. (2000).

Quality	Model	Defect Types
Consistency among diagrams	UML Class	<ul style="list-style-type: none"> - A class in the design class diagram is not a class in the system class diagram (with the same name). - The number, types, and names of the attributes of a class in the design class diagram are not the same in the class of the system class diagram. - The number, names, and arguments of the methods of a class in the design class diagram are not the same in the class of the system class diagram. - The associations with their cardinality and arity in the design class diagram are not the same in the system class diagram. - The constraints between classes of the design class diagram are not the same constraints for these classes in the system class diagram.
	UML Collaboration	<ul style="list-style-type: none"> - An object that does not correspond to a class of the class diagram. - The collaboration diagram has messages that do not correspond to the system operation. - The messages do not have the same number and type of arguments as the operations of the system described in the Operation model.
	Fusion Operation Model	<ul style="list-style-type: none"> - Operations (read, change, send, and result) that do not have the corresponding message in the collaboration diagram.
Correctness	UML Class	<ul style="list-style-type: none"> - The types of the attributes of a class are not specified. - The methods of a class are not specified. - Parameters that do not have a type associated.

3.2.3 Proposal of Conradi et al. (2003)

Conradi et al. [Conradi *et al.* 2003] present a controlled experiment that was designed to perform a comparison between an old reading technique used by the Ericsson company and an Object-Oriented Reading Technique (OORT) for detecting defects in UML models. These authors present a summary of

the defects detected using both inspection techniques for the same project. The general characteristics of the Conradi et al. proposal are the following:

- *Models.* UML Class, UML Sequence, UML State-transition, and UML Use-Case.
- *Defect Types.* The findings of the controlled experiment are that one group of subjects detected 25 defects using the old technique (without any overlaps of the defects detected), while the other group of subjects detected 39 defects using the OORT technique (with 8 overlaps in the defects detected). However, these authors do not present the types of defects found in the experiment.
- *Instrument to Apply the Proposal.* Manual application.

3.2.4 Proposal of Gomma et al. (2003)

Gomma et al. [Gomaa and Wijesekera 2003] present an approach for the identification and correction of inconsistency and incompleteness across UML views. This approach is applied in the COMET method [Gomaa 2000], which uses the UML notation. The general characteristics of the Gomma et al. proposal are the following:

- *Models.* UML Use-case, UML Sequence, UML Class, and COMET State-transition diagram.
- *Defect Types.* The authors present 7 defect types related to the consistency between models (see Table 2): 1 defect type for the consistency between use-case diagrams and sequence diagrams; 4 defect types for the consistency between class diagrams and state-transition diagrams; and 2 defect types for the consistency between sequence diagrams and state-transition diagrams.
- *Instrument to Apply the Proposal.* Manual application.

Table 2. Defect Types presented by Gomma et al. (2003).

Quality	Model	Defect Types
Consistency among diagrams	UML Use-case	- A use case that does not correspond to at least one scenario described by an interaction diagram.
	COMET state transition	<ul style="list-style-type: none"> - Each Statechart that does not correspond to a state that is dependent on the control class in a class diagram. - The values of the current states, events, actions, and activities that appear on a Statechart that are not declared as attribute values of the respective state, event, action, and activity attributes for the state that is dependent on the control class. - An event on a Statechart that does not correspond to a method of the state of the control class in the class diagram. - Variables used to define conditions in any Statechart that are not attributes of the state that is dependent on the control class in the corresponding class diagram. - Each event on a Statechart that corresponds to an incoming message on the state that is dependent on the control object, which is not represented in an interaction diagram (which executes the Statechart). - Each action on a Statechart that corresponds to an outgoing message on the state that is dependent on the control object, which is not represented in an interaction diagram (which executes the Statechart).

3.2.5 Proposal of Kuzniarz et al. (2003)

Kuzniarz et al. [Kuzniarz 2003] present a set of inconsistencies manually found in student designs produced in a sample didactic development process. This proposal corresponds to a laboratory experiment that was developed to explore the nature of inconsistency in UML designs. The general characteristics of the Kuzniarz et al. proposal are the following:

- *Models.* UML Use-case, and UML Sequence.

- *Defect Types*. The authors present 8 defect types (Table 3) based on subjective but common-sense judgment.
- *Instrument to Apply the Proposal*. Manual application.

Table 3. Defect Types presented by Kuzniarz et al. (2003).

Quality	Model	Defect Types
Consistency among diagrams	UML Use-case	<ul style="list-style-type: none"> - The actor that is defined in the use case diagram is not the same actor that takes part in the interaction that is defined in the corresponding sequence diagram. - Not all the steps that are defined in the use case description correspond to messages in the system sequence diagram. - There are extension points for the extension of use cases that are missing (not represented) in the diagram of the controller use case.
	UML Sequence	<ul style="list-style-type: none"> - An iteration symbol that is related to an iterative task is missing in the sequence diagram. - There are links used in sequence diagrams that are not associations in the class diagram. - There are sequences of messages in the sequence diagram that are not acceptable for the controller use case. - There are elements used in pre- and post-conditions of the contracts that are not defined in the class model. - There are sequences of messages in the sequence diagram that are not an acceptable subsequence for the state machines that take part in the sequence diagram.

3.2.6 Proposal of Berenbach (2004)

Berenbach [Berenbach 2004] presents a set of heuristics for analysis and design models that prevents the introduction of defects in the models. This allows semantically correct models to be developed. In addition, Berenbach presents the DesignAdvisor tool created by Siemens to facilitate the inspection of large models. This tool implements the heuristics proposed by

Berenbach for evaluating the goodness of the analysis and design models. The general characteristics of the Berenbach proposal are the following:

- *Models*. UML Use-case, UML Class, and UML Business.
- *Defect Types*. For the analysis models, Berenbach presents 10 heuristics for model organization, 5 heuristics for use case definition, 3 heuristics related to the use-case relationships, and 14 heuristics related to business object modeling. For the design models, he presents 2 heuristics for the class model. Table 4 presents the defect types inferred from the heuristics.
- *Instrument to Apply the Proposal*. DesignAdvisor tool.

3.2.7 Proposal of Lange et al. (2004)

Lange et al. [Lange and Chaudron 2004] identify the incompleteness of UML diagrams as a potential problem in the subsequent stages of a model-oriented software development. These authors refer to the completeness of a model by means of the aggregation of three characteristics: (1) the well-formedness of each diagram that comprises the model; (2) the consistency between the diagrams that comprise the model; and (3) the completeness among the diagrams that comprise the model. Note that the authors use the completeness concept to define the completeness of a model. Since this is equivalent to reusing the same concept (completeness) for its own definition, they do not really describe what is understood by completeness. The general characteristics of the Lange et al. proposal are the following:

- *Models*. UML Use-case, UML Sequence, and UML Class.
- *Defect Types*. These authors identify eleven types of defects of UML models in their study (see Table 5): 5 types of defects related to the well-formedness of the diagrams, 3 types related to the consistency among the diagrams, and 3 other types related to the completeness among the diagrams. In 2006, Lange et al. [Lange and Chaudron 2006] present an experimental study to identify the risks related to

eight defect types, four of which were presented in the previous study. Table 5 presents the defect types of both works.

- *Instrument to Apply the Proposal.* They use the MetricView tool [Lange *et al.* 2007] to analyze the models, which graphically shows the defects found in UML models.

Table 4. Defect Types presented by Berenbach (2004).

Quality	Model	Defect Types
Consistency among diagrams	UML Use-case	<ul style="list-style-type: none"> - Actors in use-case diagrams that are not specified in the context diagram. - Use case without an interaction diagram that shows the possible scenarios.
	UML Class	<ul style="list-style-type: none"> - An interface in the class diagram that is not used to communicate with a concrete use case. - A class that is not instantiated in any process of the system (sequence and collaboration diagrams). - Methods of the interface class that are not represented in the process of the system (sequence and collaboration diagrams). - Classes in the class diagram that are not specified in the use-case diagram. - Interfaces in the class diagram that are not specified in the use-case diagram.
Correctness	UML Use-case	<ul style="list-style-type: none"> - Multiple entry point for the system in the use-case diagram. - Diagrams without a description and status. - Concrete use cases without a definition. - Abstract use cases that are not realized by a concrete use-case. - Extends use-case relationship that is specified between use cases that are not concrete. - A concrete use case that includes an abstract use case.
	UML Business	<ul style="list-style-type: none"> - Services of business objects that do not have defined pre- and post-conditions.
	UML Class	<ul style="list-style-type: none"> - An interface class with private methods.

Table 5. Defect Types presented by Lange et al. (2004 and 2006).

Quality	Model	Defect Types
Consistency among diagrams	UML Sequence	- Messages between unrelated classes. - Objects of the sequence diagram that are not related to a class in the class diagram. (2006)
	UML Class	- Classes that are not called in the sequence diagram. - Interfaces that are not called in the sequence diagram. - Methods that are not called in the sequence diagram.
	UML Use-case	- Use cases without sequence diagrams. (2006)
Correctness	UML Sequence	- Objects without a name. - Abstract classes in sequence diagrams. - Messages without a name. - Messages without a method. - Message in wrong direction. (2006)
	UML Class	- Classes without methods. - Interfaces without methods. - Classes with public attributes. - Multiple definitions of classes with equal names. (2006)

3.2.8 Proposal of Leung et al. (2005)

Leung et al. [Leung and Bolloju 2005] present a study that aims to understand the defects frequently committed by novice analysts in the development of UML Class models.

These authors use Lindland et al.'s quality framework [Lindland et al. 1994] to evaluate the quality of the class diagrams. They distinguish five classifications that allow the evaluation of the syntactic quality, semantic quality, and pragmatic quality. These five classifications are: syntactic (for the syntactic quality); validity and completeness (for the semantic quality); and the expected is missing and the unexpected is present (for the pragmatic quality). The general characteristics of the Leung et al. proposal are the following:

3. State of the Art

- *Models*. UML Class.
- *Defect Types*. The authors obtain 103 different types of defects in 14 projects. However, the authors only detail 21 types of defects for one class diagram (see Table 6).
- *Instrument to Apply the Proposal*. This approach is applied manually.

Table 6. Defect Types presented by Leung et al. (2005).

Quality	Model	Defect Types
Correctness	UML Class	<ul style="list-style-type: none"> - Missing association label or cardinality detail. - Improper label for a class, an association, an attribute, or an operation. - Improper notation for an association, an aggregation, or a generalization. - The non-implicit operations that are present in sequence diagram are not included. - Implicit operation is listed. - Wrong association cardinality (reversed or wrong range). - Wrong location of an attribute or an operation. - Wrong association grouping. - Missing class, attribute, operation, or association. - Incomplete class description. - Operation that cannot be realized (using attributes and relationships). - Does not use domain-specific terminology - Poor layout of the class diagram - Insufficient distinction among sub-classes. - Operation naming is improper or ambiguous. - Associations are replicated at sub-classes. - Manual operation is represented as association. - Excessive use of generalization, PK concept, FK concept, or emphasis on statistical information. - Redundant attributes. - Redundant associations. - Implementation detail is present in the diagram.

3.2.9 Proposal of Bellur et al. (2006)

Bellur et al. [Bellur and Vallieswaran 2006] perform an impact analysis of UML design models. This analysis evaluates the consistency of the design and the impact of a design change over the code. In order to evaluate the consistency of the design models, these authors propose the evaluation of the well-formedness of UML diagrams.

The proposal of Bellur et al. [Bellur and Vallieswaran 2006] extends the proposal of Lange et al. [Lange and Chaudron 2004] focusing on the quality of UML conceptual models as well as on the code. The general characteristics of the Bellur et al. proposal are the following:

- *Models.* UML Use-Case, UML Sequence, UML Class, UML State transition, UML Component, and UML Deployment.
- *Defect Types.* These authors identify 4 types of defects for use-case diagrams, 2 types of defects for sequence diagrams, 5 types of defects for the specification of the method sequences, 3 types of defects for the class diagram, 8 types of defects for the state transition diagrams, 2 types of defects for the component diagram, and 2 types of defects for the deployment diagram. Table 7 presents the defect types found by these authors.
- *Instrument to Apply the Proposal.* They analyze the models using their consistency checking tool.

3.2.10 Proposal of Egyed (2006)

Egyed [Egyed 2006] presents an approach to automatically detect inconsistencies among UML models by means of instant checking of the models. The UML / Analyzer tool implements this approach to evaluate the models when a change happens. This tool shows the elements where the inconsistency is produced in the models.

Table 7. Defect Types presented by Bellur et al. (2006).

Quality	Model	Defect Types
Consistency among diagrams	UML Use-case	- A use case that does not reference a use-case sequence diagram.
	UML Sequence	- A variable of a general class used in the sequence diagram that is null or is not a valid class in the class diagram. - A method referenced in the sequence diagram that is null or is not a valid method in the method sequence charts. - An object that is not the sender or the receiver in any interaction. - An object that does not reference a valid class and state diagram. - A message that is not an instance of one class method for some class defined in the system.
	UML State Transition	- A state diagram that is not related to one and only one class. - A state that is not described by one or more attributes of the class. - State change events that do not correspond to messages in the method sequence diagrams.
	UML Component	- An inter-component relationship that does not have 2 terminating end classes which are valid classes in the class diagram. - A component in the component diagram that is not mapped to a physical system described in the deployment diagram.
	UML Deployment	- A deployment diagram that is not related to one or more component diagrams.
Correctness	UML Use-case	- An actor that does not use one or more use cases. - A use case that is not used by one or more actors. - A use case that does not belong to system.
	UML Sequence	- A message that does not have a sender and a receiver object. - A message that does not conform to the signature of the method corresponding to the message.
	UML Class	- A class diagram without classes. - An association without a source and target class. - A class that does not have at least one attribute or method.
	UML State Transition	- A state diagram without one start state and one end state. - A state that has overlap of attribute values describing the state. - A state that is not reachable from the start state. - A state that cannot reach the end state.
	UML Component	- A component diagram without components.

The general characteristics of the Egyed proposal are the following:

- *Models*. UML Sequence, UML Class, and UML State-transition.
- *Defect Types*. These authors uses 24 consistency rules that covered all relevant situations for the consistency of sequence diagrams with class and statechart diagrams (in their domains). However, they do not present the rules or the defect types found in the application of the tool to 29 models.
- *Instrument to Apply the Proposal*. UML/Analyzer tool.

3.2.11 General Analysis of Defect Detection Proposals

The proposals analyzed present defect types that are related to the consistency among diagrams and defect types that are related to the correctness of a particular diagram. The consistency is defined in the IEEE 610 standard [IEEE 1990] as the degree of uniformity, standardization, and freedom from contradiction among the documents or parts of a system or component. The correctness is defined by the same standard [IEEE 1990] as the degree to which a system or component is free from faults in its specification, design, and implementation.

We noticed that all the proposals for defect detection in conceptual models are focused on UML models. However, it is well-known that UML diagrams do not have enough semantic precision to allow the specification of a complete and unambiguous software application [Berkenkötter 2008] [France *et al.* 2006] [Opdahl and Henderson-Sellers 2005], which is clearly observed in the semantic extension points that are defined in the UML specification [OMG 2010]. For this reason, many methodologies have selected a subset of UML diagrams and conceptual constructs and have aggregated the needed semantic expressiveness in order to completely specify the final applications in the conceptual model, making the implementation of MDD technology a reality.

Regarding to the technique used to find defects, some of the proposals use reading techniques, which require to instruct the inspection team on what to look for and how to scrutinize software documents in a systematic manner [Laitenberger *et al.* 2000]. Empirical studies have indicated that these techniques are effective for finding defects. However, reading techniques also have limitations; for instance, the manual inspection of models takes a lot of time, which increases the costs and the delivery date of software products. The remainder of the proposals of defect detection use rules defined by the researchers from their own expertise. However, since these works do not explicitly present how they are defined these rules for the specific diagrams analyzed, it is difficult to define new rules for these diagrams or other kind of diagrams.

3.3 Conclusions

In the systematic revision of the state of the art, we noticed that none of the proposals for measurement procedures based on COSMIC allows the accurate measurement of the functional size of MIS applications from their conceptual models. Therefore, we can state that the prediction of productivity, effort, budget and other indicators using these proposals it is very far from the real values.

Also, we noticed that none of the analyzed proposals on defect detection of models present the rationale involved in the specification of defect detection rules, making it difficult to apply them to other kind of models.

The main limitation of the approaches presented above comes from the lack of expressiveness of the conceptual model that allows the generation of the final application. Additionally, none of the measurement procedures of the state of the art takes advantage of the procedure used to obtain the functional size or the automation of the procedure in order to find defects in the models. We want to tackle these limitations defining an accurate

Functional Size measurement procedure that can be used to detect defects in conceptual models of an MDD environment.

3. State of the Art

Chapter 4

Design of a FSM Procedure

The capability to accurately quantify the size of software developed with a Model-Driven Development (MDD) method is very important to software project managers for evaluating risks, developing project estimates, and having early project indicators. To accurately measure the functional size of MDD applications, a measurement procedure must be designed. This chapter presents the design of a measurement procedure developed according to the COSMIC measurement method version 3.0 [Abran *et al.* 2007].

The design of a measurement procedure includes the definition of the objective of the measurement, the artifact that will be measured, the measurement rules, and the measurement strategy. To do this, it is very important to correctly abstract the elements that will be measured according to the specifications in the selected base measurement method. It is also important to note the direct influence that the design of a measurement procedure has on the application of this procedure. For instance, if the design is incorrect, then the application of the procedure might be misinterpreted and erroneous measures may be obtained. In order to correctly design a measurement procedure, a process for the design must be systematically followed.

Thus, in this chapter we present the activities that must be performed to complete the design of a measurement procedure following the software measurement process model proposed by Jacquet and Abran [Jacquet and Abran 1997]: *definition of the objectives*, *characterization of the concept to be measured*, *selection of the metamodel*, and *definition of the numerical assignment rules*. Since the measurement procedure presented in this chapter has been designed selecting the COSMIC version 3.0 as the base measurement method, we identify that there exist correspondences between the steps and activities of the software measurement process model and the phases and activities of the COSMIC measurement method (see Figure 4.1).

The *definition of objectives* step has a direct correspondence with the determination of the purpose of the measurement in the strategy phase of COSMIC because both specify the objective of the measurement procedure. Moreover, for the correct definition of the objective, it is important to determine the input artefact to perform the measurement and the detail level that must have this artefact to perform a measurement. Thus, the definition of objectives step has also a correspondence with the determination of the scope of the measurement (where the input artefact is specified including its layers and pieces of software) and the determination of the granularity level of the strategy phase of COSMIC.

The *characterization of the concept to be measured* step does not have any correspondence with the phases of the COSMIC measurement method. This makes sense since the COSMIC measurement method has been designed to measure the functional size of software applications without having to characterize what the functional size means in its measurement process.

The *selection of the metamodel* step has a direct correspondence with the mapping phase of COSMIC, because both perform a mapping between the concepts of the artefact to be measured and the concepts of the measurement method. Also, the identification of functional users and boundaries of the strategy phase of COSMIC is performed in this step since this activity

involves the concepts of the metamodel of COSMIC. Moreover, the selection of the metamodel step has a correspondence with the activity ‘identification of data movements’ of the measurement phase of COSMIC, since this activity also involves concepts of the metamodel of COSMIC.

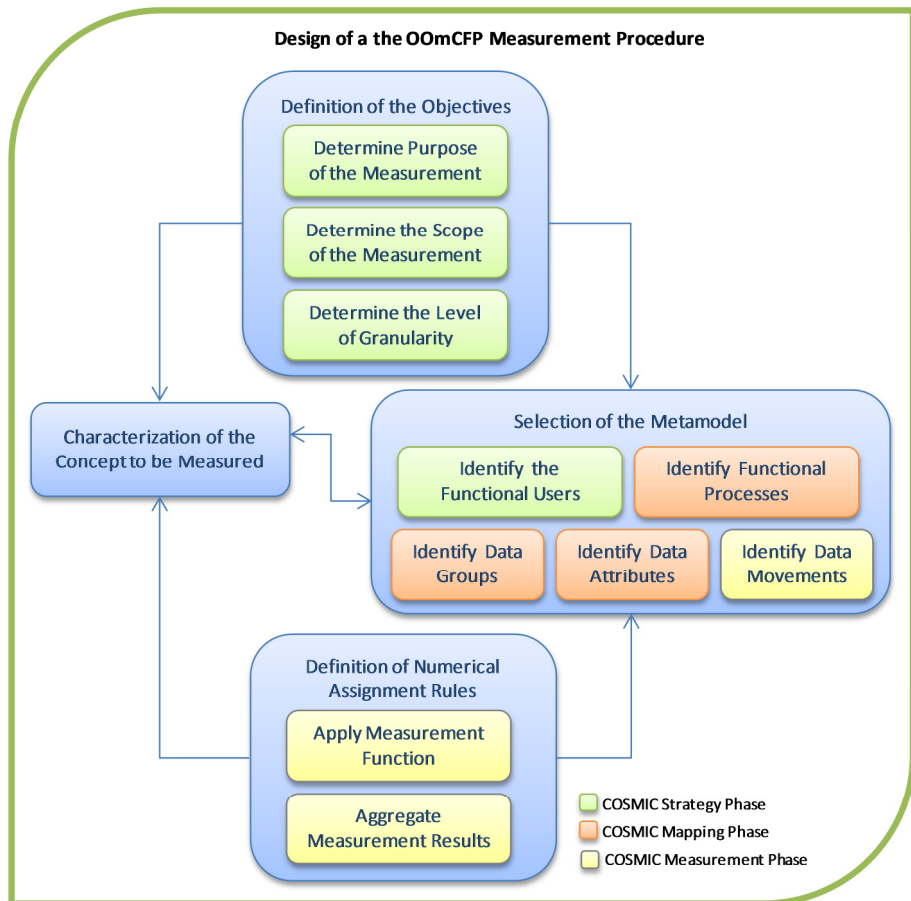


Figure 4.1 Design Process of the OOmCFP Measurement Procedure.

The *definition of the numerical assignment rules* step has a correspondence with the definition and application of measurement rules of

the measurement phase of COSMIC. Figure 4.1 shows the activities of the software measurement process model and their correspondences with the COSMIC activities, which we have followed to design the *OO-Method COSMIC Function Points* (OOmCFP) measurement procedure.

4.1 Definition of the Objectives

Before designing a measurement procedure, it is important to know what we want to measure [Jacquet and Abran 1997], for instance which kind of software, which attribute, etc. Also, it is important to know what is the viewpoint used to perform the measurement, since different viewpoints can obtain different measures of the same software. For instance, the developer viewpoint (which contains all the functionality developed), the user viewpoint (which contains the set of functions developed that the user can access), etc. Finally, in the objective must be specified the intended use of the measures obtained by the measurement procedure. Thus, we define the objective as:

To design a procedure for measuring the accurate functional size from the developer viewpoint of MDD-based software applications that are generated using an object-oriented conceptual model developed with OO-Method. The measures obtained will be used in estimation and budget models.

In this step, the COSMIC strategy phase for the measurement procedure starts (see Figure 4.1). The strategy phase is a relevant phase, which addresses the four key parameters of software functional size measurement that must be considered before actually starting to measure, namely the purpose of the measurement, the scope of the measurement, the identification of functional users, and the level of granularity that should be measured. Determining these parameters helps to address the questions of

‘Which size should be measured?’ or, for an existing measurement, ‘How should we interpret this measurement?’. As Figure 4.1 shows, the purpose, the scope, and the granularity level of the measurement must be defined.

4.1.1 Purpose of the Measurement

The purpose of a measurement is a statement that defines why a measurement is required, and what the result will be used for [Abran *et al.* 2007]. There are many reasons to measure the functional size, so that, the purpose must be clearly state in order to select the most appropriate artifacts for the measurement.

The *purpose* of the measurement in OOmCFP is defined as measuring the accurate functional size of the OO-Method applications generated in an MDD environment from the involved conceptual models. As in the specific case of CARE Technologies company [CARE-Technologies 2011], this functional size will be used to estimate the cost of the OO-Method applications that are specifically generated by the OlivaNova Suite. Although the code of the OO-Method applications is generated automatically, it is important to note that the effort of programming is substituted by a modeling effort. Thus, for the applications generated in an MDD environment, dealing with the cost means dealing with how to measure cost from the involved models.

With the OOmCFP measurement procedure, companies can use the functional size measured from the models to adjust the budget in order to prorate the cost of the software product development. CARE Technologies [CARE-Technologies 2011], which has implemented the tools that support the OO-Method approach, bases its business model on functional size to calculate the cost of the generated applications, prorating the cost of the development and improvement of the tools between the generated applications.

4.1.2 Scope of the Measurement

The scope of the measurement defines a set of functional user requirements that will be included in a measurement exercise. The functional user requirements are defined in the ISO 14143-1 [ISO 1998] as a sub-set of the user requirements. The functional user requirements represent the user practices and procedures that the software must perform to fulfill the users' needs. They exclude quality requirements and any technical requirements.

The OOmCFP measurement procedure uses the OO-Method conceptual model as the input artefact for the measurement of the functional size of applications generated in MDD environments. This conceptual model formally and unambiguously specifies the functional requirements of the applications independently of the technological characteristics that the generated applications will have. The applications generated using the OO-Method approach have a direct correspondence with the involved conceptual model. Thus, these applications do not need manual changes for their correct operation because their complete specification is performed at an abstract level in the conceptual model.

For this reason, the *scope* of the measurement in OOmCFP is the OO-Method conceptual model, which is comprised of four models (Object, Dynamic, Functional, and Presentation). The object model defines the structure and static relationships between the classes. The dynamic model defines the possible valid lives for the objects of a class and the interaction among objects. The functional model captures the semantics associated to object state changes, triggered by the occurrence of events. Finally, the presentation model allows the specification of the user interfaces in an abstract way. With all of these models, the conceptual model has all the details needed for the generation of the final application. The complete definition of the elements of the conceptual model of OO-Method is described in detail in [19].

Once the scope of the measurement has been determined, it is important to identify the layers, the pieces of software, and the peer components that make up the applications. In order to correctly identify the architecture of the software applications, we have used the definitions of layer and piece of software that are presented in the COSMIC Measurement Manual [Abran *et al.* 2007]:

- *Layer*: A layer is a partition resulting from the functional division of a software architecture that together with hardware forms a whole computer system, where: layers are organized in a hierarchy; there is only one layer at each level in the hierarchy; there is a 'superior/subordinate' hierarchical dependency between the functional services provided by the software of the layers; and the software of a layer that interchanges data with other layers interprets only the part of the data that interchanged.
- *Piece of Software*: A piece of software is the part of the software that is implemented in each layer.

The OO-Method software applications are generated according to a three-tier software architecture: presentation, logical, and database. Each tier of the architecture is associated with the other tiers in a superior/subordinate hierarchical dependency. Therefore, the presentation tier can use the services of the logic tier because the logic tier is beneath the presentation tier in the hierarchy. In the same way, the logic tier can use the services of the database tier because the database tier is beneath the logic tier in the hierarchy. These tiers correspond with the layer definition of the COSMIC Measurement Manual [Abran *et al.* 2007]. Thus, we distinguish three layers in an OO-Method application: a client layer, which contains the graphical user interface; a server layer, which contains the business logic of the application; and a database layer, which contains the persistence of the applications.

In each layer of an OO-Method application, there is a piece of software that can interchange data with the pieces of software of the other layers.

Thus, we distinguish, respectively, three pieces of software in an OO-Method application: the client piece of software, the server piece of software, and the database piece of software. Figure 4.2 illustrates the layers and pieces of software of an OO-Method application.

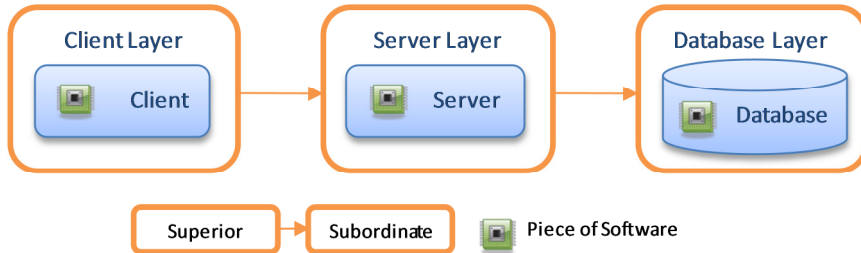


Figure 4.2 Pieces of software and layers of an OO-Method application.

4.1.3 Granularity Level of the Software

The *granularity level* is the level of detail that the pieces of software included in the measurement must have. Since the measurement will be performed in conceptual models that must be valid to generate the final applications in an MDD environment, the granularity level is a low level because all the details in the OO-Method conceptual model are needed to automatically generate the final applications.

4.2 Characterization of the Concept

The objective of OOmCFP measurement procedure states (see section 4.1) that the concept to be measured by OOmCFP is the functional size. The functional size is defined by the ISO/IEC 14143-1 standard [ISO 1998] as *the size of software derived by quantifying the functional user requirements*.

The input artefact used by OOmCFP to measure the functional size of the OO-Method applications is the conceptual model that is used to generate these applications. Considering that the functional user requirements represent a sub-set of the user requirements that specifies what must be done by an application (excluding its technological features and non-functional characteristics), the OO-Method conceptual model contains the set of functional requirements of the OO-Method applications. Thus, the entity to be measured by the OOmCFP measurement procedure will be an OO-Method conceptual model, and the attribute to be measured will be the functional size.

4.3 Selection of the Metamodel

Since software is not a tangible product, it can be made visible through its metamodel representation [Jacquet and Abran 1997]. In general terms, a metamodel is the artifact used to specify the abstract syntax of a modeling language: the structural definition of the involved conceptual constructs with their properties, the definition of relationships among the different constructs, and the definition of a set of rules to control the interaction among the different constructs specified [Selic 2007].

In EMOF, a metamodel is represented by means of a class diagram, where each class of the diagram corresponds to a construct of the modeling language involved. A metamodel for a FSM provides the basis for the design of the measurement rules that identify and measure the elements contained in the metamodel. Figure 4.3 shows the COSMIC metamodel, which illustrates the information that should be represented by the software artifact to be measured. This metamodel was designed from the COSMIC measurement manual version 3.0 [Abran et al. 2007]. We selected this metamodel for the design of OOmCFP for the simplicity of the metamodel in quantifying

functional size without being limited by maximum values (as occurs in IFPUG-FPA).

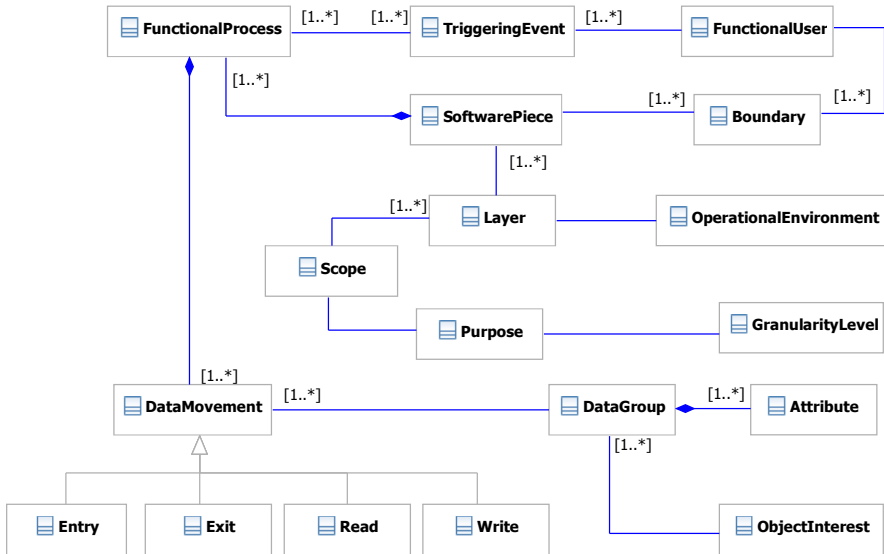


Figure 4.3 Metamodel of COSMIC version 3.0

As Figure 4.3 shows, the *scope* of the measurement is determined by the *purpose* of the measurement. The purpose and the scope define the set of pieces of the software that will be measured and the level of detail of each piece of software (*granularity level*).

Each measurand is focused on a set of *objects of interest* that can be physical or conceptual and that are related to *data groups*. Every data group has a set of *attributes*. Also, every data group participates in one or more *data movements*, which can be an entry data movement (E), an exit data movement (X), a read data movement (R), or a write data movement (W). Two or more data movements occur in a *functional process* that belongs to a *piece of software* of the *layer* to be measured. Each layer of the piece of software is associated to one *operative software environment*.

Finally, every functional process is triggered by *triggering events* carried out by the *functional user*. The functional users are the users of the pieces of software that have been measured and are separated by a *boundary* from the pieces of software. A user is defined in the ISO 14143-1 [ISO 1998] as any person that specifies Functional User Requirements and/or any person or thing that communicates or interacts with the software at any time. Also, the ISO 14143-1 clarifies that thing includes software applications, animals, sensors, other hardware, etc. For a measurement of a multilayer application, the functionality that must be built by the developer is very important because it allows the accurate size of the application to be determined.

In the selection of the metamodel step, the identification of functional users and the boundaries of the COSMIC strategy phase are performed (see Figure 4.1). Also, Figure 4.1 shows that the COSMIC mapping phase is completely performed in this step. The mapping phase presents the rules to identify the functional processes, data groups, and data attributes in the software specification (i.e., in the conceptual model) depending on the parameters defined in the strategy phase. Finally, in this step, the identification of data movements of the COSMIC measurement phase is also performed.

4.3.1 Functional Users and Boundaries

The *functional users* are users that interact with the system being measured. These users are the types of users that send (or receive) data to (from) the functional processes of a system. A boundary is a conceptual interface between the functional users and the pieces of software being measured [Abran *et al.* 2007].

In the OO-Method applications, the human user interacts with the application sending and receiving data to the functional processes. Also, the pieces of software that compose the application interact with the remainder pieces of software of the application. For the OO-Method applications, it is

possible to specify the legacy systems that interact with the system modelled. Thus, we identify the following functional users for OO-Method applications: the human users, the client piece of the software, the server piece of the software, and the legacy views specified in the conceptual model. These functional users are separated by a boundary from the pieces of software of the application (see Figure 4.4).

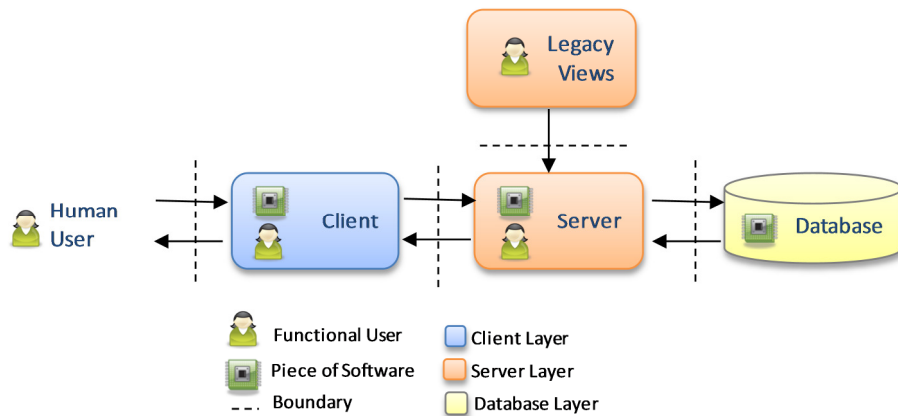


Figure 4.4 Functional users and boundaries of an OO-Method application.

The users of the OO-Method applications are represented in the conceptual model as agent classes of the object model (i.e. Agents are classes that can access specific attributes or services of other classes of the model). These users are generally human users that send or receive data to the client layer of the software. From now on, we refer to this type of users as the *'human functional user'*. This user is a functional user of the client layer of the software and is separated by a boundary from the client layer of the software. To identify the human functional user and the corresponding boundary, we have defined the following rules:

Rule 1: Identify a human functional user for each agent class in the OO-Method object model.

Rule 2: Identify one boundary between the human functional user and the client layer.

The client component of the software is a functional user that sends and receives data to (from) the server piece of software. We refer to this type of user as the '*client functional user*'. This user is a functional user of the server layer of the software and is separated by a boundary from the server layer of the software. To identify the client functional user and the corresponding boundary, we have defined the following rules:

Rule 3: Identify a client functional user for the client component of an OO-Method application.

Rule 4: Identify one boundary between the client functional user and the server layer.

The server component of the software sends and receives data to (from) the client layer of the software and the database layer of the software. We refer to this type of user as the '*server functional user*'. This user is a functional user for both layers of the software: the client layer and the database layer. The server functional user is separated by a boundary from the client layer of the software, and by a boundary from the database layer of the software. To identify the server functional user and the corresponding boundary, we have defined the following rules:

Rule 5: Identify a server functional user for the server component of an OO-Method application.

Rule 6: Identify one boundary between the server functional user and the database layer.

The OO-Method conceptual model allows the definition of legacy views (which are classes implemented in other systems) as well as the definition of their relationships with the classes of the system. The legacy views can enter values to the server layer of the software. Thus, legacy views represent functional users of the OO-Method applications and are separated by a boundary from the server layer of the software. We refer to this type of user as the '*legacy functional user*'. To identify the legacy functional user and the corresponding boundary, we have defined the following rules:

Rule 7: Identify a legacy functional user for each legacy view in the OO-Method object model.

Rule 8: If exists a legacy functional user, identify one boundary between the legacy functional user and the server layer.

4.3.2 Functional Processes

In general terms, a functional process corresponds to a set of Functional User Requirements comprising a unique, cohesive, and independently executable set of data movements [Abran *et al.* 2007]. A functional process starts with an entry data movement carried out by a functional user given that an event (triggering event) has happened. A functional process ends when all the data movements needed to generate the answer to this event have been executed. Thus, a functional process has at least two data movements (1 entry/read data movement + 1 exit/write data movement).

In the context of OOmCFP, the 'human functional user', the 'client functional user', the 'server functional user', and the 'legacy functional user' start functional processes.

Functional Processes started by the Human Functional User

The ‘human functional user’ carries out the triggering events that occur in the real world. This functional user starts the functional processes that occur in the client layer of the application. In this layer, the functional processes are represented by the interaction units of the OO-Method presentation model that can be directly accessed by the ‘human functional user’.

These interaction units correspond to the direct successors of the hierarchy action tree (HAT) of the presentation model of the OO-Method conceptual model. Therefore, every child of the HAT will be one functional process, representing either a selection of a given class population (a Population Interaction Unit (PIU)), an execution of a service (a Service Interaction Unit (SIU)), or more complex interaction units (such as a Master Detail Interaction Unit (MDIU)).

To identify the functional processes that occur in the client layer of an OO-Method application, we have defined the following rule:

Rule 9: Identify a functional process in the client layer for each Population Interaction Unit (PIU), Service Interaction Unit (SIU) or Master-Detail Interaction Unit (MDIU) that is a direct child of the hierarchy action tree (HAT) of the presentation model of the OO-Method conceptual model.

Nevertheless, in order to correctly identify the functionality that the interaction units have, it is important to identify the elements that compose the interaction unit in addition to it, since they are also relevant to the measurement of the functional size.

The functional processes that correspond to PIUs can contain display sets, actions, navigations, filters, and order criteria. In turn, actions, navigations and filters can contain interaction units that also make up the corresponding functional processes. In order to calculate the functional size

of a functional process that corresponds to a PIU, it must be decomposed in presentation patterns and interaction units. Thus, to identify the elements contained in a functional process that corresponds to a PIU we have defined the following rule:

Rule 9.1: For each PIU, identify the Display Set, Action Set, Navigation Set, Filter, Order Criteria, and the interaction units that are contained in these patterns.

The functional processes that correspond to SIUs can contain arguments (using entry patterns, selection patterns, dependency rules, or pre-charge patterns) and conditional navigations. The arguments and conditional navigations can also contain interaction units. Thus, we have defined the following rule to identify the elements that are contained in a functional process that correspond to a SIU:

Rule 9.2: For each SIU, identify the arguments, the Conditional Navigations, and the interaction units that are contained in these patterns.

The functional processes that correspond to MDIUs are comprised of a master part and a detail part. In turn, these parts can be comprised of instance interaction units (IIU), population interaction units (PIU), or other master detail interaction units (MDIU). To facilitate the identification of the elements that compose the MDIUs that have been identified as functional processes, we have defined the following rule:

Rule 9.3: For each MDIU, identify the master part and the detail part, and the interaction units that are contained in these patterns.

When a functional process contains IIUs, the elements contained in the IIU must also be identified for the functional process. These elements are display sets, actions, and navigations. In turn, actions, and navigations can contain other interaction units. Thus, we have defined the following rule to identify the elements contained in the IIUs:

Rule 9.4: For each IIU, identify the Display Set, Action Set, Navigation Set, and the interaction units that are contained in these patterns.

Therefore, in order to completely identify a functional process, once Rule 9 has been applied, then Rules 9.1, 9.2, 9.3, and 9.4 must be applied iteratively until terminating with the identification of all the elements contained in the functional process. Table 8 summarizes the elements that can be contained in the interaction units and the interaction units that are transitively contained.

Table 8. Presentation elements contained in the interaction units.

Interaction Units (IU)	Contained Elements	Contained IUs
Instance Interaction Unit (IIU)	Display Set	-
	Action Set	SIU, IIU, PIU, MDIU
	Navigation Set	IIU, PIU, MDIU
Population Interaction Unit (PIU)	Display Set	-
	Action Set	SIU, IIU, PIU, MDIU
	Navigation Set	IIU, PIU, MDIU
	Filter	PIU
	Order Criteria	-
Master Detail Interaction Unit (MDIU)	Master	IIU, PIU
	Detail	IIU, PIU, MDIU
Service Interaction Unit (SIU)	Arguments	PIU
	Conditional Navigation	SIU, IIU, PIU, MDIU

Functional Processes started by the Client Functional User

The 'client functional user' activates *triggering events* that occur in the interaction units of the presentation model of the OO-Method conceptual model. The 'client functional user' starts *functional processes*, which are the actions that carry out the server layer of the software in response to the triggering events that occur in the client layer of the software.

To identify the elements that are contained in the functional processes of the server layer, it is necessary to identify the actions that this layer performs in response to the triggering events carried out by the interactions units of the client layer. These triggering events are the following:

- The Instance Interaction Units (IIU) require from the server layer the values of the attributes that compose a display set, the execution of a service, or the default values of the arguments of a service.
- The Population Interaction Units (PIU) require from the server layer the values of the attributes that compose a display set, the values of the filter variables that have defined a default value, the execution of a service, or the default values of the arguments of a service.
- The Master Detail Interaction Units (MDIU) is comprised of combinations of IIUs or PIUs. Therefore, they require from the server layer the information that the IIUs or the PIUs require.
- The Service Interaction Units (SIU) require from the server layer the default values of the arguments of a service, the values of the derived attributes used in a service, and the initialization of arguments. SIUs also require from the server layer the execution of the following: services associated to the interaction units, dependency rules of the arguments, valuations, conditional navigation for the success or failure cases of the execution of a service, navigational filtering, validation of the preconditions of a service, check integrity constraints, triggers activated by a service,

or the change of state of an object by means of the transitions of a service.

Therefore, we have defined the following rule to identify the functional processes of the server layer:

Rule 10: A functional process corresponds to the set of formulae (derivations, default values, filters, valuations, integrity constraints, triggers, transactions, preconditions, dependency rules, control conditions, and conditional navigation) that solve the Server layer in response to the events that occur in the functional processes of the Client layer.

We have decided that the functional processes of the server layer have the same name of the functional process that contains the interaction units where the triggering events occur. Thus, we have defined the following rule to name the functional processes that occur in the server layer:

Rule 10.1: For each functional process in the server layer, name it using the name of the client functional process that started it.

Functional Processes started by the Server Functional User

The 'server functional user' carries out the *triggering events* that occur in the server layer of the software. The 'server functional user' starts *functional processes*, which are the actions that the database layer carries out in response to the triggering events of the server layer, and the actions that the client layer carries out in response to triggering events of the server layer of the software.

The database layer adds, edits, and deletes the persistent information of the system. To do this, the database layer receives triggering events from the server layer of a system. Since the OO-Method applications use commercial Database Management Systems (DBMS), which has functionality that is not generated from the OO-Method model, the OOmCFP measurement procedure does not consider the functional processes that occur in the database layer for the calculation of the functional size of OO-Method applications. However, OOmCFP considers the communication with the database layer that is performed to write or read data by means of the services executed by the functional processes of the server layer.

On the other hand, the ‘server functional user’ starts functional processes in the client layer that deliver information by means of display sets, default values of filter variables and arguments, or error messages. These conceptual elements are contained in the interaction units defined in the client layer, so that, they are identified in the functional process that occur in the client layer since they cannot be executed outside the interaction units. Therefore, OOmCFP does not consider different functional processes for these elements.

Functional Processes started by the Legacy Functional User

The ‘legacy functional user’ activates *triggering events* that occur in the legacy system. The ‘legacy functional user’ starts *functional processes*, which are the actions that the server layer of the software carries out to interact with the legacy system. These actions correspond to the set of formulae that the server layer solves in response to requests of the legacy system.

Since the legacy views represent external systems (pre-existing software) that interacts with the system modelled using OO-Method, some changes must be developed outside the models to connect both systems. Thus, when the system modeled needs information of the legacy system, it uses the

attributes defined in the legacy views specified in the model. In contrast, when the legacy system needs some information of the system modeled, it recovers the information using its own services. Taking into account that the information needed by a legacy system is not specified in the model, it is not possible to infer the functional processes that the 'legacy functional user' starts. Hence, we have not defined rules to identify these functional processes.

Eliminate Duplicity in the Identification of Functional Processes

It is important to note that the interaction units that already have been identified as a functional process must not be identified as a contained element since they will be analyzed separately. Thus, we have defined the following rule to avoid counting the interaction units that correspond to a functional process inside other functional process:

Rule 11: Do not consider in the functional size of a functional process FP_B the functional size of a functional process FP_A that is contained in the functional process FP_B.

Many times, in order to facilitate to the user the interaction with the functionality of an application, the functional processes have many ways to go to the same functionality (for instance, a contained interaction unit can be accessed by an action or a navigation). However, the functional size must be counted only one time. So that, we have defined the following rule to avoid the duplicate counting of the functionality of the interaction units contained in a functional process:

Rule 12: Only consider one time the functional size of an IIU, PIU, MDIU or SIU that is auto contained.

4.3.3 Data Groups

The *data groups* correspond to a set of different attributes that describe an object of interest. The object of interest corresponds to physical objects, conceptual objects, or even parts of conceptual objects.

Taking into account that the OOmCFP uses the OO-Method conceptual model to measure the functional size, the data groups are the classes of the object model of the OO-Method conceptual model that participate in a functional process. Of course, identified data groups always correspond to a conceptual object of interest.

Nevertheless, when a class is part of an inheritance hierarchy, the parent class corresponds to a data group, and when the child class has different attributes than the parent class, it will correspond to another data group. Thus, to correctly identify the data groups, we have defined the following rules:

Rule 13: Identify a data group for each class that is not part of an inheritance hierarchy in the object model that participates in a functional process.

Rule 14: Identify a data group for the parent class of an inheritance hierarchy in the object model of a class that participates in a functional process belongs to.

Rule 15: Identify a data group for each child class that has different attributes than his parent in an inheritance hierarchy in the object model of a class that participates in a functional process belongs to.

4.3.4 Data Attributes

The *data attributes* correspond to the smallest pieces of information of a data group. In the context of OOmCFP, the *data attributes* of a data group correspond to the attributes of the classes that have been identified as a data group. We have defined the following rule to identify the attributes:

Rule 16: Identify a data attribute for each attribute of the classes in the object model that are identified as data groups.

4.3.5 Data Movements

With regard to the identification of *data movements*, every functional process has a set of *data movements* that can be entry data movements (E), exit data movements (X), read data movements (R) or write data movements (W). Each single data movement must move a single data group. An *entry data movement* is a movement that moves one data group from a functional user across the boundary into the functional process where it is required. An *exit data movement* is a movement that moves a data group from a functional process across the boundary to the functional user that requires it. A *read data movement* is a movement that moves a data group from the persistence storage that is in contact with the functional process that requires it. A *write data movement* is a movement that moves a data group from a functional process to the persistence storage.

Figure 4.5 shows the data movements that could occur in the OO-Method applications. In this figure, it is possible to observe that the human functional user enters and receives data from the client layer of the software, the client layer and the server layer interchanges data to each other, only the server layer reads and writes the database, and only the server layer receives data from legacy systems.

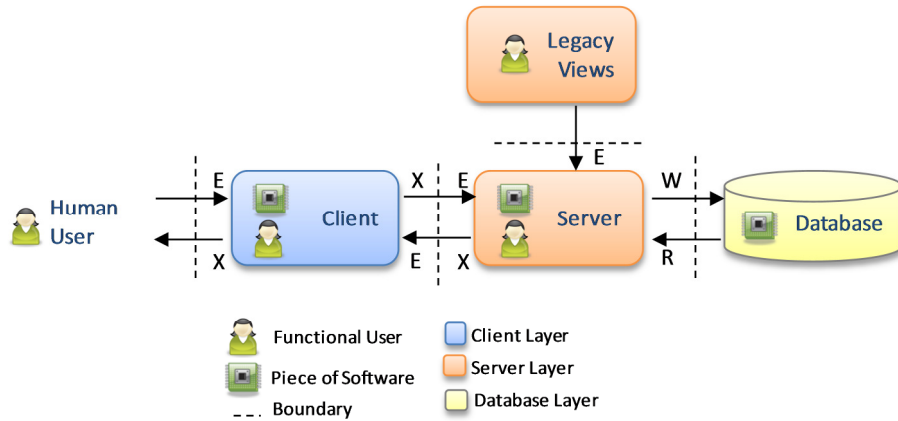


Figure 4.5 Data movements between users and layers of an OO-Method application.

We have defined 74 counting rules for the identification of the data movements that occur in the OO-Method applications. All these counting rules are structured with a concept of the COSMIC measurement method, a concept of the OO-Method approach, and the cardinalities that associate these concepts. Thus, these counting rules detect the data movements of all the functionality needed for the correct operation of the generated application, which is built by the model compiler of the MDD method. Since all the data movements of the generated applications can be identified focusing in three main conceptual constructs (display sets, filters, and services), the counting rules are grouped by these constructs.

Data Movements in Display Sets

A display set presents the information of the system to the ‘human functional user’. To do this, the attributes that will be shown by the display set must be specified in the presentation model. Once the application has been generated,

the following data movements occur in the display set if it does not contain derived attributes (see Figure 4.6):

- 1) The server layer reads from the database the values of the attributes that will be shown by the display set.
- 2) If the display set has attributes that are specified in a legacy view, the server layer receives the values for these attributes from the legacy views.
- 3) The server layer delivers the values to the client layer.
- 4) The client layer receives the values from the server layer.
- 5) The client layer displays the values to the 'human functional user'.

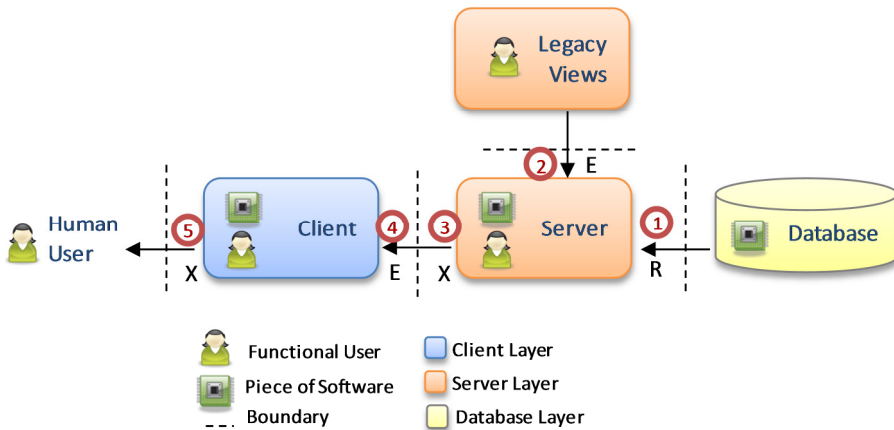


Figure 4.6 Data movements in a Display Set without derived attributes.

In order to correctly identify the data movements that occur in a display set, we have defined the following rules:

- Counting Rule 1: 1 **read** data movement for each different *class* that contributes with *attributes* to the display set of a PIU or IIU that participates in a functional process of the **server layer**.

Counting Rule 2: 1 **entry** data movement for each different *legacy view* that contributes with *attributes* to the display set of a PIU or IIU that participates in a functional process in the **server layer**.

Counting Rule 3: 1 **exit** data movement for each different *class* or *legacy view* that contributes with *attributes* to the display set of a PIU or IIU that participates in a functional process in the **server layer**.

Counting Rule 4: 1 **entry** data movement for each different *class* or *legacy view* that contributes with *attributes* to the display set of a PIU or IIU that participates in a functional process of the **client layer**.

Counting Rule 5: 1 **exit** data movement for *all* the *attributes* that are shown in a display set of a PIU or IIU that participates in a functional process of the **client layer**.

However, if the display set contains derived attributes, the following data movements occur (see Figure 4.7):

- 1) The server layer reads from the database the values of the attributes that are used to calculate the condition of the derivation.
- 2) If the derivation condition has legacy views, the server layer receives the values of the attributes that are used to calculate the condition of the derivation from the legacy views.
- 3) If the derivation condition is true, the server layer reads from the database the values of the attributes that are used to calculate the derivation value.

- 4) If the derivation value has legacy views, and the derivation condition is true, the server layer receives the values of the attributes that are used to calculate the derivation value from the legacy views.
- 5) The server layer reads from the database the values of the not derived attributes that will be shown by the display set.
- 6) The server layer delivers the values to the client layer.
- 7) The client layer receives the values from the server layer.
- 8) The client layer displays the values to the 'human functional user'.

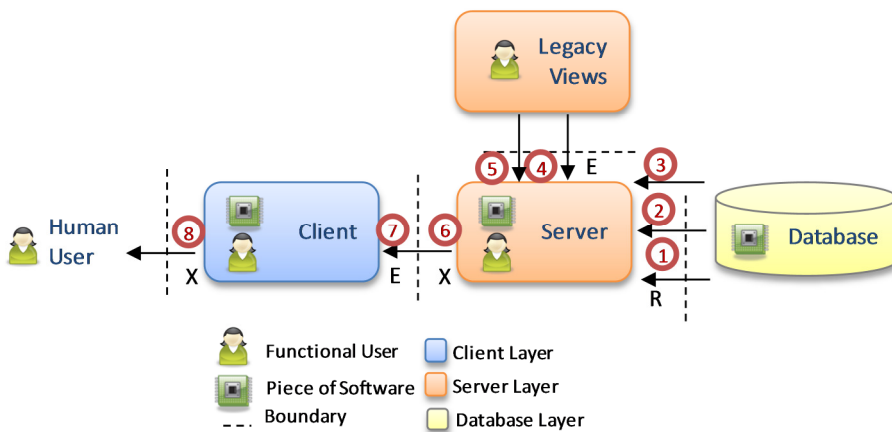


Figure 4.7 Data movements in a Display Set with derived attributes.

We have defined extra rules to identify the data movements that occur in a display set that has derived attributes. These rules are the following:

Counting Rule 6: 1 **read** data movement for each different *class* that is used in the *derivation formula* of the derived attributes of the display set of a PIU or IIU that participates in a functional process in the **server layer**.

Counting Rule 7: 1 **read** data movement for each different *class* that is used in the *condition of the derivation formula* of the derived attributes of the display set of a PIU or IIU that participates in a functional process in the **server layer**.

Counting Rule 8: 1 **entry** data movement for each different *legacy view* that is used in the *derivation formula* of an attribute of a display set of a PIU or IIU that participates in a functional process in the **server layer**.

Counting Rule 9: 1 **entry** data movement for each different *legacy view* that is used in the *condition of a derivation formula* of an attribute of a display set of a PIU or IIU that participates in a functional process in the **server layer**.

Data Movements in Filters

A filter allows restricting the set of instances that will be shown by the system to the ‘human functional user’ using conditions over the values of a set of filter variables previously specified by the ‘human functional user’. To do this, a filter for the involved class must be specified in the presentation model. In this definition, the filter condition and the filter variables must be specified. The filter variables can be data-valued or object-valued, and also, they can have a default value specified.

Once the application has been generated, the following data movements occur in a filter that has filter variables with a default value specified (see Figure 4.8):

- 1) The ‘human functional user’ enters the adequate values for the filter variables.

- 2) The client layer delivers the values entered by the ‘human functional user’ to the server layer.
- 3) The server layer receives the values for the filter variables.
- 4) The server layer reads from the database the information that is necessary to solve the filter formula.
- 5) If a legacy view is used in the filter formula, the legacy system delivers the corresponding values to solve the filter formula to the server layer.

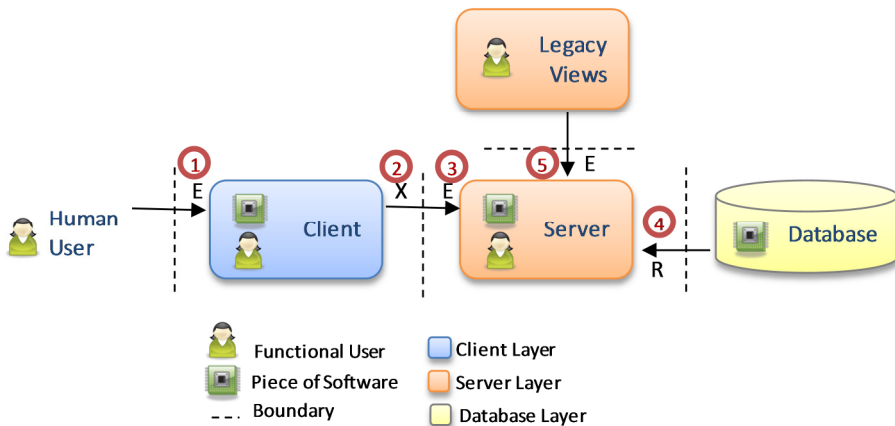


Figure 4.8 Data movements in a Filter.

We have defined the following rules to identify the data movements that occur in a filter:

Counting Rule 10: 1 **entry** data movement (represented by the class that contains the filter) for the *set of data-valued variables* that are associated to a filter of a PIU that participates in a functional process of the **client layer**.

- Counting Rule 11: 1 **entry** data movement for each different *object-valued variables* that is associated to a filter of a PIU that participates in a functional process in the **client layer**.
- Counting Rule 12: 1 **exit** data movement (represented by the class that contains the filter) for the *set of data-valued variables* that are associated to a filter of a PIU that participates in a functional process in the **client layer**.
- Counting Rule 13: 1 **exit** data movement for each different *object-valued variable* that is associated to a filter of a PIU that participates in a functional process in the **client layer**.
- Counting Rule 14: 1 **entry** data movement (represented by the class that contains the filter) for the *set of data-valued variables* that are associated to a filter of a PIU that participates in a functional process in the **server layer**.
- Counting Rule 15: 1 **entry** data movement for each different *object-valued variable* that is associated to a filter of a PIU that participates in a functional process in the **server layer**.
- Counting Rule 16: 1 **read** data movement for each different *class* that is used in the *filter formula* of a filter of a PIU that participates in a functional process in the **server layer**.

Counting Rule 17: 1 **entry** data movement for each different *legacy view* that is used in the *filter formula* of a filter of a PIU that participates in a functional process in the **server layer**.

However, if the filter has defined default variables for its variables, then the following data movements occur in a filter (see Figure 4.9):

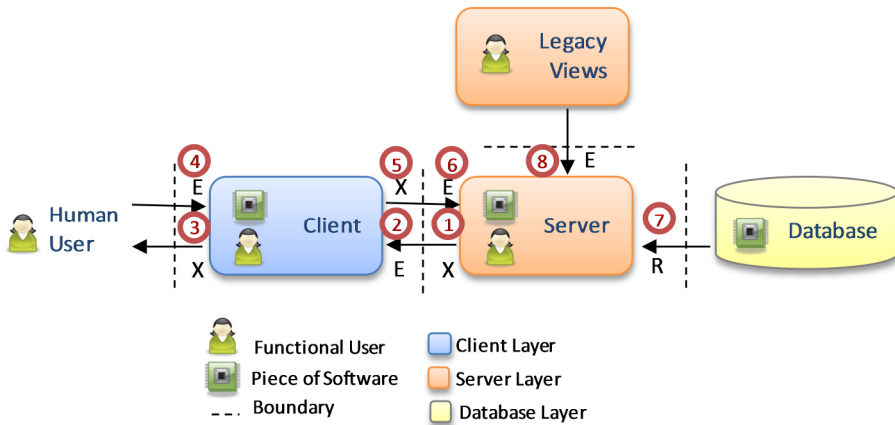


Figure 4.9 Data movements in a Filter with variables that have default values specified.

- 1) The server layer calculates the default values of the data-valued and object-valued filter variables using constants and functions that are inside the server layer, and deliver the default values to the client functional user.
- 2) The client layer receives the values for the filter variables from the server layer.
- 3) The client layer displays the values of the filter variables to the ‘human functional user’.

- 4) The 'human functional user' observes the default values displayed in the filter variables and enters the adequate values for these variables.
- 5) The client layer delivers the values entered by the 'human functional user' to the server layer.
- 6) The server layer receives the values for the filter variables.
- 7) The server layer reads from the database the information that is necessary to solve the filter formula.
- 8) If a legacy view is used in the filter formula, the legacy system delivers the corresponding values to solve the filter formula to the server layer.

Thus, we have defined the following extra rules to identify the data movements that occur in the filter variables that have default values specified:

Counting Rule 18: 1 **exit** data movement for each different *class* that is used in the formula of the *default value* of an *object-valued variable* that is associated to a filter of a PIU that participates in a functional process in the **server layer**.

Counting Rule 19: 1 **exit** data movement (represented by the class that contains the filter) for the *set of data-valued variables* that has a *default value*, and that are associated to a filter of a PIU that participates in a functional process in the **server layer**.

Counting Rule 20: 1 **entry** data movement for the *default value* of an *object-valued variable* that is associated to a filter of a PIU that participates in a functional process in the **client layer**.

Counting Rule 21: 1 **entry** data movement (represented by the class that contains the filter) for the *set of data-valued variables* that has a *default value*, and that are associated to a filter of a PIU that participates in a functional process in the **client layer**.

Counting Rule 22: 1 **exit** data movement (represented by the class that contains the filter) for the *set of data-valued variables* that has a *default value*, and that are associated to a filter of a PIU that participates in a functional process in the **client layer**.

Counting Rule 23: 1 **exit** data movement for the *default value* of an *object-valued variable* that is associated to a filter of a PIU that participates in a functional process in the **client layer**.

Data Movements in Services

A service allows changing the state of an object. To do this, a set of services are defined in each class of the object model. The arguments (data-valued or object-valued), the preconditions, the valuations for the events, and the service formula for the transactions and operations must be specified for each service. Also, in the class that contains the services, the integrity constraints (conditions that must be satisfied by all the objects of a class at any state) must be specified.

Once the application has been generated, different kinds of data movements may occur depending of the conceptual constructs that the service has and the satisfaction of the conditions that these conceptual constructs reach. If a service has preconditions, the following data movements occur for each precondition (Figure 4.10):

4. Design of a Functional Size Measurement Procedure

- 1) The server layer reads from the database the values that are necessary to solve the precondition of the service.
- 2) If a legacy view is used in the precondition formula, the legacy system delivers the corresponding values to solve the precondition formula to the server layer.
- 3) The server layer calculates the value of the precondition. If the precondition is fulfilled, the server layer continues with the following precondition repeating step 1 and 2 until it finishes the analysis of all the preconditions.

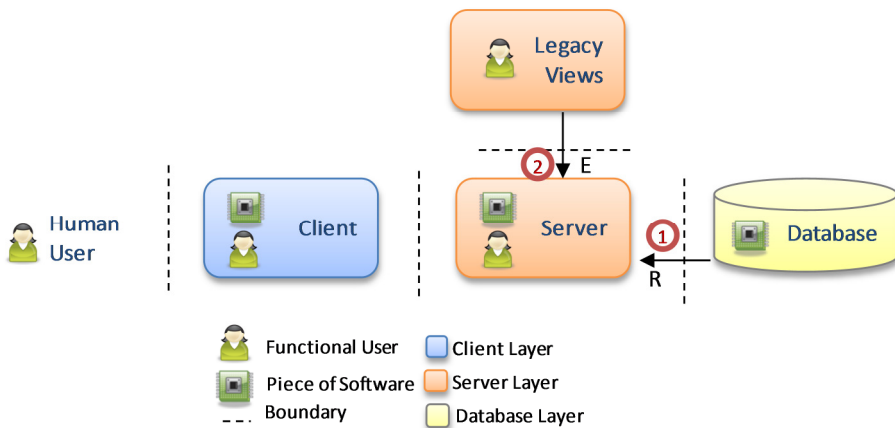


Figure 4.10 Data movements in a Service with preconditions that are fulfilled.

We have defined the following rules to identify the data movements that occur in the preconditions of a service:

Counting Rule 24: 1 **read** data movement for each different *class* that is used in the *formula of the preconditions* of a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 25: 1 **entry** data movement for each different *legacy view* that is used in the *formula of the preconditions* of a SIU that participates in a functional process in the **server layer**.

However, for a service that has a precondition that is not fulfilled, the following data movements occur (see Figure 4.11):

- 1) If the precondition is not fulfilled, the server layer reads from the database the values that are necessary to solve the error formula of the precondition of the service.
- 2) If a legacy view is used in the error formula, the legacy system delivers the corresponding values to solve the error formula to the server layer.
- 3) The server layer calculates the value of the error formula and delivers it to the client layer.
- 4) The client layer receives the value and generates an error message.
- 5) The client layer displays the error message to the ‘human functional user’.

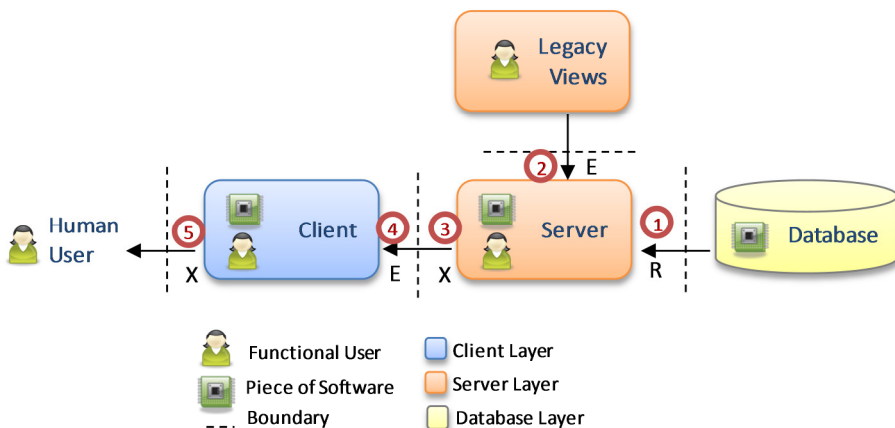


Figure 4.11 Data movements in a Service with preconditions that are not fulfilled.

We have defined the following rules to identify the data movements that occur in the error messages of the preconditions of a service:

Counting Rule 26: 1 **read** data movement for each different *class* that is used in the *formula of the error messages* associated to the *preconditions* of a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 27: 1 **entry** data movement for each different *legacy view* that is used in the *formula of the error messages* associated to the *preconditions* of a SIU that participates in a functional process in the **server layer**.

Counting Rule 28: 1 **exit** data movement for each different *class* or *legacy view* that is used in the *formula of the error messages* associated to the *preconditions* of a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 29: 1 **entry** data movement for each different *class* or *legacy view* that is used in the *formula of the error messages* associated to the *preconditions* of a service related to a SIU that participates in a functional process in the **client layer**.

Counting Rule 30: 1 **exit** data movement for all the *error messages* associated to the *preconditions* of a service related to a SIU that participates in a functional process in the **client layer**.

Taking into account that a service can be an event, a transaction, an operation, or a global service; the following data movements occur for a service (see Figure 4.12):

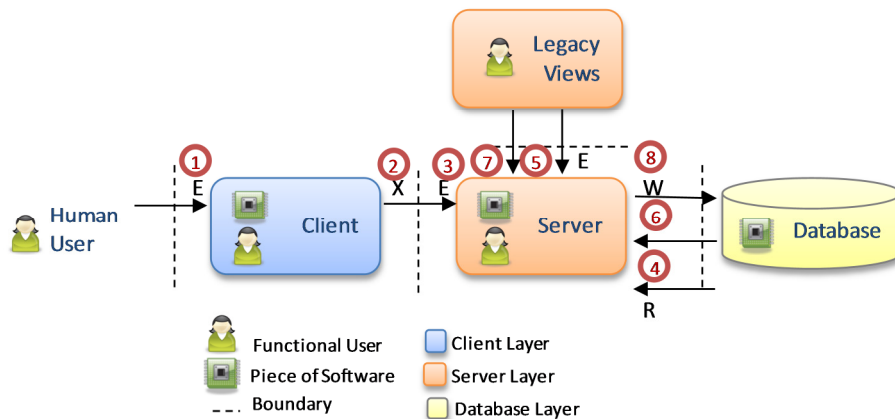


Figure 4.12 Data movements in a Service.

- 1) The ‘human functional user’ enters the adequate values for the arguments of the service.
- 2) The client layer delivers the values entered by the ‘human functional user’ to the server layer.
- 3) The server layer receives the values for the arguments of the service.
- 4) If the service is an event with a valuation specified, the server layer reads from the database the information that is necessary to solve the condition of the valuation.
- 5) If a legacy view is used in the condition of the valuation formula, the legacy system delivers the corresponding values to solve the condition of the valuation formula to the server layer.
- 6) The server layer reads from the database the information that is necessary to solve the service formula (valuation, transaction formula, operation formula, or global formula).

4. Design of a Functional Size Measurement Procedure

- 7) If a legacy view is used in the service formula, the legacy system delivers the corresponding values to solve the service formula to the server layer.
- 8) If the service is an event for creation, destruction, carrier, liberator or it has valuations, the server layer writes in the database.

We have defined the following rules to identify the data movements that occur in a service:

Counting Rule 31: 1 **entry** data movement (represented by the class that contains the SIU) for the set of *data-valued arguments* of a SIU that participates in a functional process in the **client layer**.

Counting Rule 32: 1 **entry** data movement for each different *class* that corresponds to an *object-valued argument* of a SIU that participates in a functional process in the **client layer**.

Counting Rule 33: 1 **exit** data movement (represented by the class that contains the SIU) for the set of *data-valued arguments* of a SIU that participates in a functional process in the **client layer**.

Counting Rule 34: 1 **exit** data movement for each different *class* that corresponds to an *object-valued argument* of a SIU that participates in a functional process in the **client layer**.

Counting Rule 35: 1 **entry** data movement (represented by the class that contains the SIU) for the set of *data-valued*

arguments of a SIU that participate in a functional process in the **server layer**.

Counting Rule 36: 1 **entry** data movement for each different *class* that corresponds to an *object-valued argument* of a SIU that participates in a functional process in the **server layer**.

Counting Rule 37: 1 **read** data movement for each different *class* that is used in the *condition of the valuation formula* of the event related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 38: 1 **read** data movement for each different *class* that is used in the *valuation formula* of the event related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 39: 1 **read** data movement for each different *class* that is used in the *formula of the transaction, operation or global service* related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 40: 1 **entry** data movement for each different *legacy view* that is used in the *condition of the valuation formula* of a SIU that participates in a functional process in the **server layer**.

Counting Rule 41: 1 **entry** data movement for each different *legacy view* that is used in the *valuation formula* of a SIU that

participates in a functional process in the **server layer**.

Counting Rule 42: 1 **entry** data movement for each different *legacy view* that is used in the *formula of the transaction, operation or global service* related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 43: 1 **write** data movement for the *class* that contains a *destroy event* or a *liberator event* related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 44: 1 **write** data movement for the *class* that contains a *creation event* of a *carrier event* related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 45: 1 **write** data movement for the *class* that contains an *event that has valuations* and that is related to a SIU that participates in a functional process in the **server layer**.

Also, for a service that has default values specified for its entry arguments, the following data movements occur (see Figure 4.13):

- 1) The server layer calculates the default values of the data-valued and object-valued entry arguments of the service using constants and functions that are inside the server layer, and deliver the default values to the client functional user.

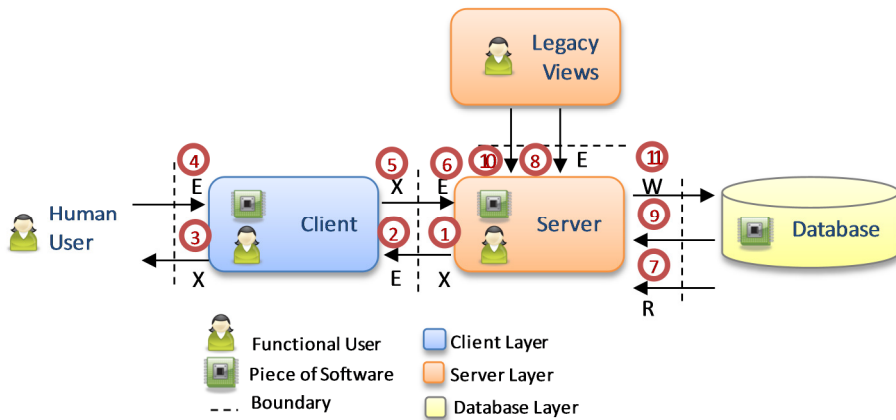


Figure 4.13 Data movements in a Service with default values for its entry arguments.

- 2) The client layer receives the values for the arguments that have a default value specified from the server layer.
- 3) The client layer displays the values of the arguments to the ‘human functional user’.
- 4) The ‘human functional user’ observes the values of the arguments and enters the adequate values for the arguments of the service.
- 5) The client layer delivers the values entered by the ‘human functional user’ to the server layer.
- 6) The server layer receives the values for the arguments of the service.
- 7) If the service is an event with a valuation specified, the server layer reads from the database the information that is necessary to solve the condition of the valuation.
- 8) If a legacy view is used in the condition of the valuation formula, the legacy system delivers the corresponding values to solve the condition of the valuation formula to the server layer.
- 9) The server layer reads from the database the information that is necessary to solve the service formula (valuation, transaction formula, operation formula, or global formula).

- 10) If a legacy view is used in the service formula, the legacy system delivers the corresponding values to solve the service formula to the server layer.
- 11) If the service is an event for creation, destruction, or it has valuations, the server layer writes in the database.

Thus, we have defined the following extra rules to identify the data movements that occur in a service that has default values specified for its arguments:

Counting Rule 46: 1 **exit** data movement for each different *class* that is used in the formula of the *default value* of an *object-valued argument* that is associated to a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 47: 1 **exit** data movement (represented by the class that contains the SIU) for the set of *data-valued argument* that has a *default value* and that are associated to a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 48: 1 **entry** data movement for each different *class* that is used in the formula of the *default value* of an *object-valued argument* that is associated to a service related to a SIU that participates in a functional process in the **client layer**.

Counting Rule 49: 1 **entry** data movement (represented by the class that contains the SIU) for the set of *data-valued argument* that has a *default value* and that are associated to a

service related to a SIU that participates in a functional process in the **client layer**.

Counting Rule 50: 1 **exit** data movement (represented by the class that contains the SIU) for the set of *data-valued argument* that has a *default value* and that are associated to a service related to a SIU that participates in a functional process in the **client layer**.

Counting Rule 51: 1 **exit** data movement for the *default value of an object-valued argument* that is associated to a service related to a SIU that participates in a functional process in the **client layer**.

If the class that contains the service has integrity constraints defined, the following data movements occur for each integrity constraint after the execution of the service (see Figure 4.14):

- 1) The server layer reads from the database the values that are necessary to solve the integrity constraint of the service.
- 2) If a legacy view is used in the integrity constraint formula, the legacy system delivers the corresponding values to solve the integrity constraint formula to the server layer.
- 3) The server layer calculates the value of the integrity constraint. If the integrity constraint is fulfilled, the server layer continues with the following integrity constraint repeating step 1 and 2 until it finishes the analysis of all the integrity constraints.

We have defined the following rules to identify the data movements that occur in the integrity constraints of the class that contains a service:

Counting Rule 52: 1 **read** data movement for each different *class* that is used in the *formula of the integrity constraints* of a class that contains a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 53: 1 **entry** data movement for each different *legacy view* that is used in the *formula of the integrity constraints* of a class that contains a service related to a SIU that participates in a functional process in the **server layer**.

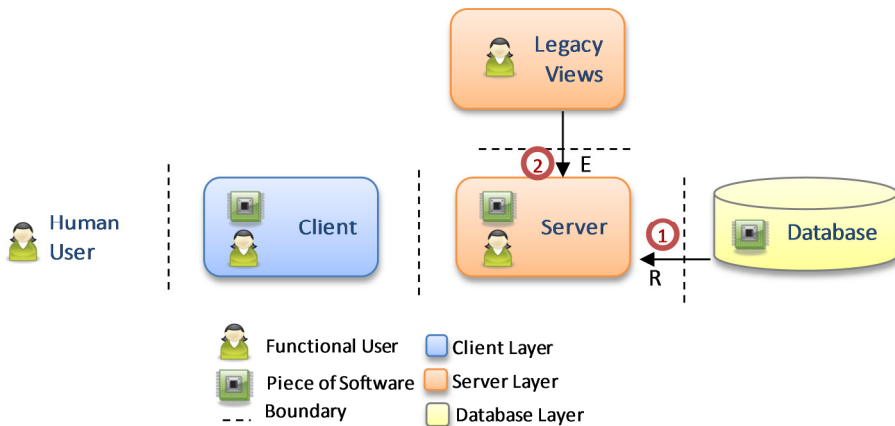


Figure 4.14 Data movements in a Service of a class with integrity constraints that are fulfilled.

But, when the integrity constraints of the class that contains the service are not fulfilled, the following data movements occur (see Figure 4.15):

- 1) If the integrity constraint is not fulfilled, the server layer reads from the database the values that are necessary to solve the error formula of the integrity constraint.

- 2) If a legacy view is used in the error formula, the legacy system delivers the corresponding values to solve the error formula to the server layer.
- 3) The server layer calculates the value of the error formula and delivers it to the client layer.
- 4) The client layer receives the value and generates an error message.
- 5) The client layer displays the error message to the 'human functional user'.

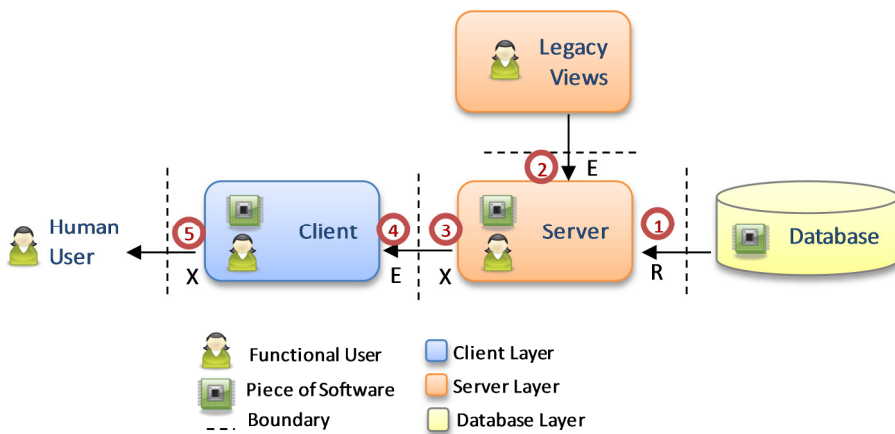


Figure 4.15 Data movements in a Service of a class with an integrity constraint that is not fulfilled.

Thus, we have defined the following rules to identify the data movements that occur in the error messages of the integrity constraints of the class that contains the service:

Counting Rule 54: 1 **read** data movement for each different *class* that is used in the *formula of the error messages* associated to the *integrity constraints* of a class that contains a

service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 55: 1 **entry** data movement for each different *legacy view* that is used in the *formula of the error messages* associated to the *integrity constraints* of a class that contains a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 56: 1 **exit** data movement for each different *class* or *legacy view* that is used in the *formula of the error messages* associated to the *integrity constraints* of a class that contains a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 57: 1 **entry** data movement for each different *class* or *legacy view* that is used in the *formula of the error messages* associated to the *integrity constraints* of a class that contains a service related to a SIU that participates in a functional process in the **client layer**.

Counting Rule 58: 1 **exit** data movement for all the *error messages* associated to the *integrity constraints* of a class that contains a service related to a SIU that participates in a functional process in the **client layer**.

In the dynamic model of the system, it is possible to specify restrictions to the execution of the services specified in the object model. Thus, in the state transition diagram it is possible to specify a condition that must be

previously fulfilled to allow the execution of the service. Therefore, the following data movement occur in a service with a control condition (see Figure 4.16):

- 1) The server layer reads from the database the values that are necessary to solve the control condition of the service.
- 2) If a legacy view is used in the control condition, the legacy system delivers the corresponding values to solve the control condition to the server layer.
- 3) The server layer calculates the value of the control condition. If the condition is fulfilled, the server layer executes the service performing all the data movements specified for the service and its conceptual constructs.

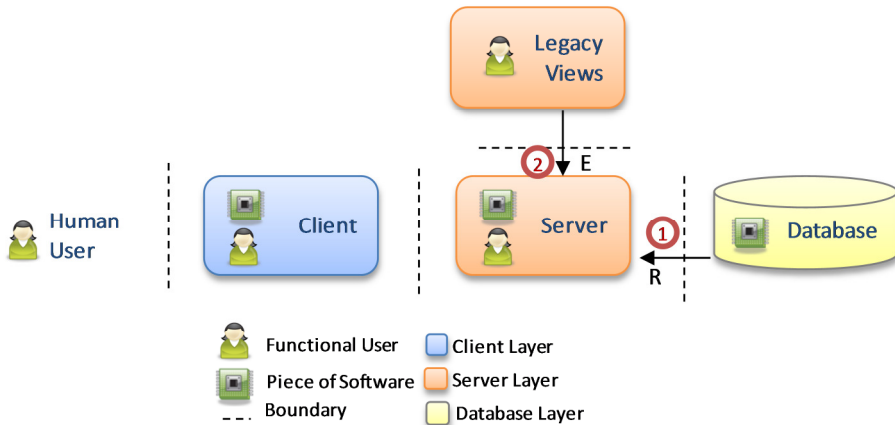


Figure 4.16 Data movements in a Service with a control condition.

Thus, we have defined the following rules for the services that have control conditions specified in the state transition diagram:

Counting Rule 59: 1 **read** data movement for each different *class* that is used in the *formula of the control condition of a*

service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 60: 1 **entry** data movement for each different *legacy view* that is used in the *formula of the control condition* of a service related to a SIU that participates in a functional process in the **server layer**.

In the object interaction diagram of the dynamic model, it is possible to specify the services that will be automatically executed by means of triggers when a condition of the class that contains the service is fulfilled. These conditions are evaluated when a service has been executed. Thus, the following data movements occur for each trigger specified in the class that contains the executed service (see Figure 4.17):

- 1) The server layer reads from the database the values that are necessary to solve the condition of the trigger that are related to the class that contains the service.
- 2) If a legacy view is used in the condition of a trigger, the legacy system delivers the corresponding values to solve the trigger condition to the server layer.
- 3) When the condition of the trigger is fulfilled, the server layer triggers the service specified performing all the data movements specified for the service and its conceptual constructs.
- 4) If there are more triggers specified, the server layer continues with the following trigger repeating step 1 and 2 until it finishes the analysis of all the triggers.

Thus, we have defined the following rules for the services that are contained in a class that has triggers specified:

Counting Rule 61: 1 **read** data movement for each different *class* that is used in the *condition formula of the triggers* of the class that contains a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 62: 1 **entry** data movement for each different *legacy view* that is used in the *condition formula of the triggers* of the class that contains a service related to a SIU that participates in a functional process in the **server layer**.

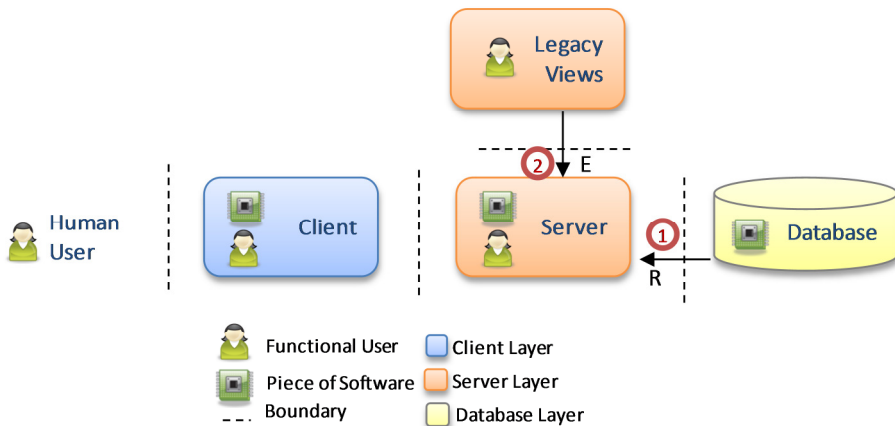


Figure 4.17 Data movements in a Service that is contained in a class with triggers specified.

In the presentation model of a system, it is possible to specify the value of an argument when an interface event occurs in other argument of a service by means of Event-Condition-Action (ECA) rules. To do this, dependency rules are specified in the arguments of a service. The following data

movements occur for each argument that has dependency rules specified (see Figure 4.18):

- 1) The server layer reads from the database the information that is necessary to solve the condition of the dependency rule.
- 2) If a legacy view is used in the condition of the dependency rule, the legacy system delivers the corresponding values to solve the condition of the dependency rule to the server layer.
- 3) The server layer reads from the database the information that is necessary to solve the dependency formula.
- 4) If a legacy view is used in the dependency formula, the legacy system delivers the corresponding values to solve the dependency formula to the server layer.

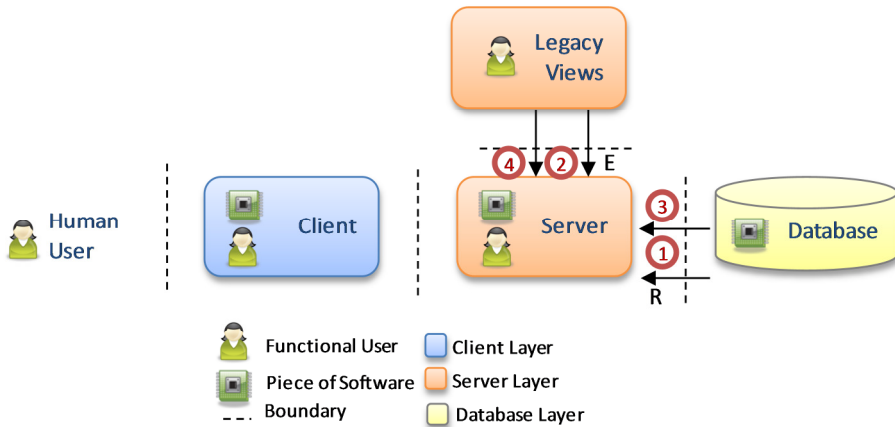


Figure 4.18 Data movements in a Service with arguments that have defined dependency rules.

We have defined the following rules to identify the data movements that occur in the arguments of a service that have dependency rules specified, which has a SIU related:

Counting Rule 63: 1 **entry** data movement for each different *legacy view* that is used in the *action formulae of the dependency rules* of the arguments of a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 64: 1 **entry** data movement for each different *legacy view* that is used in the *condition formulae of the dependency rules* of the arguments of a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 65: 1 **read** data movement for each different class that is used in the *formulae of the dependency rules of the arguments* of a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 66: 1 **read** data movement for each different class that is used in the *condition formulae of the dependency rules of the arguments* of a service related to a SIU that participates in a functional process in the **server layer**.

In addition, in the presentation model of the system, it is possible to specify the graphical interface that will be shown when the execution of a service ends. To do this, conditional navigations are specified. A conditional navigation allows the specification of the interaction units that must be displayed to the 'human functional user' when a condition is fulfilled after the execution of a service, whether be success or failure. If a SIU is specified in the conditional navigation, it is possible to specify the values of its entry

arguments using the initialization of arguments pattern. Thus, once the application has been generated, the following data movements occur for each condition (success or failure) that has a SIU related in the conditional navigation (see Figure 4.19):

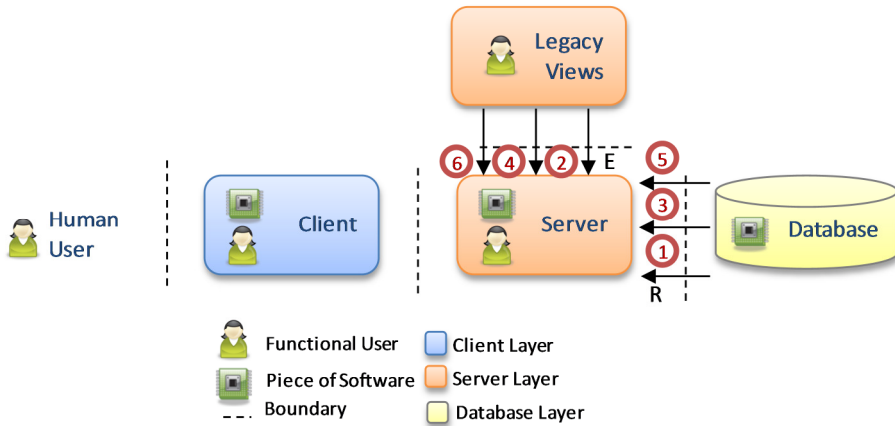


Figure 4.19 Data movements in a Service with a conditional navigation fulfilled and an initialization of an argument.

- 1) The server layer reads from the database the values that are necessary to solve the condition of the conditional navigation.
- 2) If a legacy view is used in the conditional navigation, the legacy system delivers the corresponding values to solve the conditional navigation to the server layer.
- 3) The server layer calculates the value of the condition of the conditional navigation. If the condition of the conditional navigation is fulfilled, the server layer reads from the database the values that are necessary to calculate the condition of the initialization of arguments of the SIU.
- 4) If a legacy view is used in the condition of the initialization of arguments, the legacy system delivers the corresponding values to

solve the condition of the initialization of arguments to the server layer.

- 5) The server layer calculates the value of the condition of the initialization of arguments. If the condition of the initialization of arguments is fulfilled, the server layer reads from the database the values that are necessary initialize for the arguments of the SIU.
- 6) If a legacy view is used in the initialization of arguments, the legacy system delivers the corresponding values to solve the initialization of arguments to the server layer.
- 7) The server layer executes the SIU performing all the data movements specified for a service and its conceptual constructs.

We have defined the following rules to identify the data movements that occur in the conditional navigations of a service that have a SIU related:

Counting Rule 67: 1 **read** data movement for each different *class* that is used in the *conditional navigation formula* of a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 68: 1 **entry** data movement for each different *legacy view* that is used in the *conditional navigation formula* of a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 69: 1 **read** data movement for each different *class* that is used in the *condition of the formula of the arguments initialization* of a SIU associated to the conditional navigation of a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 70: 1 **entry** data movement for each different *legacy view* that is used in the *condition of the formula of the arguments initialization* of a SIU associated to the conditional navigation of a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 71: 1 **read** data movement for each different *class* that is used in the *formula of the arguments initialization* of a SIU associated to the conditional navigation of a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 72: 1 **entry** data movement for each different *legacy view* that is used in the *formula of the arguments initialization* of a SIU associated to the conditional navigation of a service related to a SIU that participates in a functional process in the **server layer**.

If an IIU, a PIU, or a MDIU are specified in the conditional navigation, it is possible to specify the set of instances that will be shown in the corresponding IIU, PIU, or MDIU using the navigational filtering pattern. Thus, the following data movements occur for each condition (success or failure) that has an IIU, PIU, or MDIU related in the conditional navigation (see Figure 4.20):

- 1) If the condition of the conditional navigation is fulfilled, the server layer reads from the database the values that are necessary to filter the instances of the IIU, PIU, or MDIU.

- 2) If a legacy view is used in the navigational filtering formula, the legacy system delivers the corresponding values to solve the filter to the server layer.
- 3) The server layer executes the IIU, PIU, or MDIU performing all the data movements specified for its conceptual constructs.

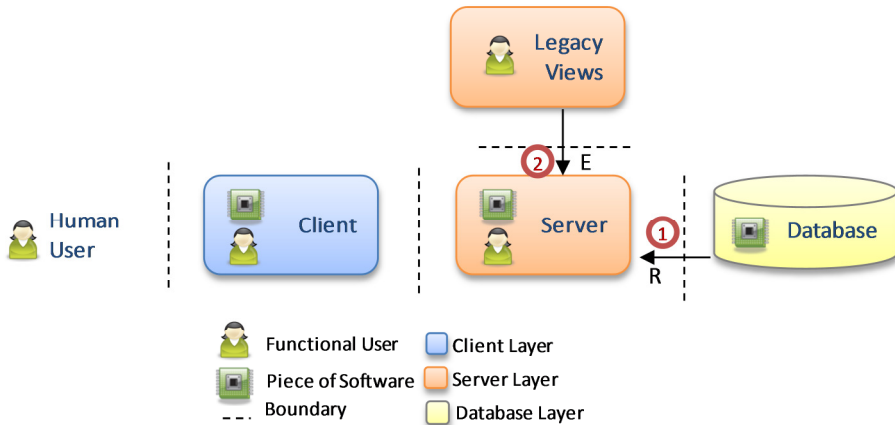


Figure 4.20 Data movements in a Service with a conditional navigation fulfilled and a navigational filtering.

Thus, we have defined extra rules to identify the data movements that occur in the navigational filtering of conditional navigations of a service:

Counting Rule 73: 1 **read** data movement for each different *class* that is used in the *formula of the navigational filtering* of an IIU, PIU, or MDIU associated to the conditional navigation of a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 74: 1 **entry** data movement for each different *legacy view* that is used in the *formula of the navigational filtering* of an IIU, PIU, or MDIU associated to the conditional navigation of a service related to a SIU that participates in a functional process in the **server layer**.

4.4 Definition of the Numerical Assignment Rules

The measurement rules are rules that assign a numerical value to the data movements that take place between the functional users and the pieces of software of an application. This numerical value represents the functional size of the generated software application.

In this step, the definition of the measurement function and the aggregation of the results are performed. With this, the COSMIC measurement phase ends (see Figure 4.1), which implies that the design of the OOmCFP measurement procedure is complete.

4.4.1 Measurement Function

According to the COSMIC functional size measurement method, each data movement will be assigned one size unit, which is referred to as *1 CFP* (COSMIC Function Point).

In the OOmCFP procedure, the measurement function is identical to the COSMIC measurement function. That is, OOmCFP assigns 1 CFP to each data movement.

4.4.2 Aggregation of Results

To measure the functional size of a functional process, the functional size of all the data movements of the functional process should be added. Formula (1) illustrates how to calculate the functional size of a functional process:

$$\text{SizeFunctionalProcess (CFP v3.0)} = \sum_{i=1}^n \text{DataMovement}_i \quad (1)$$

Taking into account that the measurement function assigns 1 CFP to each data movement, the result of Formula 1 corresponds to the functional size that is measured in CFPs. From Formula 1 it can be observed that the functional size is expressed in CFP v3.0. We have defined the following rules to aggregate the data movements of each functional process of an OO-Method application:

Measurement Rule 1: Aggregate the *data movements* of a *functional process* that occur in the *client layer* to obtain the **functional size** of the functional process.

Measurement Rule 2: Aggregate the *data movements* of a *functional process* that occur in the *server layer* to obtain the **functional size** of the functional process.

Once all the functional processes are measured, the functional size of these processes should be added to obtain the functional size of the layer that contains them. Formula (2) shows how to calculate the functional size of a layer:

$$\text{SizeLayer (CFP v3.0)} = \sum_{i=1}^n \text{SizeFunctionalProcess}_i \quad (2)$$

Thus, we have defined the following rules to obtain the functional size of each layer:

Measurement Rule 3: Aggregate the **functional size** of each *functional process* of the client layer to obtain the functional size of the **client layer**.

Measurement Rule 4: Aggregate the **functional size** of each *functional process* of the server layer to obtain the functional size of the **server layer**.

For the measurement of the generated software applications from the developer's viewpoint, it is necessary to aggregate the functional size of every layer of the software application. This calculation is represented in formula (3).

$$\text{SizeOOMethodApplication (CFP v3.0)} = \sum_{i=1}^n \text{SizeLayer}_i \quad (3)$$

We have defined the following rule to obtain the functional size of an entire OO-Method application:

Measurement Rule 5: Aggregate the **functional size** of each *layer* of the OO-Method application to obtain the functional size of the **application**.

Finally, with the measurement rules, it is possible to measure the functional size of the OO-Method applications that are generated from their conceptual models in a MDD environment. Thus, the measurement result of OOmCFP corresponds to the functional size of OO-Method applications. The measurement rules add all the data movements that occur in the application for its correct operation; so that, the measurement result includes all the functionality from the developer's viewpoint.

4.5 Conclusions

In this chapter, we have presented OOmCFP, which is a FSM procedure for applications that are generated from object-oriented conceptual models in MDD environments. This procedure was designed in accordance with the COSMIC standard method, which is a FSM method that allows the measurement of each layer of an application as well as the whole application. This is in contrast to other FSM standard methods that only allow the measurement of the whole application.

OOmCFP has been systematically designed applying a set of steps of a software measurement process model. Since there is no consensus in the concepts and terminology used in the software measurement field, the concepts and terminology used in the design of OOmCFP have been carefully selected from the standards ISO VIM [ISO 2004], ISO 14143 series [ISO 1998] [ISO 2003] and ISO 19761 [ISO/IEC 2003a].

In the design of OOmCFP, we have identified a set of counting rules that allow the relevant constructs of the OO-Method conceptual model to be selected according to the COSMIC concepts. Moreover, a set of measurement rules has been defined to obtain the functional size of each software layer of an application. Even though OOmCFP has been designed to be used specifically in the OO-Method context, many conceptual constructs of the OO-Method conceptual model can be found in other object-oriented methods. Moreover, the main modelling constructs used by OO-Method are basic constructs that have UML representation support. That is why the OOmCFP procedure could be generalized to other object-oriented development methods and the presented results can be applied to any UML-based method where those constructs are present (for instance [Fink *et al.* 2006], [Kim *et al.* 2004], [Olivé and Raventós 2006]).

The OOmCFP procedure has been designed to obtain accurate measures of the application that is generated from the conceptual models. This is feasible because we have selected a conceptual model that has enough

4. Design of a Functional Size Measurement Procedure

semantic expressiveness to specify all the functionality of the final application. Therefore, the measurement results obtained can be accurate because all the data movements that occur in the final application could be traceable to the conceptual model.

Chapter 5

Evaluation of the Design of OOmCFP

Hundreds of measures have been proposed in the literature for different software attributes, such as size, complexity, maintainability, usability, etc. Many of them have been defined using intuitive approaches, resulting in that there is no precise definition of the concepts involved in the measures. Taking into account that a valid measure is that one that effectively measures the concepts that it needs to measure, the majority of the measures found in the literature have not been validated properly.

In order to validate software measures, many validation approaches have been proposed (such as [Kitchenham *et al.* 1995], [Schneidewind 1992], [Zuse 1998], etc.), but none of which has been widely used by designers or users of the software measures [Sellami and Abran 2003]. In addition, the lack of consensus in the concepts of the measurement field has caused that different concepts are used to refer to the same things, or even the same concept is used to refer to different things.

There exist some proposals defined to harmonize these concepts and provide a consistent terminology, such as the software measurement

ontology (SMO) [García *et al.* 2006] and a framework for the verification of software measurement methods [Habra *et al.* 2008]. Both, the SMO and the framework are mainly based in the ISO VIM concepts. The SMO is comprised of four sub-ontologies: (1) software measurement characterization and objectives, (2) software measures, (3) measurement, and (4) measurement approaches. OOmCFP is aligned with the first three sub-ontologies because it uses the concepts that they define. However, the measurement approach sub-ontology classifies a measurement approach in ‘measurement method’, ‘measurement function’, and ‘analysis model’. We do not agree with this classification because a measurement method can have a measurement function defined, as the COSMIC measurement method. For this reason, OOmCFP is not aligned with the measurement approach sub-ontology of SMO.

The framework for the verification of measurement methods [Habra *et al.* 2008] defines a set of concepts and terminology for the design and verification of the measurement methods. These concepts are similar to the concepts of the SMO, for instance the framework defines the concepts of measure, measurement result, unit of measurement, entity, and attribute. Thus, OOmCFP is aligned with the terms and concepts defined in the framework. In contrast to the SMO, the framework defines a classification for measurement methods and measurement procedures. OOmCFP is characterized by this classification. The framework also defines the life cycle of the measurement methods: design, application, and exploitation.

Even though there are approaches that attempt to harmonize the concepts used by the measurement field, they neither are widely used and sometimes they disagree with the standards (such as the SMO [García *et al.* 2006]). Taking into account this context, this chapter presents the validation of the design of the OOmCFP FSM procedure. This validation has been carried out by using the following standards: the ISO/IEC 14143-2 standard of the conformity evaluation of software size measurement methods [ISO 2002], the International vocabulary of basic and general terms in metrology (VIM)

[ISO 2004], and the ISO 5725-2 – Accuracy (trueness and precision) of Measurements Methods and Results [ISO 1994].

5.1 Conformity Evaluation of OOmCFP

The conformity evaluation of OOmCFP refers to the evaluation of the OOmCFP FSM procedure according to the last version of the COSMIC FSM method [Abran et al. 2007]. This means that the design of OOmCFP is analyzed to know if all the essential concepts of COSMIC are addressed by OOmCFP. To perform the conformity evaluation of OOmCFP, the following elements were necessary:

1. Candidate FSM procedure documentation. The documentation shall include all the materials needed to the proper use of the FSM procedure, for instance, manuals, guidelines, examples, case studies, etc. For the conformity evaluation of OOmCFP, this material corresponds to the measurement guide of OOmCFP (see Appendix A).
2. Conformity evaluation checklist. This checklist consists in 29 questions that is used to evaluate the candidate FSM procedure with the COSMIC FSM method [Abran *et al.* 2007]. The checklist is structured in four parts: questions related to the strategy phase of COSMIC, questions related to the mapping phase of COSMIC, questions related to the measurement phase of COSMIC, and question related to the presentation of the measurement results. The checklist used to evaluate OOmCFP is presented in Appendix B.
3. Evaluators. In order to increment the objectivity of the conformity evaluation, an expert certified in COSMIC FSM v. 3.0 was responsible to verify the evaluation performed.

5. Evaluation of the Design of OOmCFP

The conformity evaluation of OOmCFP was performed using the process presented in the ISO/IEC 14143-2 standard of the conformity evaluation of software size measurement methods [ISO 2002] (see Figure 5.1). Thus, for each question of the checklist, evaluators decide if it is applicable to the conformity evaluation or not. For the questions that are applicable to the evaluation, evaluators locate the information related to the question in the documentation of OOmCFP. If there is enough information to satisfy the requirements related to the question, evaluators register the question in the checklist as satisfied. If the information does not satisfy the requirements, evaluators mark it as unsatisfied and also justify their decision in the checklist. Finally, if there is no enough information to solve the question, evaluators mark it as unresolved and justify their decision.

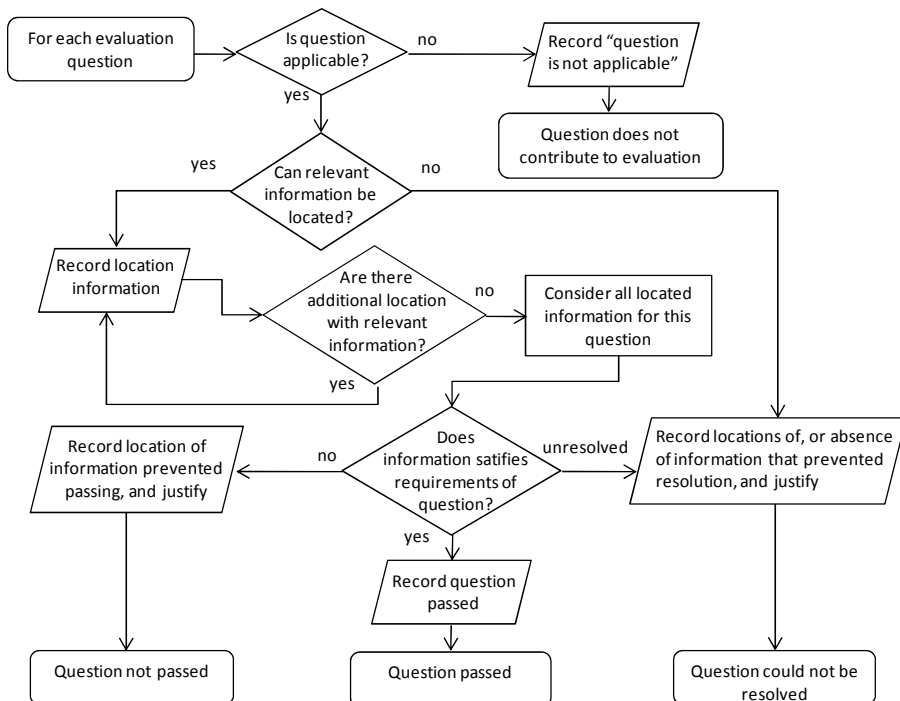


Figure 5.1 Conformity evaluation process for OOmCFP.

Results of the conformity evaluation of OOmCFP indicate that all the questions (10) of the strategy phase were approved by OOmCFP, all the questions (7) of the mapping phase were approved by OOmCFP, 8 of 10 questions of the measurement phase were approved by OOmCFP, and all the questions (2) of the measurement report were approved by OOmCFP. In summary, 27 questions of the conformity evaluation checklist were approved by OOmCFP and 2 questions of the measurement part of the checklist could not be resolved due to the absence of information.

One of these unresolved questions is related to the definition of an aggregation function to obtain the functional size of data movements' modifications. This definition was not necessary in OOmCFP since it obtains the functional size of OO-Method applications from their conceptual models, and when a change in the software is required, it is necessary to first change the model, and then, re-generate completely the application. Thus, to obtain the functional size of the modifications it is necessary to calculate the functional size of the applications with the changes and subtract to it the functional size of the original application.

The other unresolved question is related to the definition of extensions to obtain the functional size. In OOmCFP, it was not necessary to define extensions to the COSMIC measurement method to obtain the size of OO-Method applications.

Therefore, we can state that the OOmCFP measurement procedure version 1.0 conforms to the concepts and principles of the COSMIC measurement method version 3.0 (2007).

5.2 Metrology Evaluation of the Design of OOmCFP

Metrology is defined in the VIM [ISO 2004] as “*field of knowledge concerned with measurement*”. Metrology includes theoretical and practical

aspects of measurement. It has a long tradition of use in sciences such as physics or chemistry, but it is rarely referred in software engineering [Abran 2010].

The International vocabulary of basic and general terms in metrology (VIM) [ISO 2004] represents the official consensus in measurement concepts. It has 122 terms grouped in five categories: Quantities and Units (26 terms), Measurement (43 terms), Devices for Measurements (12 terms), Characteristics of Measurement Systems (23 terms), and Measurement Standards-Etalons (18 terms). In contrast to traditional dictionaries, this vocabulary presents the terms in a textual format following a complexity order: from the less complex term to the most complex term.

To facilitate the understanding of the terms specified in VIM and clarify their relationships with the remainder terms, a high-level model of the terms and categories was presented [Abran and Sellami 2002] (see Figure 5.2). In this figure, the concepts are represented as follows: the input is *Measurand*, the output is *Measurement Result*, the process itself is *Measurement*, the control variables are *Quantities and Units* and *Measurement Standards-Etalons*, the set of concepts is represented as *Devices for Measurements*, and the operations of the measurement are influenced by the *Characteristics of Measurement Systems*.

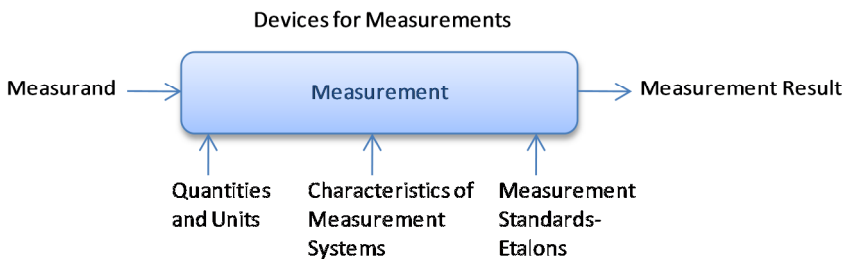


Figure 5.2 High-level model of categories of metrology terms [Abran and Sellami 2002].

Taking into account these sets of metrology concepts, the quality criteria that must be accomplished for the design of a measurement method have been specified in the book *Software Metrics and Software Metrology* [Abran 2010]. Thus, the design of a measurement method is considered ‘good’ when it refers to these quality criteria. Table 9 presents these criteria for the design step of the process model that was followed to define the OOmCFP measurement procedure. It is important to consider that the measurement set of concepts is comprised by both the measurement foundation and the measurement procedure. And, to validate the design, only the measurement foundation must be taken into account.

Table 9. Quality criteria for the design of a measurement method.

Step in Process Model	Quality Criteria
Design of the Measurement Method	Measurement Foundation
	Quantities and Units
	Measurement Standards-Etalons

In addition, each category of VIM was analyzed in order to identify sub-concepts within the terms of the category. In order to clarify these sub-concepts, [Abran and Sellami 2002] also modeled the categories and organized topologically the involved concepts. The analysis of the design of OOmCFP following these topologies that groups metrology concepts is presented below.

5.2.1 Measurement Foundation

The measurement foundation corresponds to a hierarchy of related measurement concepts. This hierarchy is defined from the most general to the specific (top-down). Figure 5.3 shows the levels of the measurement foundation [Abran 2010].

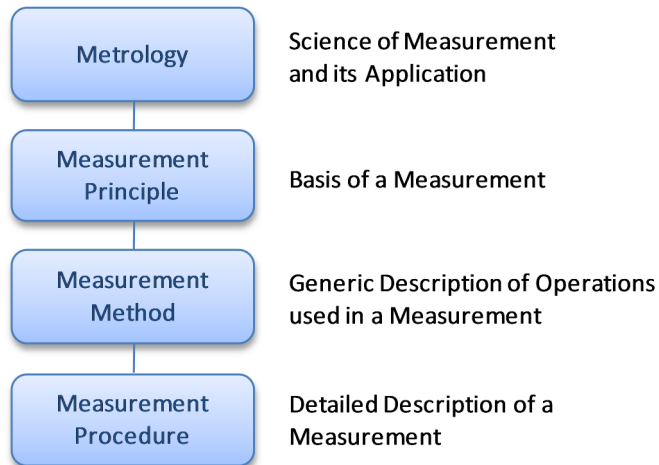


Figure 5.3 Levels of the Measurement Foundation.

Metrology includes all the theoretical and practical aspects on measurement. The measurement principle represents the phenomenon that serves as the basis of a measurement. The measurement method is a generic description of a logical sequence of operations used in a measurement. The measurement procedure is the instantiation of the measurement method; it has a detailed description of the measurement and includes the calculations needed to obtain a measurement result.

The hierarchy of the measurement foundation has been taken into account in the design of OOmCFP (see Figure 5.4). The OOmCFP has been designed according to the Standard ISO 19751 [ISO/IEC 2003a], therefore, the measurement principle states that the functional size is directly proportional to the number of data movements. The measurement method corresponds to the COSMIC Functional Size Measurement Method version 3.0 [Abran *et al.* 2007], and the instantiation of this method to apply it in the conceptual models of the OO-Method approach correspond to the OOmCFP Functional Size Measurement Procedure.

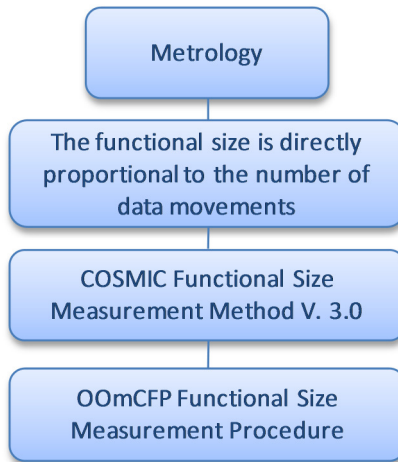


Figure 5.4 Measurement Foundations for OOmCFP.

5.2.2 Quantities and Units

A quantity is defined as *a property to which a magnitude can be assigned* [ISO 2004]. Normally, a quantity is expressed as numbers. However, to recognize a quantity as a measure, the following additional concepts are necessary: system of quantities, dimension of a quantity, unit of measurement, value of a quantity, kind of quantity, and the quantity calculus. Figure 5.5 shows the model of this category (adapted from [Abran 2010]).

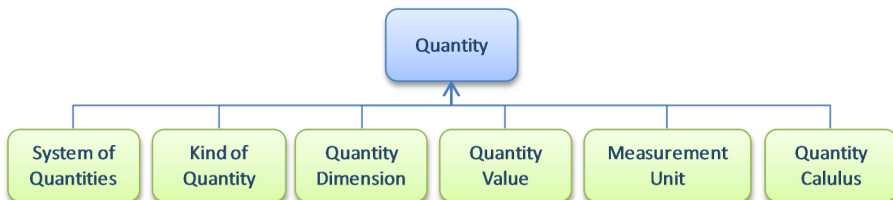


Figure 5.5 High-level model of Quantity category.

The system of quantities set of concepts include the following terms: base quantities, derived quantities, and international system of quantities [Abran 2010]. A base quantity is a quantity chosen by convention that is used to define other quantities. A derived quantity is a function of base quantities. An international system of quantities (ISQ) is a system of quantities that has a set of quantities and equations that relate those quantities. Currently, the ISQ is published in the International Standard of Quantities and Units [ISO 1992]. The OOmCFP FSM procedure is focused only in the measurement of the functional size, which is considered a base quantity. Some derived quantities related to the functional size are the productivity and the budget, but they are not addressed by the OOmCFP FSM procedure. The definition of an international system of quantities is not applicable for OOmCFP.

The concept of a quantity can be divided in two levels: a general concept and an individual concept when it is applied to the object under consideration. The kind of a quantity corresponds to a common aspect in comparable quantities. Thus, the kind of quantity refers to the general concept of a quantity (for instance, length). For OOmCFP, all the quantities correspond to the same kind: the functional size.

The quantity dimension is represented by the product of the powers of factors, where the factors are base quantities. For OOmCFP, the quantity dimension is one-dimensional since it only has one base quantity.

The quantity value is a magnitude of a quantity that is represented by a number and a reference. It has related the concepts of true value, conventional true value, numerical value, and conventional reference scale [Abran and Sellami 2002]. The true value is the value that would be obtained with perfect measuring instruments and without committing any error of any type [OECD 2010]. In OOmCFP, this value is considered only as a theoretical concept since several external factors can affect the measurements. Nevertheless, OOmCFP considers the conventional true value as the value obtained by experts applying the COSMIC Functional

Size Measurement Method. The numerical value corresponds to the value obtained by the application of the measurement function defined in the COSMIC measurement manual version 3.0. [Abran *et al.* 2007]. The conventional reference scale in OOmCFP corresponds to the set of discrete numbers, which its minimum value is 1 data movement and its maximum value is not determined. Since 1 data movement in OOmCFP is related to one data group, the conventional reference scale is expressed in data group movements.

The measurement unit is a scalar quantity, defined and adopted by convention, with which other quantities of the same kind are compared in order to express their magnitudes. Its definition also includes the definition of the symbol. The measurement unit set of concepts has related the terms of base unit, derived unit, coherent derived unit, system of units, coherent system of units, off-system measurement unit, international system of units (SI), multiple of a unit, and sub-multiple of a unit.

A base unit is a unit chosen by convention for a base quantity of a system of quantities. In OOmCFP, the measurement unit is a base unit that corresponds to a data movement, which uses the symbol “CFP”. A derived unit is a unit for a derived quantity. Since OOmCFP do not consider derived quantities, there are no derived units defined in OOmCFP. Thus, the system of units for OOmCFP only has the base unit stated before.

A coherent derived unit is a unit obtained by the by the product of the powers of base units with the proportionally factor. The coherent system of units is a system comprised of coherent derived units. For OOmCFP is not necessary to define neither coherent derived units nor the coherent system of units nor the off-system measurement unit. The international system of units (SI) is a coherent system of units based on the ISQ. The definition of an international system of units is not applicable for OOmCFP.

The multiple of a unit is a unit formed from a unit by multiplying it by an integer greater than one; and the sub-multiple of a unit is a unit formed from a unit by dividing it by an integer greater than one. Since in OOmCFP

the measurement unit correspond to a data movement that moves a single data group, to define a multiple of a unit, it is necessary to find a conceptual construct that be make up by data groups (for instance, packages). Also, to define a sub-multiple of a unit, it is necessary to find a conceptual construct that be contained in a data group (for instance, attributes). However, it is not possible to define multiples and sub-multiples of the measurement unit of OOmCFP since there are not a specific number of packages or attributes related to a data group.

The quantity calculus corresponds to the set of algebraic rules applied to quantities. This set of concepts has the following related terms: quantity equation, unit equation, conversion factor between units, and numerical value equation [Abran 2010]. A quantity equation refers to an equation relating quantities. In OOmCFP there are three equations defined: (1) one equation to obtain the functional size of each functional process that adds the functional size obtained for each conceptual construct contained in a functional process, (2) one equation to obtain the functional size of each layer of the application that adds the functional size of the functional processes contained in a layer, and (3) one equation to obtain the functional size of the entire application generated from the conceptual model measured, which adds the functional size of the layers that comprise the final application.

The unit equation corresponds to an equation of the units of the quantities that are used in a quantity equation. Since the equations defined in OOmCFP are sums of data movements, it is not necessary to define a unit equation. The conversion factor between units is a ratio of two units for quantities of the same kind. In OOmCFP, there is only a quantity defined (i.e., the functional size) that is related to a particular measurement unit, so that conversion factors between units have not been defined.

The numerical value equation is an equation relating numerical quantity values. Thus, when the OOmCFP equations are instantiated with

corresponding numbers of data movements or functional size, they correspond to numerical value equations.

5.2.3 Measurement Standards - Etalons

A measurement standard or etalon is defined as a material measure, measuring instrument, reference material or measuring system intended to define, realize, conserve or reproduce a unit or one or more values of a quantity to serve as a reference. Figure 5.6 shows the model of this category.

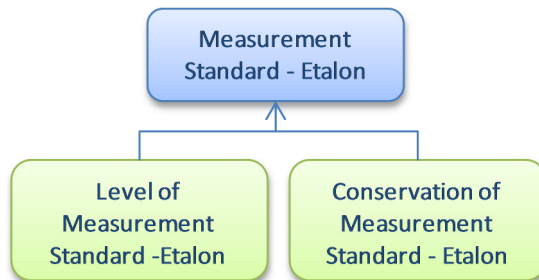


Figure 5.6 High-level model of Measurement Standards - Etalons category.

There are several concepts related to the different levels of measurement standards, such as international/national measurement standard, primary/secondary measurement standard, reference measurement standard, working measurement standard, transfer measurement standard, travelling measurement standard, and intrinsic measurement standard [Abran 2010]. The design of OOmCFP has been performed according to the last version of the International Standard Functional Size Method ISO 19761 [ISO/IEC 2003a]. This standard is also a primary standard since its quantity value and measurement uncertainty have been established without relation to another measurement standard for quantities of the same kind. Since the ISO 19761 is a widely recognized basis for OOmCFP, it was not necessary to use other

measurement standards in the design of OOmCFP (i.e., secondary, reference, working, transfer, travelling, or intrinsic measurement standards were not necessary).

There are also several concepts related to the conservation of a measurement standard, such as calibrator, reference material, certified reference material, commutability of a reference material, reference data, standard reference data, and reference quantity value [Abran 2010]. Since OOmCFP is not a standard but it is defined according to an international primary standard, these concepts related to the conservation of the measurement standard are not addressed by OOmCFP due to they should be addressed by the ISO/IEC 19761. Consequently, in the literature it is possible to find reference material for etalons for the ISO/IEC 19761 [Khelifi 2005] and methodology concepts to define a measurement standard etalon for the COSMIC Functional Size method [Khelifi and Abran 2007].

Finally, from the analysis of OOmCFP according to the measurement foundation, the quantities and units, and the measurement standards – etalons sets of concepts, we can state that the design of the OOmCFP measurement procedure is valid according to the metrology concepts.

5.3 Precision Evaluation of OOmCFP

Software measurement currently plays a crucial role in software engineering given that the evaluation of software quality depends on the values of the measurements carried out. Software measurements must therefore have quality attributes that assure measurement reliability, such as accuracy and precision.

Accuracy is related to the closeness to the ‘true value’ of the measurements [ISO 2003]. Since the ‘true value’ is only a theoretical concept, when a measurement is performed it is important to obtain the value

for the measurement and the estimation of the degree of uncertainty. However, the degree of uncertainty of a software measurement is very difficult to obtain due to there are external factors that affect measurements. Thus, it is essential to first obtain precise measures in order to obtain accuracy measures.

The International Standard for Accuracy (trueness and precision) of Measurements Methods and Results - ISO 5725 [ISO 1994]- defines precision as a part of the accuracy of measures (see Figure 5.7). In this standard, precision is defined as the closeness of agreement of test results. To evaluate the precision of measures, the ISO 5725 presents the formulae for calculating the repeatability and reproducibility of the measures.

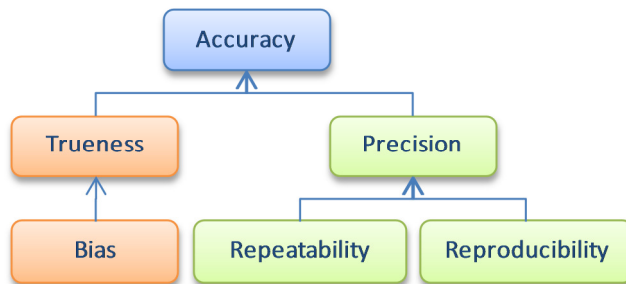


Figure 5.7 Measurement of accuracy under ISO 5725.

Furthermore, the International Vocabulary of General Terms in Metrology (VIM) [ISO 2004] defines the precision term as the closeness of agreement between quantity values obtained by replicated measurements of a quantity, under specified conditions.

In the software measurement area, the International Standard for Functional Size Measurement ISO 14143 [ISO 2003] does not specifically define the term precision. This standard only has a note that advises that the term precision should not be used as a synonym for accuracy. However, ISO 14143 presents the definition of repeatability and reproducibility of the results of measurements. This standard provides a brief example of the

calculation of repeatability, but does not show the formulae applied to obtain repeatability and the conditions of the measurement task that assures the same conditions for each measure. For the measurement of reproducibility, not even a brief example is provided.

Repeatability is defined as the closeness of the agreement between the results of successive measurements of the same measurand carried out under the same conditions, taking into account that the same conditions imply the same measurement procedure, the same observer, the same measuring instrument, the same conditions of use of the measuring instrument, the same location, and repetitions over a short period of time [ISO 2003].

Reproducibility is defined as the closeness of the agreement between the results of measurements of the same measurand carried out under changed conditions of measurement. The changed conditions may include: the measurement principle, the measurement method, the observer, the measuring instrument, the reference standard, the location, the conditions of use, and the time [ISO 2003].

In the software engineering community, the term precision is sometimes incorrectly confused with the term accuracy. For instance, Kemerer in [Kemerer 1993] uses indistinctly the terms accuracy and precision, and also uses the term reliability instead of reproducibility of precision. These terms should not be used indistinctly because they have a different meaning.

Some authors have taken into account the measurement of the reproducibility of software measures, such as [Abraham et al. 2004] [Condori-Fernández and Pastor 2006]. These authors evaluate the reproducibility of measurement results of functional size procedures based on IFPUG-FPA and COSMIC, respectively. Both approaches use a statistical equation similar to the one proposed by Kemerer [Kemerer 1993], which calculates the difference in absolute value between the measure produced by a subject and the average measure (for the same FSM method) produced by the other subjects in the measurement task, divided by the average measurement results. The main disadvantage of this formula is that it uses

the average, which is not correct if the results of the measurements are not homogeneous. Thus, the use of this formula does not allow the generalization of the measurement procedure used in [Abrahao et al. 2004] and [Condori-Fernández and Pastor 2006].

Other authors have taken into account the measurement of the repeatability of software measures, such as [Diab et al. 2005]. They affirm that repeatability is assured with the automation of the measurement procedure. Although we agree with this affirmation, it is important to control repeatability from the design phase of the measurement procedure in order to detect weakness and improve the measurement procedure before its automation.

We have not found any analyses of the precision of software measures in terms of repeatability and reproducibility. By evaluating both attributes we can detect the causes that produce variability of measurements, such as: the knowledge of the subjects that perform the measurement task; the concentration level of these subjects; their understanding of the instrumentation material; the legibility of the instrumentation material; the correctness of the explanation of the measurement procedure, etc.

The next section presents a method for evaluating the precision of software measures taking into account repeatability and reproducibility of measures.

5.3.1 A Method for the Evaluation of Precision

A method is a set of successive steps that leads to a goal. A method is described by specification of each step that makes up the method and the means used to achieve the goal. We design a method that allows measurement of the precision of software measures according to the standard ISO 5725 [ISO 1994], which is widely used in other sciences but surprisingly is not used in software engineering.

We adapted ISO 5725 by instantiating this standard with concepts used in software measurement – see Table 10.

Table 10. Instantiation of ISO 5725 with software engineering concepts.

ISO 5725	Software Engineering
Measurement yield	The measurement method must have a continuous scale and must give a single value as the result of the test. The continuous scale means that there is no limit on the number of possible values of the measures.
Operator	Subjects that will measure the software artifacts (managers, analysts, designers, etc). These subjects must have knowledge of software engineering and must also be familiar with the use of software artifacts, such as conceptual models (case use diagrams, class diagrams, activity diagrams, process diagrams, etc.).
Test site	The place where the subject will measure the software products (office, classroom, etc).
Equipment	The software artifact to be measured and the instruments to measure this artifact. The instruments to measure a software artifact can be for manual, semi-automatic, or automatic use.
Laboratories	A laboratory is the combination of the subjects with the artifacts and the instruments to measure in the place where they will do the measurement. In our field of study, a laboratory is the combination of: subjects (i.e. manager, programmer, and analyst.), artifacts (which are the software products obtained from any phase of the development process, i.e. use cases, class diagram, and source code), and instruments (i.e. the measurement procedure).
Different levels of the test	These levels can be the complexity or size levels of the conceptual models that will be measured. Both criterions of levels are representatives for the measurement of the precision. For instance, for the measurement of the size of an artifact, the levels small, medium and large are required.

The method designed for the evaluation of the precision of software measures is comprised of three phases: a definition phase, a measurement phase, and an evaluation phase. Figure 5.8 schematizes the method for the evaluation of precision using a UML activity diagram.

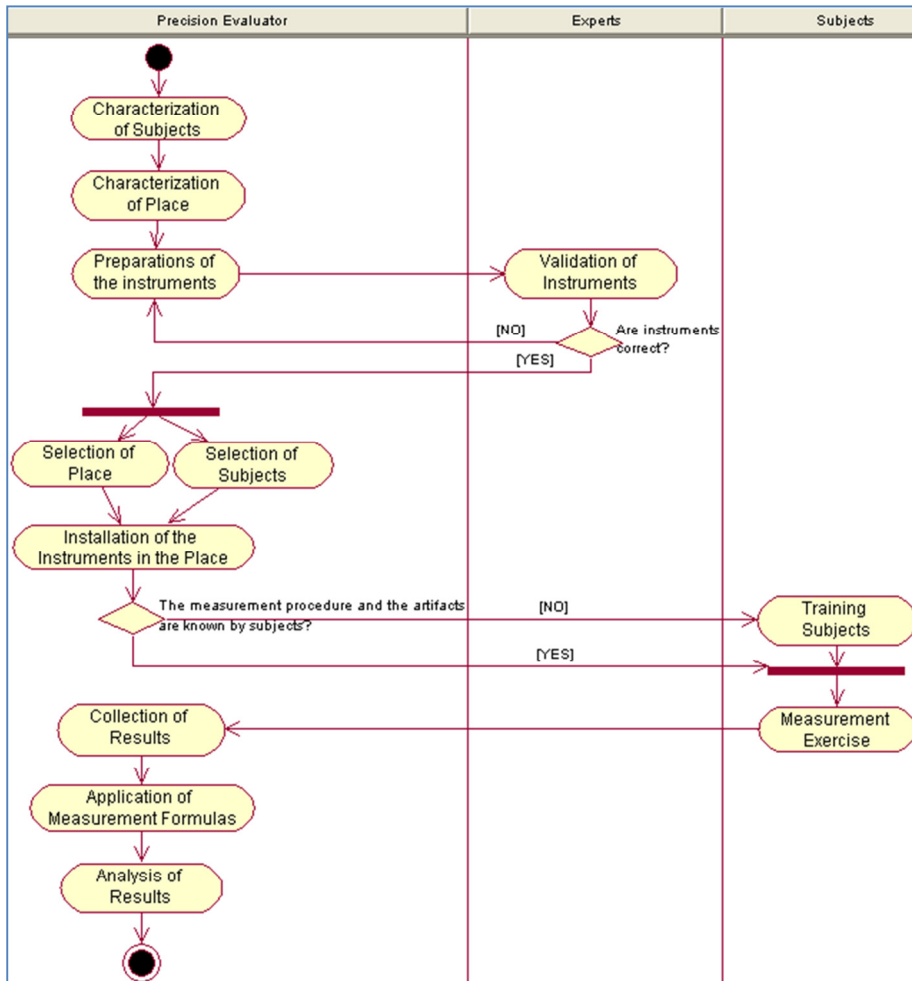


Figure 5.8 Method for evaluation of precision of software measures.

The Definition Phase

This phase is comprised of four activities: characterization of subjects, characterization of place, preparation of the instruments for the measurement exercise, and validation of the instruments.

Characterization of subjects includes the identification of the level of knowledge of the measurement procedure and the software artifacts of the subjects.

Characterization of place includes the specification of the size of the place, illumination, movables, and the computers (i.e. OS, RAM, and software installed) used for the measurement.

Preparation of instruments corresponds to the preparation of the materials that the subjects must use to perform the measurement exercise. The preparation of the instruments will depend of the knowledge of the subjects. If the selected subjects don't have knowledge of the measurement procedure or of the software artifacts to be measured, then it is necessary to prepare the instruments for training the subjects and the instruments to perform the measurement exercise. In contrast, if the selected subjects have a good knowledge of the measurement procedure and the use of the artifacts to be measured, then it is only needed the preparation of the instruments to perform the measurement exercise.

When training is required, preparation of the instruments to measure will require the specification of the techniques that will be used in the training activity. A widely used training method is the demonstration / practice method. To do this, the training material consists of the presentation of the measurement procedure, the presentation of an example of the application of the measurement procedure, a software artifact to provide a guided application of the measurement procedure, and a results sheet to register the measurement results of the software artifact.

The material for the measurement exercise consists of instructions to perform the measurement, a set of software artifacts of different levels, and a

results sheet for every software artifact to be measured. In the instructions for the experiment, every measurement of the software artifacts must be specified. Each measurement must be carried out at least twice to allow the measurement of repeatability. The instructions of the measurement exercise will be the same for all the subjects to allow the measurement of reproducibility.

In the *validation of instruments* activity, the instruments must be reviewed by a small group (for instance three people) of experts in the software artifacts and in the measurement procedure. The number of people that will review the instruments must be small in order to diminish the noise in the validation of the instruments. If experts decide that instruments are not correct, the instruments must be modified and the validation performed again.

The Measurement Phase

This phase is comprised of five activities: selection of subjects, selection of place, installation of instruments in the place, training of subjects if it is required, and the measurement exercise.

In the *selection of subjects* activity, the people that will carry out the measurement exercise must be selected according to the characterization of the subjects. The subject must be selected randomly from a set of subjects that have similar backgrounds in the software artifacts that will be measured.

It is important to note that the number of selected subjects will affect the significance of the measures. A greater number of selected subjects allows a greater significance of measures; at least 30 subjects would be required to obtain measures with a 95% certainty.

In the *selection of place* activity, the place where the measurement exercise will be carried out must be selected according to the characterization of the place. The selected place must have one computer for

each subject selected, and every computer must have the same conditions (i.e. OS, RAM, software installed).

The activity of *installation of the instruments in the place* consists in copying the prepared instruments in each computer that will be used by the subjects. Also this activity includes the printing of the instruments that will be given in paper to the subjects and the location of the instruments in the place.

The *training of subjects* activity is performed if the selected subjects don't have a high knowledge of the software artifact and the measurement procedure. In this activity, the subjects are trained using the training instruments; the subjects can ask whether they have questions that may be required for the correct understanding of the software artifact and the measurement procedure.

In the *measurement exercise* activity, the subjects perform the measurement following the instructions specified in the *preparation of the instruments*. In this step, the subjects cannot ask questions.

The Evaluation Phase

This phase is comprised of three activities: collection of results, application of measurement formulae, and analysis of the values calculated.

We select and adapt some tables and formulae presented in the standard ISO 5725 to the collection of results and application of measurement formulae activities. The standard ISO 5725 has 50 formulae that in conjunction allow the calculation of repeatability and reproducibility of the measures and the analysis of the values obtained. This standard also has six tables to input data. We select and adapt six formulae and three tables.

The activity of the *recollection results* consists in the collection of measures obtained by the subjects in each level of the software artifacts. In this activity, the measures are recorded using Table 11, which shows the subjects that perform the measurement in the first column, and the levels of

the artifacts used to perform the measurement in the first row. Each measure obtained by the subjects at each level is recorded in the intersection of subjects and levels.

Table 11. Table to collect the results of the measures (adapted from ISO 5725).

Subject	Level				
	1	...	j	...	m
1	Measure 111 Measure 112				Measure 1m1 Measure 1n2
...					
i			... Measure ijk ...		
...					
n	Measure n11 Measure n12				Measure nm1 Measure nm2

In this activity the data obtained must be also validated. Since the subjects must follow the instructions to perform the measurement exercise, redundant and missing data is avoided, because in the instructions there is no indication on performing redundant measures and all the steps detailed in the instructions must be strictly followed by the subjects.

The validation of the data obtained also includes identification and the analysis of outliers and outlying subjects. The outliers here are measures that deviate greatly from comparable entries in Table 11. The outlying subjects are subjects that have several unexplained abnormal measures. The outliers and the outlying subjects must be investigated and analyzed to ascertain the reasons for their divergence. Once the investigation and the analysis have been performed, the person that planned the measurement exercise must

decide whether the outliers and the outlying subjects will be ignored or must be corrected (and must document the reason).

When the measurement results registered in Table 11 have been validated, the cell arithmetic means must be calculated. Each cell is the set of measures recorded for each subject at each level. The cell means are calculated using the following formula:

$$Cell_{ij} = \frac{1}{n_{ij}} \sum_{k=0}^{n_{ij}} Measure_{ijk} \quad (4)$$

where $Measure_{ijk}$ corresponds to each measure obtained for a subject i at level j , and n_{ij} is the number of measurement results obtained for a subject i at level j . The cell means calculated are recorded in a table with the structure of Table 12. As Table 12 shows, the mean of the cell is recorded at the intersection of subjects and levels.

Table 12. Table to collect the arithmetic means of cells (adapted from ISO 5725).

Subject	Level				
	1	...	j	...	m
1	Cell 11				Cell 1m
...					
i			Cell ij		
...					
n	Cell n1				Cell nm

Once the mean of each cell has been calculated, the spread of each cell must be quantified by means of the application of Formula 5:

$$Spread_{ij} = \sqrt{\frac{1}{n_{ij} - 1} \sum_{k=0}^{n_{ij}} (Measure_{ijk} - Cell_{ij})^2} \quad (5)$$

where $Cell_{ij}$ is the mean of the cell measure for a subject i at level j . The spreads calculated are recorded in a table with the structure shown in Table 13; the spread is registered in the intersection of the subjects and the levels.

Table 13. Table for recording the spread of cells.

Subject	Level				
	1	...	j	...	m
1	Spread 11				Spread 1m
...					
i			Spread ij		
...					
n	Spread n1				Spread nm

In the *application of measurement formulae* activity the precision is calculated for each level using the repeatability variance and the reproducibility variance.

To calculate the repeatability variance, Formula 6 is used.

$$S^2_{rj} = \frac{\sum_{i=1}^p (n_{ij} - 1) Spread_{ij}^2}{\sum_{i=1}^p (n_{ij} - 1)} \quad (6)$$

where $Spread_{ij}$ corresponds to the spread of cell measures for a subject i at level j , and p is the number of subjects that perform the measurement exercise.

To calculate the reproducibility variance, Formula 7 is used.

$$S^2_{Rj} = S^2_{rj} + S^2_{Lj} \quad (7)$$

where S^2_{rj} corresponds to the repeatability variance, and S^2_{Lj} is the between-subjects variance.

To obtain the value of S^2_{Lj} , the application of Formula 8 is required to calculate the general mean of a level j represented by m_j , and Formula 9 that uses the values calculated previously to obtain the between-subjects variance.

$$m_j = \frac{\sum_{i=1}^p n_{ij} Cell_{ij}}{\sum_{i=1}^p n_{ij}} \quad (8)$$

$$S^2_{Lj} = \frac{\left(\left(\frac{1}{p-1} \sum_{i=1}^p n_{ij} (Cell_{ij} - m_j)^2 \right) - S^2_{rj} \right)}{\frac{1}{p-1} \left[\sum_{i=1}^p n_{ij} - \frac{\sum_{i=1}^p n_{ij}^2}{\sum_{i=1}^p n_{ij}} \right]} \quad (9)$$

Finally, in the *analysis of results* activity, the precision of the measurement results is evaluated. If the repeatability and the reproducibility variance are low, it indicates that the measurement results have high precision. These values should be low because they represent the magnitudes of the expected measurement error within and between the measurement results, respectively.

If the repeatability variance is high, it indicates that the instruments prepared for the empirical study must be reviewed and corrected or redesigned. On the other hand, if the reproducibility variance is high, it indicates the possibility that the knowledge of the selected subjects might be dissimilar, the measurement procedure has not been correctly understood, or better training of the subjects is required.

5.3.2 A Pilot Study to Evaluate the Precision of OOmCFP

The term *pilot study* (or feasibility study) is used to refer to mini versions of a full-scale study, as well as specific pre-testing of a particular research instrument. Pilot studies are a crucial element of a good study design, which might give in advance warnings about where the main project could fail. Conducting a pilot study does not guarantee success in the main study, but it makes success more likely [Teijlingen and Hundley 2001]. Given that pilot studies are very important in empirical research, in this section we present a pilot study carried out to apply the precision evaluation method to OOmCFP.

In order to define the goal of the pilot study, we used the Goal/Question/Metric (GQM) template [Basili and Rombach 1988], which describes the goal as follows: “To analyze the OOmCFP measurement procedure and the instruments used for the measurement exercise for the purpose of evaluating its correctness from the viewpoint of the researcher in the context of Computer Science students measuring OO-Method conceptual models with OOmCFP”.

This pilot study has focused on detecting any warning for both Definition and Measurement phases of the precision evaluating method, since the evaluation phase is based on selected formulae from ISO 5725 that allow quantifying the precision of software measures.

The Definition Phase

The subjects were characterized as people without any knowledge of OOmCFP and with minimum knowledge of OO-Method. The place was characterized as a room with enough computers for the subjects. The computers must have Windows as Operative System, the Olivenova Modeler tool [CARE-Technologies 2011] for the work with the OO-Method conceptual models, and Microsoft Office installed.

In this phase the instruments, consisting of training instruments and the instruments for the measurement exercise, were also prepared. The training instruments were the following: a set of slides to teach the main concepts of the OO-Method conceptual model that are used by OOmCFP; a set of slides to teach the OOmCFP procedure; an illustrative example of the application of OOmCFP to the conceptual model of an invoice application; a measurement guide; a results sheet; and the application of OOmCFP to a conceptual model of a Rent-a-Car application to verify the training process carried out.

The instruments for the measurement exercise were the following: three conceptual models of OO-Method with different levels of functional size (small, medium, and large – described next), the instructions, and the blank results sheet. The conceptual model of a Publishing application was used as the small model (five classes); the conceptual model of a Photography Agency application was used as the medium model (seventeen classes); and the conceptual model of an Expense Report application was used as a large model (twenty-three classes).

The instructions have six steps: three for the specification of the purpose of the measurement of each conceptual model, and three for the repetition of the measurements. There were six results sheets, one for each measurement task. The conceptual models were designed to use 70% of the counting rules and 100% of the measurement rules of OOmCFP.

The instruments were validated by two experts in OO-Method and two experts in OOmCFP. The measurement guide and the results sheet were not well structured, and these instruments had to be changed until the experts validated all the instruments.

The Measurement Phase

The subjects were selected from the students enrolled in the “Master’s Degree in Software Engineering, Formal Methods, and Information

Systems” at the Universidad Politécnica de Valencia from September 2006 to September 2008. The group of students was made up of 12 students with at least some knowledge of the OO-Method conceptual model and without knowledge of the OOmCFP procedure.

The place selected was Room 0S02 of the Department of Information Systems and Computation of the Universidad Politécnica de Valencia. This room has twenty identical computers with the same programs installed. Each computer has the Windows OS, the Olivenova Modeler tool, and the Microsoft Office software already installed.

The installation of the instruments in the classroom consisted in copying the training and the measurement instruments into each computer of this room. Also in this activity, the measurement guide and the instructions were printed and located close to each computer of the room.

The training of subjects activity was carried out. Thus, students could develop the expertise required to measure the functional size of the OO-Method conceptual models using OOmCFP. The training method used was the demonstration/practice method. For the demonstration part, we envisaged the following tasks: (a) presentation of the OO-Method Conceptual Model, (b) use of the Olivenova Modeler tool, (c) presentation of the OOmCFP measurement procedure, and (d) illustration of the use of OOmCFP with the conceptual model of an Invoice application. For the practical part, we considered the following tasks: (e) guided application of OOmCFP to the conceptual model of a Rent-a-Car application. During the guided application of OOmCFP the students were able to clarify their doubts.

The training of subjects activity was planned to take 2 hours: 20 minutes for task (a), 10 minutes for task (b), 20 minutes for task (c), 20 minutes for task (d), and 50 minutes for task (e). However, the demonstration part (i.e., tasks a, b, c, and d) took only 1 hour and the practice part (i.e., task e) took 3 hours.

It is important to note that 4 students were experts using the tool, 5 students had some idea of the use of the tool, and 3 students had never used the tool. The expert students obtained the measurement of the functional size of the Rent-a-Car application. The students that had notions of the use of the tool carried out the measurement of some functional processes, but they didn't obtain the functional size of the application. The students that didn't know the tool, did not achieve the measurement of any functional process, but they correctly identified the functional processes.

The measurement exercise was not carried out because we identify that the students had different levels of knowledge of the Olivanova tool in the training activity. The different levels of knowledge of the tool affected the measurement of the precision; for instance, the experts correctly applied the mapping and measurement rules defined in OOmCFP, but the inexperienced students confused the elements of the conceptual model when they try to identify the data movements. The knowledge of the tool was not taken into account in the characterization of the subjects and the pilot study indicated that this is an important factor that affects the measurement of the precision.

Understanding the measurement procedure and the instruments is a crucial factor in the evaluation of the precision of measures. While the experts had validated the instruments in their expertise area, the pilot study reflected that the measurement procedure and the instruments were not correctly understood by the students. This implies that the procedure and the instruments must be changed; thus the objective of the pilot study carried out was achieved.

5.3.3 Lessons Learned from the Pilot Study

The pilot study was carried out in the design phase of OOmCFP. From the results of the study, we have learned six lessons related to the OOmCFP procedure, the measurement guide, the results sheet, and the training models. These lessons are the following:

With respect to the OOmCFP procedure:

- When the subjects carried out the measurement, many questions arose about how to properly identify each *functional process*. As follow up, we changed the design of the OOmCFP measurement procedure, detailing the rules to identify the functional processes and the elements that are contained in a functional process.
- When an *inheritance of classes* participates in a functional process, some subjects considered one data group for the entire inheritance of classes and other subjects considered one data group for each class of the inheritance. As result, we included in OOmCFP a rule that indicates that when an inherited class participates in a functional process it must be counted as a single data group.
- When the subjects carried out the measurement of the practice model, some rules defined in OOmCFP were never used. For instance, some rules that only explain functional processes, but they do not allow the identification of functional processes. We, therefore, eliminated the rules that were never used because these tended to confuse the subjects performing the measurement.

With respect to the measurement guide:

- When the subjects identified the data movements, they had difficulties in finding the data movements that occur when a conceptual element participates in a functional process because the rules were organized according to the layers of the OO-Method applications. Thus, the subjects took longer than expected to carry out the practical part of the training exercise. In response, we changed the measurement guide and reorganized the rules for the identification of data movements in accordance with the conceptual elements involved in the functional processes.

With respect to the results sheet:

- When the subjects entered the functional processes and the elements contained in each functional process, we noted that the subjects had difficulties when the elements had different levels of abstraction in the OO-Method conceptual model. We changed the results sheet in order to differentiate the elements of each level of abstraction that comprise the functional process.

With respect to the training conceptual model:

- The measurement of the conceptual model used in the practice part of the training phase was intended to take 50 minutes. However, some subjects took two hours to do the measurement. Therefore, we simplified the model used for the practical part.

With all these lessons learned, this pilot study allowed the improvement of the design of the OOmCFP measurement procedure with less cost than if it will be carried out when the procedure will be already automated. Thus, improvements performed to the design of the OOmCFP were the following:

- 1) streamlining the OOmCFP measurement procedure by eliminating two rules relating to identification of functional process and adding two rules for the identification of data groups;
- 2) reorganizing the measurement guide by structuring the rules to identify data movements according to conceptual elements;
- 3) redesigning the results sheet by assigning a specific column to record the conceptual elements of each level of abstraction;
- 4) simplifying the conceptual model used for the practical part of the training by deleting those functionalities that do not affect the measurement exercise.

Finally, we can state that the OOmCFP measurement procedure and the instruments designed to apply the OOmCFP procedure can obtain precise measurement results.

5.4 Conclusions

In this chapter, the evaluation of the design of OOmCFP has been presented. Since there is no consensus in the terminology used in the measurement field and, consequently, there is no a wide used validation frameworks for software measures, we decide to validate the design of the OOmCFP FSM procedure using standards.

The conformity evaluation of the OOmCFP FSM procedure regarding to the COSMIC FSM method has been performed by experts using the ISO 14143-2 [ISO 2002]. This evaluation indicates that OOmCFP is conformant to the COSMIC FSM method.

A metrological analysis of the design of OOmCFP has been performed using VIM [ISO 2004]. This analysis shows that the concepts that were used in the design of OOmCFP are aligned with the metrology terms and it also presents the corresponding justifications for the concepts that were not used in the design of OOmCFP.

In order to evaluate the accuracy of the OOmCFP measurement procedure, precise measures must first be obtained. We designed a method based on the ISO 5725 standard [ISO 1994] for the evaluation of the precision of measures based in the reproducibility and the repeatability of measurement results. We conducted a pilot study to evaluate the precision of OOmCFP, and results were used to improve the design of the OOmCFP procedure.

In terms of theoretical validation, since the validation of COSMIC has been carried out successfully from the perspective of measurement theory in [Condori-Fernández 2007] using the DISTANCE framework [Poels and Dedene 2000], and since the OOmCFP measurement procedure has been designed on the basis of COSMIC, we can infer that the OOmCFP measurement procedure has also been theoretically validated.

Chapter 6

Application of OOmCFP

The OOmCFP measurement procedure has been designed to obtain accurate measurement results of applications that have been generated from their conceptual models in MDD environments. This chapter presents the application of the OOmCFP measurement procedure to a rent-a-car system and the evaluation of the results obtained by this FSM procedure. Depending on how OOmCFP is applied (manual or automatic), interesting findings are observed. Thus, in this chapter we first present the activities that must be performed to complete the manual application of a measurement procedure following the software measurement process model proposed by Jacquet and Abran [Jacquet and Abran 1997]: *software documentation gathering*, *construction of the model*, and *application of numerical assignment rules* (see Figure 6.1).

The *software documentation gathering* activity consists in collecting the documentation of the software that will be measured, which is required to apply the OOmCFP measurement procedure. In the *construction of the model* activity, the model of the software to be measured is built when this model is not available. If the appropriate model is already available, this activity can be bypassed. In this activity, the construction of the COSMIC model is performed by the application of OOmCFP rules. The *application of*

numerical assignment rules is the last activity in the application of the OOmCFP FSM procedure. In this activity, the OOmCFP rules to measure the functional size are applied to the software model in order to obtain the measurement result.

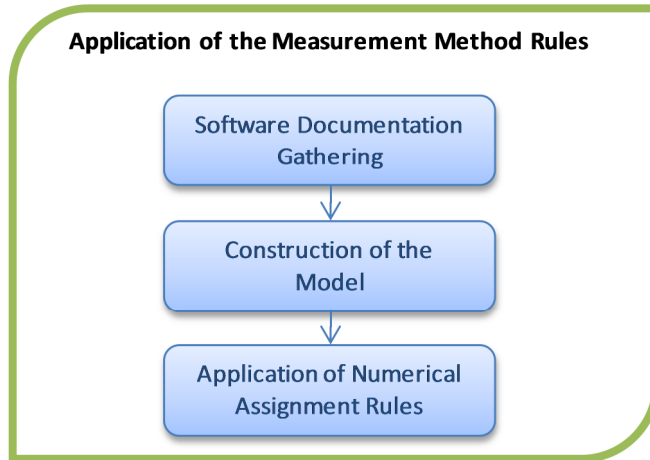


Figure 6.1 Application of OOmCFP Measurement Procedure.

After that, in this chapter we present the automated application of the OOmCFP measurement procedure. Thus, this chapter also presents a tool that automates the application of the OOmCFP measurement procedure in order to obtain accurate functional size measurement results.

6.1 Manual Application of OOmCFP

To illustrate the application of the OOmCFP procedure, we have manually measured the functional size of a rent-a-car system. The following sections show the activities carried out to apply the OOmCFP functional size measurement procedure.

6.1.1 Software Documentation Gathering

The documents that should be gathered to apply OOmCFP to an OO-Method application correspond to the OO-Method conceptual model, which is comprised of the object model, the dynamic model, the functional model, and the presentation model. This conceptual model has all the details needed for the generation of the fully working OO-Method application.

The mission statement for the rent-a-car system is: “*To allow the management of vehicle rentals*”, taking into account the following:

- A client can have many vehicle rentals.
- For each client, it is important to know his/her DNI, name, address, and phone. In addition, each client must have a unique code assigned by the system.
- A rental is performed by only one client and it is related to only one vehicle.
- For each vehicle rental, it is important to maintain the delivery and the return dates of the rental, the price of the rental, and the amount of liters of gasoline that are in the vehicle when it is rented.
- The price of a rental is calculated using a value that is fixed by the company, plus a rate that depends on the size of the vehicle.
- Each vehicle has a group, which corresponds to the size of each vehicle.
- For each vehicle it is important to maintain the registration number, the colour, the brand, and the corresponding model.
- The vehicles can be cars or minibuses. For each car it is important to maintain the number of doors, and if it consumes diesel or gasoline. All minibuses consume diesel, and for each minibus it is important to maintain the number of passengers that it can transports.

- Each vehicle rental is performed in a specific office of the company.

Figure 6.2 shows the OO-Method object model for the rent-a-car system that was developed following the mission statement of this system. In this figure it is possible to observe the following classes: Client, Rental, Office, VehicleGroup, and Vehicle (which can be specialized in Minibus or Car). In this model, it has also been created the Administrator class, which is an agent that can execute the services specified in the object model (dashed lines shows the services that the Administrator class can execute).

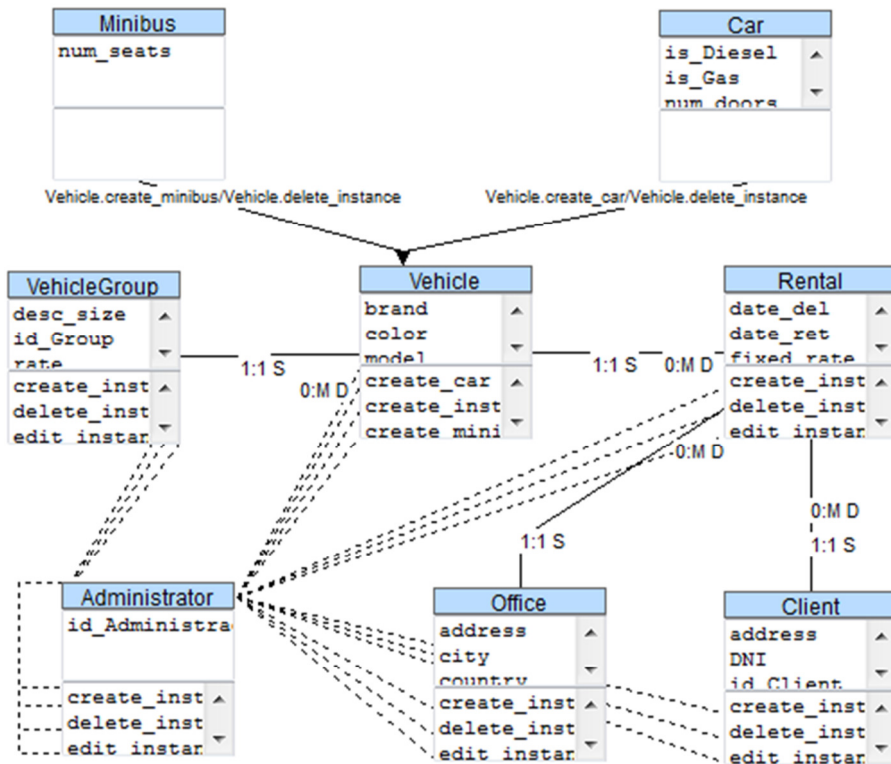


Figure 6.2 Object model of a rent-a-car system.

The functional model of the OO-Method conceptual model for the rent-a-car system captures the semantics associated to the changes of state due to the events execution. For instance, the event *edit_instance* of the class *Vehicle* assigns the value *p_color* to the attribute *color*.


Attribute	Event		Effect	Condition	Current value
 color	edit_instance	=	p_color		

Figure 6.3 Example functional model of the *Vehicle* class.

The dynamic model of the OO-Method conceptual model of the rent-a-car system represents the valid lives of the objects of the classes shown in Figure 6.2. This model is created automatically from the object model, specifying the services that change the state of the objects and the possible states that these objects could have.

For instance, the objects of the class *Client* are created using the service *create_instance*. The execution of this service changes the initial state of the objects to the state *Client0*. Figure 6.4 shows the dynamic model of the *Client* class of the rent-a-car system.

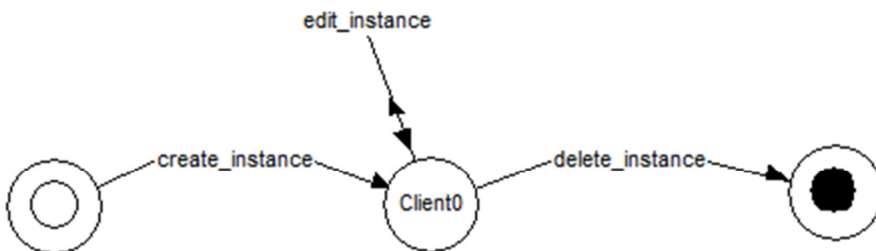


Figure 6.4 Dynamic model of the *Client* class of the rent-a-car system.

The presentation model of the rent-a-car system defines a set of patterns that allow the specification of user interfaces in an abstract way. The information of the rent-a-car system is presented to the user of the application by a menu of options that are modelled by means of a *hierarchy*

action tree (HAT) in the presentation model. For the rent-a-car system, the options of the menu are three groups of registers called Population Interaction Units (PIU), which group instances of the classes of the object model. Figure 6.5 shows the HAT of the rent-a-car system.

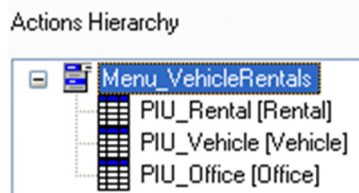


Figure 6.5 Menu (HAT) specified for the rent-a-car system.

For each PIU defined in the menu of the rent-a-car system, the attributes, the actions, the navigations, and the filters contained in the interaction units are specified. For instance, Figure 6.6 shows the attributes that will be shown in the PIU_Rental interaction unit.

Path	Alias
◆ id_Rental	id_Alquiler
◆ date_del	fecha_inicio
◆ date_ret	fecha_fin
◆ liters	litros_gas
◆ price	precio
◆ Cliente.DNI	DNI
◆ Cliente.nameC	Nombre
◆ Vehiculo.registration_number	matricula

Figure 6.6. Attributes for the PIU_RentsDetails of the rent-a-car system.

When all the models of the OO-Method conceptual model are specified, it is possible to verify the conceptual model and generate the final application from that model. Therefore, the conceptual model of the rent-a-

car system corresponds to the documentation needed to apply the OOmCFP functional size measurement procedure.

6.1.2 Construction of the COSMIC Software Model

The construction of the COSMIC software model is performed by using the OOmCFP measurement procedure. Thus, all the elements that will be measured are identified in this activity.

The Strategy Phase

The *purpose* of applying OOmCFP is to measure the functional size of the rent-a-car application that is specifically generated by the OlivaNova tool. The *scope* of this purpose is the rent-a-car OO-Method conceptual model that specifies the rent-a-car application in an abstract way. The *granularity level* is low because all the details are available and needed for the generation of the rent-a-car application.

The *pieces of software* of the rent-a-car application are the client component, the server component, and the database component. Each component of an OO-Method application is built for a particular software environment (for instance, the client component will be built with C#, ASP or JSP). Since the rent-a-car system uses ASP for the client component, EJB for the server component, and SQL for the database component, every component of the application is also one layer of the software application.

The *functional users* of the rent-a-car application have been identified using Rule 1 to Rule 6 of the OOmCFP measurement procedure. Applying Rule 1, the Administrator class of Figure 6.2 is identified as a functional user. The instances of this class can execute all the services of the rent-a-car application. Between every functional user of the rent-a-car application and its components there exists a boundary. Thus, applying Rule 2, one boundary between the administrator and the client piece of software is identified.

In addition, using Rule 3 of OOmCFP, the client component of the rent-a-car system is identified as a functional user. It is a user of the server component of the application, and using Rule 4 of OOmCFP, one boundary between the client and the server component is identified.

Moreover, using Rule 5 of OOmCFP, the server component of the rent-a-car system is identified as a functional user. It is a user of the client component and the database component of the rent-a-car system. By the application of Rule 6 of OOmCFP, one boundary between the server and the database component is identified.

Figure 6.7 presents the functional users, pieces of software, layers, and boundaries of the rent-a-car system, which has been obtained by the instantiation of Figure 4.4.

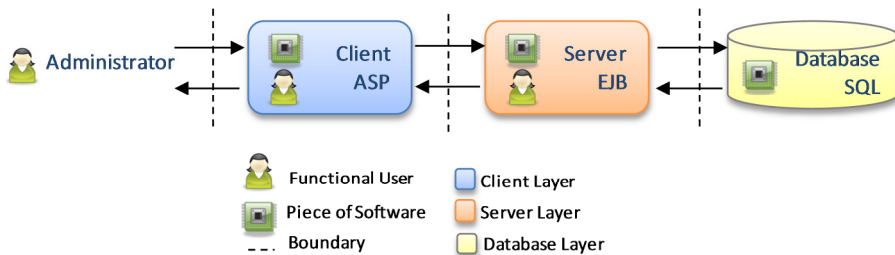


Figure 6.7. Functional users, Pieces of software, layers, and boundaries of the rent-a-car system.

The Mapping Phase

In the mapping phase of OOmCFP, the identification of the functional processes, data groups, data attributes, and data movements must be performed.

Identification of Functional Processes

The Administrator carries out triggering events that occur in the real world, for instance the registration of the rents. To register a rent in the system, the Administrator starts a set of functional processes, which are identified by using Rule 9 and Rule 10 of the OOmCFP measurement procedure.

Rule 9 specifies that a direct child of the hierarchy action tree (HAT) of the presentation model of the OO-Method conceptual model corresponds to a functional process in the client layer. These children can be Population Interaction Unit (PIU), Service Interaction Unit (SIU) or Master Detail Interaction Unit (MDIU). Therefore, focusing on Figure 6.5 that shows the HAT of the rent-a-car system, the functional processes that occur in the client layer are: PIU_Rental, PIU_Vehicle, and PIU_Office.

Rule 10 specifies that the set of formulae that solve the Server layer in response to the events that occur in the functional processes of the Client layer corresponds to a functional process. These formulae can be derivations, default values, filters, valuations, integrity constraints, triggers, transactions, preconditions, dependency rules, control conditions, and conditional navigation. Thus, the actions that the server layer carries out in response to the petitions of the client functional user correspond to functional processes. We use Rule 10.1 to name these functional processes. Thus, the functional processes that occur in the server layer of the rent-a-car system are: PIU_Rental, PIU_Vehicle, and PIU_Office.

Afterwards, the elements contained in the functional processes must be identified. To do this, Rule 9.1 is applied to identify the elements contained in each functional process that is a PIU. Applying Rule 9.1 for the functional process PIU_Rental, the display pattern DS_Rental and the action pattern A_Rental have been identified. The interaction units contained in the A-Rental action pattern have also been identified using Rule 9.1. These interaction units are the following Service Interaction Units (SIUs): SIU_create_instance, SIU_edit_instance, and SIU_delete_instance.

Next, Rule 9.2 is applied in order to identify the elements contained in each SIU. This rule allows the identification of the arguments that have related a PIU, and the conditional navigations with their related interaction units. Since SIU_create_instance, SIU_edit_instance, and SIU_delete_instance do not have arguments with a PIU related or conditional navigations to other interaction units defined, the identification of elements contained in the SIUs ends. Consequently, the identification of elements contained in the functional process PIU_Rental ends. Table 14 presents the elements contained in this functional process.

Table 14. Contained elements in PIU_Rental.

Functional Process	Contained Elements	
PIU_Rental	DS_Rental	
	A_Rental	SIU_create_instance
		SIU_edit_instance
		SIU_delete_instance

Subsequently, the functional process PIU_Vehicle is analyzed in order to identify the elements that it has contained. Applying Rule 9.1, the display pattern DS_Vehicle, the action pattern A_Vehicle, and the navigation pattern N_Vehicle have been identified. Next, applying Rule 9.1 to the action pattern A_Vehicle, three interaction units have been identified: SIU_create_instance, SIU_delete_instance, and SIU_edit_instance. Since there is no interaction units related to the arguments of the SIUs or conditional navigations defined in these SIUs, the identification of elements contained in the SIUs and the A_Vehicle action pattern ends.

By applying Rule 9.1 to the N_Vehicle navigation pattern, two interaction units have been identified: PIU_Car and PIU_Minibus. After that, Rule 9.1 is applied to identify the elements contained in PIU_Car. Thus, the display pattern DS_Car and the action pattern A_Car have been identified. The interaction unit SIU_create_car contained in the A_Car action pattern

has also been identified using Rule 9.1. Then, applying Rule 9.2 to SIU_create_car, other related interaction units have no been found. Thus, the identification of elements contained in PIU_Car ends.

By the application of Rule 9.1 to PIU_Minibus, the display pattern DS_Minibus and the action pattern A_Minibus have been identified. The interaction unit SIU_create_minibus contained in the A_Minibus action pattern has also been identified using Rule 9.1. Then, applying Rule 9.2 to SIU_create_minibus, other related interaction units have not been found. Thus, the identification of elements contained in PIU_Minibus ends, and consequently, the identification of elements contained in PIU_Vehicle ends. Table 15 presents the elements contained in this functional process.

Table 15. Contained elements in PIU_Vehicle.

Functional Process	Contained Elements			
PIU_Vehicle	DS_Vehicle			
	A_Vehicle	SIU_create_instance		
		SIU_edit_instance		
		SIU_delete_instance		
	N_Vehicle	PIU_Car	DS_Car	
			A_Car	SIU_create_car
		PIU_Minibus	DS_Minibus	
A_Minibus			SIU_create_minibus	

Afterwards, the functional process PIU_Office is analyzed. Applying Rule 9.1, the DS_Office display pattern and the A_Office action pattern have been identified. Then, by the application of Rule 9.1 to the action pattern, SIU_create_instance, SIU_edit_instance, and SIU_delete_instance have been identified. Since there is no other interaction units contained in these SIUs, the identification of elements contained in the PIU_Office functional process ends. Table 16 presents the elements contained in the PIU_Office functional process.

Table 16. Contained elements in PIU_Office.

Functional Process	Contained Elements	
PIU_Office	DS_Office	
	A_Office	SIU_create_instance
		SIU_edit_instance
		SIU_delete_instance

In order to avoid counting functional processes duplicated and to avoid counting interaction units that are auto-contained, OOmCFP has defined Rule 11 and Rule 12. Since neither duplicated functional processes nor interaction units auto-contained were identified in the rent-a-car system, these rules have not been used.

Identification of Data Groups

Using Rule 13 of the OOmCFP measurement procedure, we identify some data groups of the rent-a-car system. This rule specifies that the data groups of each functional process are the classes that participate in this functional process and that do not participate in an inheritance hierarchy. Thus, using this rule, *Rental*, *Client*, *Office*, *VehicleGroup*, and *Administrator* were identified as data groups.

Then, using Rule 14 of OOmCFP, the *Vehicle* class was identified as a data group, since this rule specifies that the parent class of an inheritance hierarchy that participate in a functional process corresponds to a data group. After that, using Rule 15 of the OOmCFP measurement procedure, classes *Car* and *MiniBus* were identified as data groups due to they are child classes that have different attributes than their parent class.

Identification of Data Attributes

By applying Rule 16 of OOmCFP, the data attributes of the data groups were identified. For instance, *desc_size*, *id_Group*, and *rate* were identified using this rule as attributes of the data group VehicleGroup (see Figure 6.2).

Table 17 shows the functional processes, the data groups that participate in each functional process, and the attributes of every data group.

Table 17. Functional Processes, data groups, and data attributes of the rent-a-car application.

Functional process	Data groups	Data attributes
PIU_Rental	Rental	Id_Rental, date_del, date_ret, liters, fixed_rate, price
	Client	id_Client, DNI, nameC, address, phone
	Vehicle	Registration_number, color, model, brand
	VehicleGroup	id_Group, desc_size, rate
	Office	id_Office, pone, address, city, country
	Administrator	id_Administrator
PIU_Vehicle	Vehicle	Registration_number, color, model, brand
	VehicleGroup	id_Group, desc_size, rate
	Car	num_doors, is_Diesel, is_Gas
	Minibus	num_seats
	Administrator	id_Administrator
PIU_Office	Office	id_Office, pone, address, city, country
	Administrator	id_Administrator

Identification of Data Movements

Figure 6.8 presents the data movements that occur between the functional users and the functional processes of the rent-a-car system, which has been obtained by the instantiation of Figure 4.5.

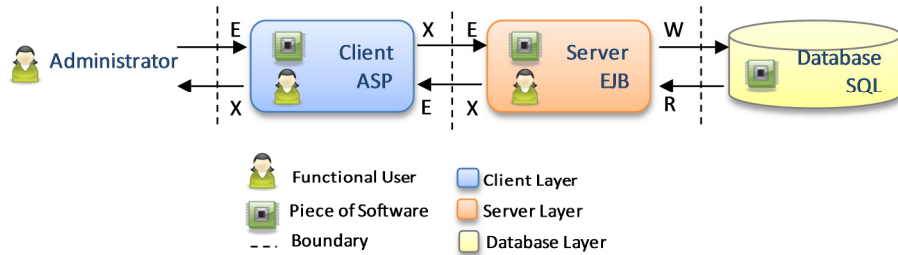


Figure 6.8. Data movements that occur in the rent-a-car system.

To identify the data movements that occur in the rent-a-car system, Counting Rules 1 to 74 of OOmCFP have been applied. For each functional process these rules are applied to the elements that it contains.

Regarding to the PIU_Rental functional process, applying the Counting Rule 1 to the DS_Rental display pattern, three read (R) data movements of the *Rental*, *Client*, and *Vehicle* data groups have been identified to the server layer. Then, applying the Counting Rule 3, three exit (X) data movements of the *Rental*, *Client*, and *Vehicle* data groups have been also identified to the server layer. By the application of the Counting Rule 4, three entry (E) data movements of the *Rental*, *Client*, and *Vehicle* data groups have been identified to the client layer. Next, applying the Counting Rule 5, one exit (X) data movement has been identified to the client layer.

Subsequently, the SIUs contained in the A_Rental action pattern are analyzed. Applying the Counting Rule 31 to SIU_create_instance, one entry (E) data movement of the *Rental* data group has been identified to the client layer; and, applying the Counting Rule 32, three entry (E) data movements of *Office*, *Client*, and *Vehicle* data groups have been identified to the client layer. Then, by the application of the Counting Rule 33, one exit (X) data moment of the *Rental* data group has been identified to the client layer. Also, by the application of the Counting Rule 34, three exit (X) data movements of the *Office*, *Client*, and *Vehicle* data groups have been identified to the client layer. After that, applying the Counting Rule 35, one entry (E) data

movement for the *Rental* data group has been identified to the server layer. Also, applying the counting Rule 36, three entry (E) data movements of the *Office*, *Client*, and *Vehicle* data groups have been identified to the server layer. Since the *SIU_create_instance* has related a creation event, one write (W) data movement for the *Rental* data group has been identified to the server layer by the application of the Counting Rule 44.

Applying the Counting Rule 31 to *SIU_edit_instance*, one entry (E) data movement of the *Rental* data group has been identified to the client layer. Then, by the application of the Counting Rule 33, one exit (X) data moment of the *Rental* data group has been identified to the client layer. After that, applying the Counting Rule 35, one entry (E) data movement for the *Rental* data group has been identified to the server layer. Since the *SIU_edit_instance* has related an event with valuations specified, using the Counting Rule 45, one write (W) data movement for the *Rental* data group has been identified to the server layer.

Later, by the application of the Counting Rule 32 to *SIU_delete_instance*, one entry (E) data movement of the *Rental* data group has been identified to the client layer. Then, by the application of the Counting Rule 34, one exit (X) data moment of the *Rental* data group has been identified to the client layer. Applying the Counting Rule 36, one entry (E) data movement for the *Rental* data group has been identified to the server layer. Due to *SIU_delete_instance* has related a destroy event, applying the Counting Rule 43, one write (W) data movement for the *Rental* data group has been identified to the server layer. Table 18 summarizes the data movements that occur in the *PIU_Rental* functional process.

Regarding to the *PIU_Vehicle* functional process, the OOmCFP rules have been applied to the elements that has contained. By the application of the Counting Rule 1 to *DS_Vehicle*, two read (R) data movements of the *Vehicle* and *VehicleGroup* data groups have been identified to the server layer, and applying the Counting Rule 3, two exit (X) data movements have been identified for the same data groups to the server layer. Then, applying

the Counting Rule 4, two entry (E) data movement of the *Vehicle* and *VehicleGroup* data groups have been identified for the client layer; and by the application of the Counting Rule 5, one exit (X) data movement has been identified to the client layer.

Table 18. Data movements that occur in PIU_Rental.

Functional Process	Contained Elements		Client		Server			
			E	X	E	X	R	W
PIU_Rental	DS_Rental		3	1	0	3	3	0
	A_Rental	SIU_create_instance	4	4	4	0	0	1
		SIU_edit_instance	1	1	1	0	0	1
		SIU_delete_instance	1	1	1	0	0	1

Then, the data movements that occur in the interaction units contained in the A_Vehicle action pattern have been identified. Regarding to SIU_create_instance, applying the Counting Rule 31, one entry (E) data movement of the *Vehicle* data group has been identified to the client layer. And, applying the Counting Rule 32, one entry (E) data movement of the *VehicleGroup* data group has been identified to the client layer. Then, by the application of the Counting Rule 33, one exit (X) data movement of the *Vehicle* data group has been identified to the client layer. Also, by the application of the Counting Rule 34, one exit (X) data movement of the *VehicleGroup* data group has been identified to the client layer. Applying the Counting Rule 35, one entry (E) data movement of the *Vehicle* data group has been identified to the server layer; and, applying the Counting Rule 36, one entry (E) data movement of the *VehicleGroup* data group has been identified to the server layer. Due to SIU_create_instance has a creation event related, then, applying the Counting Rule 44, one write (W) data movement of the *Vehicle* data group has been identified to the server layer.

Applying the Counting Rule 31 to SIU_edit_instance, one entry (E) data movement of the *Vehicle* data group has been identified to the client layer.

Then, by the application of the Counting Rule 33, one exit (X) data moment of the *Vehicle* data group has been identified to the client layer. After that, applying the Counting Rule 35, one entry (E) data movement for the *Vehicle* data group has been identified to the server layer. Since the *SIU_edit_instance* has related an event with valuations specified, using the Counting Rule 45, one write (W) data movement for the *Vehicle* data group has been identified to the server layer.

After that, *SIU_delete_instance* has been analyzed. Applying the Counting Rule 32 to *SIU_delete_instance*, one entry (E) data movement of the *Vehicle* data group has been identified to the client layer. Then, by the application of the Counting Rule 34, one exit (X) data moment of the *Vehicle* data group has been identified to the client layer. By the application of the Counting Rule 36, one entry (E) data movement for the *Vehicle* data group has been identified to the server layer. Due to *SIU_delete_instance* has related a destroy event, applying the Counting Rule 43, one write (W) data movement for the *Vehicle* data group has been identified to the server layer.

Regarding to the *PIU_Car* interaction unit that is related to the *N_Vehicle* navigation pattern, applying the Counting Rule 1 to *DS_Car*, two read (R) data movements of the *Vehicle* and *Car* data groups have been identified to the server layer. Applying the Counting Rule 3, two exit (X) data movements to the *Vehicle* and *Car* data groups have been identified to the server layer. Then, by the application of the Counting Rule 4, two entry (E) data movements of the *Vehicle* and *Car* data groups have been identified to the client layer. At last, applying the Counting Rule 5, one exit (X) data movement has been identified to the client layer.

Next, applying the Counting Rule 31 to *SIU_create_car* that is related to the *A_Car* action pattern, one entry (E) data movement of the *Car* data group has been identified to the client layer; and, applying the Counting Rule 32, one entry (E) data movement of the *Vehicle* data group has been identified to the client layer. Then, applying the Counting Rule 33, one exit (X) data movement of the *Car* data group has been identified to the client layer; and,

applying the Counting Rule 34, one exit (X) data movement of the *Vehicle* data group has been identified to the client layer. After that, by the application of the Counting Rule 35, one entry (E) data movement of the *Car* data group has been identified to the server layer; and, by the application of the Counting Rule 36, one entry (E) data movement of the *Vehicle* data group has been identified to the server layer. Since SIU_create_car has related a carrier event, one write (W) data movement of the *Car* data group has been identified to the sever layer.

Then, the data movements that occur in the PIU_Minibus interaction unit are analyzed applying the OOmCFP rules. By the application of the Counting Rule 1 to DS_Minibus, two read (R) data movements of the *Vehicle* and *Minibus* data groups have been identified to the server layer. By the application of the Counting Rule 3, two exit (X) data movements of the *Vehicle* and *Minibus* data groups have been identified to the sever layer. Later, applying the Counting Rule 4, two entry (E) data movements of the *Vehicle* and *Minibus* data groups have been identified to the client layer. Then, applying the Counting Rule 5, one exit (X) data movement to the client layer has been identified.

After that, SIU_create_minibus that is related to the A_Minibus action pattern has been analyzed. By the application of the Counting Rule 31, one entry (E) data movement of the *Minibus* data group has been identified to the client layer. Also, by the application of the Counting Rule 32, one entry (E) data movement of the *Vehicle* data group has been identified to the client layer. Then, applying the Counting Rule 33, one exit (X) data movement of the *Minibus* data group has been identified to the client layer; and, applying the Counting Rule 34, one exit (X) data movement of the *Vehicle* data group has been identified to the client layer. Later, applying the Counting Rule 35, one entry (E) data movement of the *Minibus* data group has been identified to the server layer; and, applying the Counting Rule 36, one entry data movement of the *Vehicle* data group has been identified to the server layer. Due to the SIU_create_minibus interaction unit has related a carrier event,

applying the Counting Rule 44, one write (W) data movement of the *Minibus* data group has been identified to the server layer. Therefore, the identification of data movements in the PIU_Vehicle functional process ends. Table 19 summarizes the data movements that occur in the PIU_Vehicle functional process.

Table 19. Data movements that occur in PIU_Vehicle.

Functional Process	Contained Elements				Client		Server				
					E	X	E	X	R	W	
PIU_Vehicle	DS_Vehicle				2	1	0	2	2	0	
	A_Vehicle	SIU_create_instance			2	2	2	0	0	1	
		SIU_edit_instance			1	1	1	0	0	1	
		SIU_delete_instance			1	1	1	0	0	1	
	N_Vehicle	PIU_Car	DS_Car			2	1	0	2	2	0
			A_Car	SIU_create_car		2	2	2	0	0	1
		PIU_Mini bus	DS_Mini bus			2	1	0	2	2	0
			A_Mini bus	SIU_create_minibus		2	2	2	0	0	1

After that, the OOmCFP rules are applied to the PIU_Office functional process. Applying the Counting Rule 1 to DS_Office, one read (R) data movement of the *Office* data group has been identified to the server layer. Then, applying the Counting Rule 3, one exit (X) data movement of the *Office* data group has been identified to the server layer. By the application of the Counting Rule 4, one entry (E) data movement of the *Office* data

group has been identified to the client layer; and, applying the Counting Rule 5, one exit (X) data movement has been identified to the client layer.

Then, the SIUs contained in the A_Office action pattern are analyzed. By the application of the Counting Rule 31 to SIU_create_instance, one entry (E) data movement of the *Office* data group has been identified to the client layer. By the application of the Counting Rule 33, one exit (X) data movement of the *Office* data group has been identified to the client layer. Next, applying the Counting Rule 35, one entry (E) data movement of the *Office* data group has been identified to the server layer; and, since SIU_create_instance has related a creation event, one write (W) data movement of the *Office* data group has been identified using the Counting Rule 44.

By the application of the Counting Rule 31 to the SIU_edit_instance interaction unit, one entry (E) data movement of the *Office* data group has been identified to the client layer. Then, by the application of the Counting Rule 33, one exit (X) data movement of the *Office* data group has been identified to the client layer. After that, using the Counting Rule 35, one entry (E) data movement of the *Office* data group has been identified to the server layer. Due to SIU_edit_instance is an event that has valuations specified, one write (W) data movement of the *Office* data group has been identified using the Counting Rule 45.

Subsequently, applying the Counting Rule 32 to SIU_delete_instance, one entry (E) data movement of the *Office* data group has been identified to the client layer; and, applying the Counting Rule 34, one exit (X) data movement of the *Office* data group has been identified to the client layer. Next, by the application of the Counting Rule 36, one entry (E) data movement of the *Office* data group has been identified to the server layer. At the end, one write (W) data movement of the *Office* data group has been identified to the server layer because SIU_delete_instance has related a destroy event. Table 20 summarizes the data movements that occur in the PIU_Office functional process.

Table 20. Data movements that occur in PIU_Office.

Functional Process	Contained Elements		Client		Server			
			E	X	E	X	R	W
PIU_Office	DS_Office		1	1	0	1	1	0
	A_Office	SIU_create_instance	1	1	1	0	0	1
		SIU_edit_instance	1	1	1	0	0	1
		SIU_delete_instance	1	1	1	0	0	1

Once the identification of data movements that occur between the functional users and the functional process ends, the construction of the software model ends.

6.1.3 The Measurement Phase: Assignment of Numerical Rules

In the assignment of numerical rules, 1 CFP (Cosmic Function Point) is assigned to each data movement identified above. Therefore, the aggregation of the identified data movements will determine the functional size.

Applying the Measurement Rule 1, the data movements identified in the client layer are aggregated by each functional process. Thus, the functional size of the PIU_Rental functional process is 16 CFP, PIU_Vehicle functional process is 25 CFP, and PIU_Office functional process is 8 CFP in the client layer.

Then, applying the Measurement Rule 2, the data movements identified in the server layer are also aggregated by each functional process. Thus, the functional size of the PIU_Rental functional process is 15 CFP, PIU_Vehicle functional process is 25 CFP, and PIU_Office functional process is 8 CFP in the server layer.

Later, by the application of the Measurement Rule 3, 49 CFP has been obtained for the client layer; and applying the Measurement Rule 4, 48 CFP

has been obtained for the server layer. Finally, applying the Measurement Rule 5 to the rent-a-car system, 97 CFP has been obtained for the whole rent-a-car application.

6.2 Automatic Application of OOmCFP

For industrial MDD developments, it is essential to perform the measurements quickly and in a precise way. Thus, a tool that allows the automatic measurement of conceptual models used in MDD environments is needed to avoid the excessive time and the precision errors involved in a manual measurement process.

The tool that has been developed to automatically apply the rules defined in OOmCFP to conceptual models is presented in this section. This tool has been developed using Visual Studio .Net 2003 with the language C#. The OOmCFP tool has a flexible multi-tier architecture that allows the easy adaptation to the evolution of models, facilitating the incorporation of new rules or changing existing ones. Also, the OOmCFP tool provides mechanisms to speed up the measurement process.

6.2.1 Architecture of OOmCFP Tool

Since the analysis of the models has been separated in different layers of the tool (see Figure 6.9), every layer analyses a small part of the information of the model. Consequently, each layer of the tool is made up of no more than 100 lines of code, and has comments and regions for each conceptual construct that is analyzed in this layer. Therefore, for programmers is easy to find the rules related to a specific conceptual construct, and hence, it is easier to aggregate new rules or change existing ones.

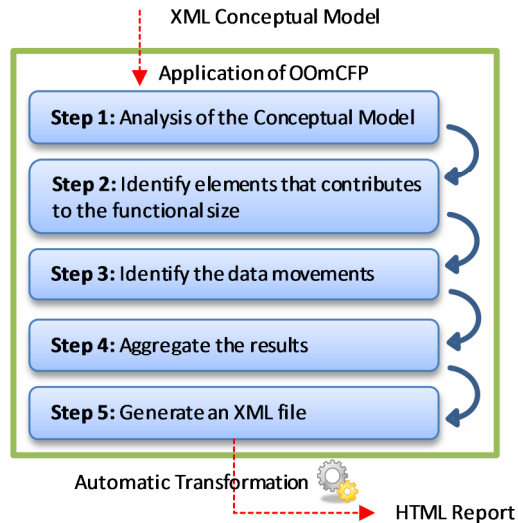


Figure 6.9. Analysis process in the OOmCFP tool.

To start (Step 1 of Figure 6.9), an XML file that represents the defined OO-Method conceptual model is loaded into the OOmCFP tool. This XML file is automatically created by the OO-Method modelling tool: the *Olivanova Suite* [CARE-Technologies 2011]. In this step, the OOmCFP tool identifies the functional users and the functional processes by applying Rules 1-10 of the OOmCFP measurement procedure.

In the next step (Step 2 of Figure 6.9), the tool identifies the elements that contribute to the functionality of the final system. In OOmCFP, the functional processes correspond to the interaction units defined in the presentation model of the OO-Method approach. Thus, for each interaction unit, the tool identifies the display sets, filters, and services that have contained. Additionally, if the interaction units contain other interaction units, these are recursively analyzed to identify the complete set of elements contained in each functional process. To do this, Rules 9.1, 9.2, 9.3, and 9.4 are applied to identify the elements that are contained in the interaction units. Also, Rules 11-16 of the OOmCFP measurement procedure are applied in

this step in order to eliminate the duplication in the functional processes analyzed and to identify the data groups and attributes.

Later, the data movements that occur between the functional users and the functional processes are identified (Step 3 of Figure 6.9). To do this, all 74 rules of OOmCFP are applied to the elements that participate in each functional process. To reduce the coupling in the identification of data movements, each rule has been implemented according to the conceptual element involved in the data movements that can occur in the OO-Method applications. Then, the result of the data movements identified of each element is stored in the same element.

In the next step (Step 4 of Figure 6.9), the tool aggregates the data movements identified according to the measurement rules 1-5 defined by the OOmCFP procedure. Thus, the tool obtains the functional size of each functional process, the functional size of each component of the application, and the functional size of the complete application that will be generated from the analyzed model.

When the step 4 finishes, the tool generates an XML file with the result obtained (Step 5 of Figure 6.9). This file contains the COSMIC functional size of the application that will be generated from the analyzed conceptual model, which has been obtained by means of the use of the OOmCFP measurement procedure. Finally, the tool transforms the generated XML file in a friendly format for the user. To do this, the tool applies XSLT transformations to obtain an HTML page or an Excel sheet from the generated XML file.

6.2.2 Efficiency in the Measurement

Using the OOmCFP tool, we have learned that the second and third step of the application of OOmCFP have the longest processing time. In these steps, this is produced due to the same modeling element must be analyzed several times. To improve this situation, we have implemented a cache mechanism

to reduce the time required to analyze elements that already have been analyzed. Thus, when a new functional element is identified, the cache mechanism verifies whether or not it already exists. If so, the value of the measurement is recovered.

In addition, we have implemented a mechanism to avoid the overflow in the execution of the tool, which may occur in the analysis of large models due to the iterative nature of the functional size measurement process. In the mechanism to avoid overflow, the related elements are stored in an auxiliary array (see Figure 6.10). Once the analysis of the first element finishes, the analysis of the elements stored in the auxiliary array continues sequentially. If the related elements are also related to other elements, these elements are added at the end of the auxiliary array, eliminating the loop of iterations and consequently avoiding overflow.

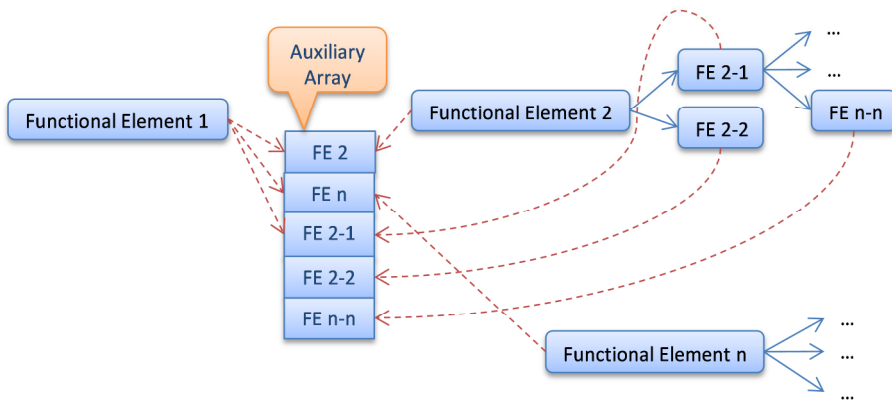


Figure 6.10 Schema of the solution to avoid overflow problems.

Therefore, the architecture of the OOmCFP tool and the mechanisms implemented provide an efficient measurement process. Thus, the measurement of conceptual models that generate real applications is done in few seconds, expediting the process of estimating the cost of the final application.

6.2.3 Using the OOmCFP Tool

The OOmCFP tool has been designed to be easily used. Thus, this tool has only three graphical user interfaces that allow the application of the OOmCFP measurement procedure to a specific conceptual model.

The first graphical user interface of the OOmCFP tool is shown in Figure 6.11. In this interface, the user must specify where is located the XML file with the representation of the conceptual model that will be measured (Source File). Also, the user must specify the location where the results of the measurement will be stored (Destination Directory). And finally, the user must select the scope of the measurement, which can be only the client layer, only the server layer, or the whole application.

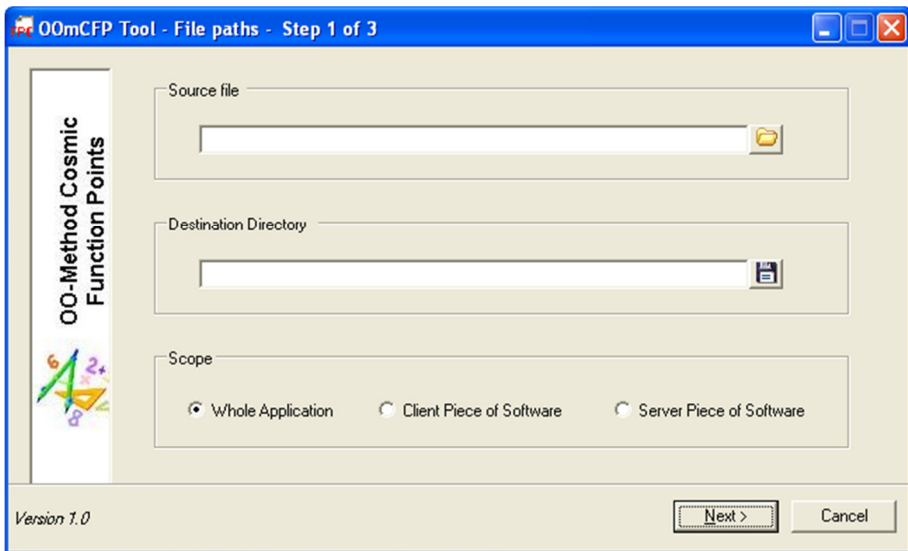


Figure 6.11 First Interface of OOmCFP tool.

With the information received from the first interface, the OOmCFP tool validates the following things:

1. The XML file loaded must correspond to an OO-Method conceptual model. If the file loaded does not correspond to an XML file, or it does not correspond to an OO-Method model, the OOmCFP tool shows an error-message that indicates that the user must select a file with the characteristics mentioned above.
2. The destination directory must exist. If it does not exist, the OOmCFP tool creates the directory. But, if the directory exists, the OOmCFP tool validates that the directory has permissions to write. In that case, the tool shows error-messages to the user.

The second graphical user interface of the OOmCFP tool is merely informative (see Figure 6.12). In this interface, the tool displays to the user the following information: the name of the model that will be measured, the path where the model is located, the path where the report will be located, and the scope of the measurement.

Then, the tool applies the OOmCFP measurement procedure to the XML file that contains the representation of the conceptual model. Thus, in the third graphical interface (see Figure 6.13), the OOmCFP tool shows the number of functional processes that have been measured and the function points for every layer of the application. In this step, the report with all the results of the measurement is saved in the path indicated in the first step.

Figure 6.14 shows the home page of the report generated by the OOmCFP tool to the rent-a-car system that was manually measured before. In this page it is possible to observe that the functional size obtained by the tool is the same than the functional size obtained by the manual application of OOmCFP.

6. Application of OOmCFP

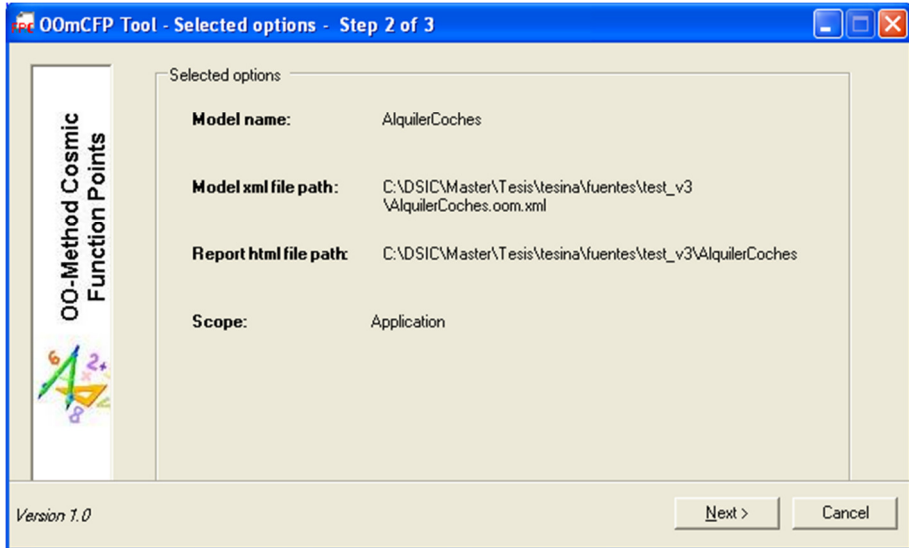


Figure 6.12 Second Interface of OOmCFP tool.

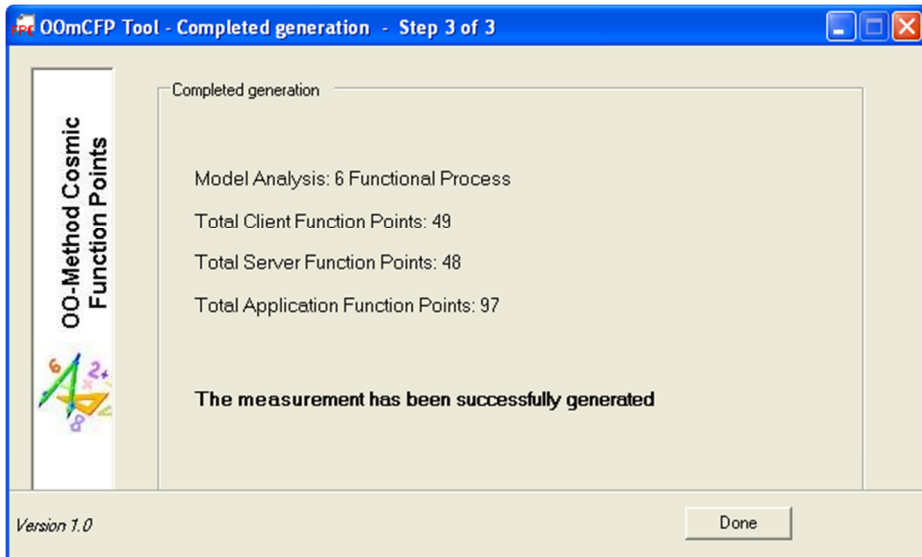


Figure 6.13 Third Interface of OOmCFP tool.

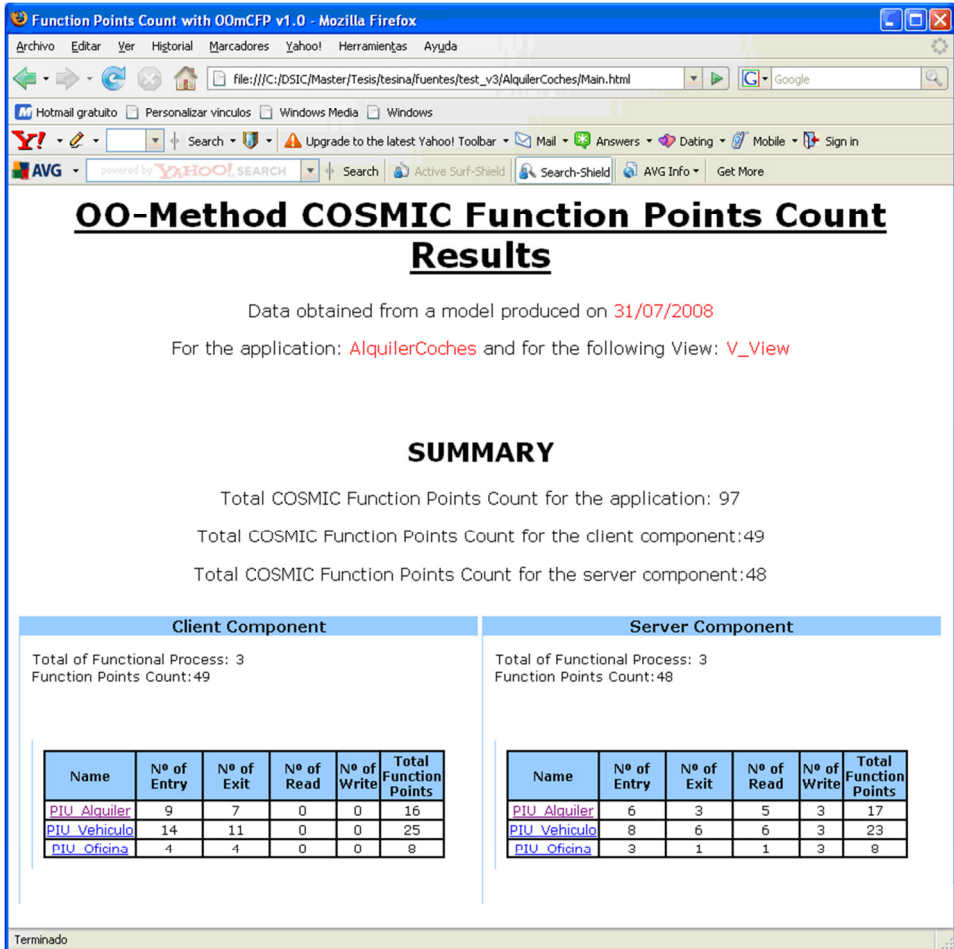


Figure 6.14 Main page of the measurement report for the rent-a-car application.

From this page, it is possible to navigate to the elements contained in each functional process (see Figure 6.15), and also, from each contained element it is possible to navigate to the data groups related to the data movements identified (see Figure 6.16).

6. Application of OOmCFP

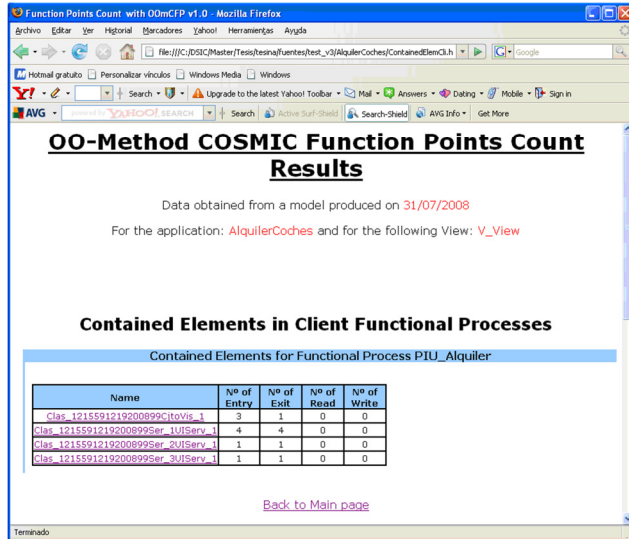


Figure 6.15 Data movements in the elements contained in the functional processes of the rent-a-car application.

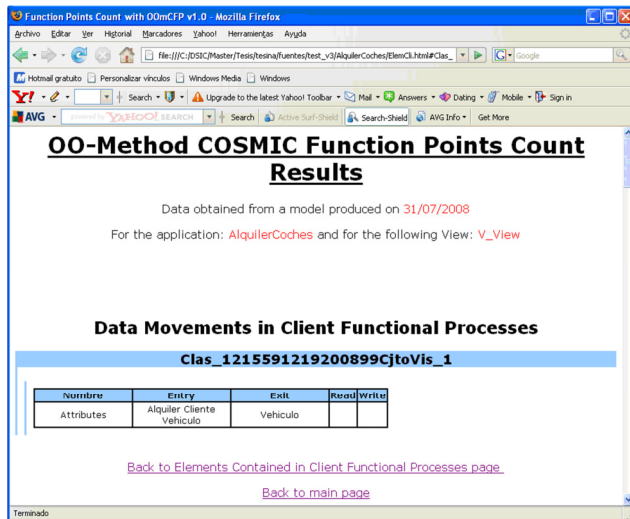


Figure 6.16 Data groups related to the data movements identified in the rent-a-car application.

6.2.4 Verification of the OOmCFP Tool

First of all, the OOmCFP tool was verified using unitary test cases to assure the correct implementation of each rule defined by the OOmCFP measurement procedure. The unitary test cases were defined by an expert in OOmCFP. This expert knows each rule defined in the OOmCFP measurement procedure and has applied the OOmCFP procedure to several conceptual models. In summary, 96 unitary test cases were defined in relation to the OOmCFP procedure, and 5 unitary test cases were defined in relation to the measurement report generated by OOmCFP.

Once all the unitary test cases were passed by the OOmCFP tool, the OOmCFP tool was verified using integrated testing. To do this, 28 test cases were defined. These 28 test cases cover the totality of OOmCFP rules. Thus, to measure these test cases, it is necessary to combine a set of rules defined in the OOmCFP measurement procedure with some instructions related to the generation of the measurement report.

Once all the integrated test cases were passed, a fully specified conceptual model was measured by the OOmCFP tool and the results were compared with the results obtained by the manual measurement. This conceptual model corresponded to the rent-a-car system. Once the OOmCFP tool obtains the same results than the manual measurement (97 CFP was obtained for the whole rent-a-car application), we consider that the OOmCFP tool has been verified.

To illustrate the application of the OOmCFP tool, 6 conceptual models of real projects were also measured. In order to maintain the confidentiality of the applications evaluated, we assigned the code Model1, Model2, Model3, Model4, Model5, and Model6 to each model. Table 21 shows the measurement results obtained by the OOmCFP tool. This table shows the functional size of each layer of the application (i.e., client and server layer). Some applications are modeled only to automatically generate the client layer of an application, which is used to facilitate working with ERPs (server

layer). In this context, counting with the functional size of each layer of an application allows the correct budget estimation of these applications.

Table 21. Results obtained by the OOmCFP tool for 6 conceptual models of real projects.

Model	Number of Functional Processes	Functional Size of the Client Layer (CFP)	Functional Size of the Server Layer (CFP)	Total Functional Size (CFP)	Measurement Time (seconds)
Model1	30	201	174	375	3 seconds
Model2	2	25	21	46	3 seconds
Model3	38	435	400	835	4 seconds
Model4	18	489	481	970	6 seconds
Model5	192	1491	1395	2886	55 seconds
Model6	108	1838	1759	3597	45 seconds

In addition, we have tested the processing time of the OOmCFP tool using the same 6 models related to real projects. The OOmCFP tool has been implemented in several layers in order to diminish the processing time of the measurement. Nevertheless, results of testing have demonstrated that the mayor processing time is used to analyze elements that already have been analyzed. Thus, we have implemented a cache mechanism in the OOmCFP tool in order to reduce the time required to analyze the elements that are used to measure the functional size.

Table 22 shows the number of classes of each model, the size of the XML representation of the models measured in KB, the initial response time of OOmCFP, and the response time of OOmCFP after the improvements of the OOmCFP tool. In the improved response time column of Table 22, it is possible to observe that the maximum response time was 55 seconds, which is understandable for very large models.

Table 22. Models used to test the performance of the OOmCFP tool.

Model	Number of Classes	Size (KB)	Initial Response Time	Improved Response Time
Model1	7	476	58 seconds	3 seconds
Model2	17	966	2 minutes 16 seconds	3 seconds
Model3	17	995	3 minutes 53 seconds	4 seconds
Model4	9	2446	4 minutes 41 seconds	6 seconds
Model5	87	9526	11 minutes 27 seconds	55 seconds
Model6	71	10689	9 minutes 49 seconds	45 seconds

Finally, Table 21 shows that some real applications have high values of functional size (such as Model7 and Model8). For these applications, manual measurement of the functional size could need too much time and, consequently, too many resources. Since the measurement has been performed using the OOmCFP tool, these measurement results have been obtained in few seconds and avoiding the introduction of errors due to the big amount of functional processes and data movements that must be identified.

6.3 Conclusions

This chapter has illustrated the manual and the automatic application of the OOmCFP measurement procedure to a rent-a-car system.

The manual application of the OOmCFP procedure has obtained 97 COSMIC Function Points. However, we can identify some validation threats for the results obtained; for instance, one threat is the natural variation in human performance because a human may erroneously identify the data movements that occur in an application (duplicates, omissions, etc.). To

avoid this validation threat for the results obtained, we have implemented a tool that automates the measurement procedure.

The OOmCFP tool has been developed taking into account a set of aspects related to the performance of the functional measurement process and implementation aspects. The performance aspects are focused on the reduction of the measurement time. The implementation aspects consider a correct execution of the OOmCFP measurement procedure avoiding the overflows that can be produced by the measurement of large OO-Method conceptual models. We have verified how the OOmCFP tool works in practice using some predefined OO-Method conceptual models. Finally, the measurement of some real conceptual models by the OOmCFP tool has been presented.

Chapter 7

Evaluation of the Application of OOmCFP

The OOmCFP measurement procedure has been designed to obtain accurate measurement results of MDD applications that have been generated from their conceptual models. The design of OOmCFP has been validated according to its conformity with the COSMIC measurement method, and its concepts have also been validated according to metrology. Moreover, the measurement instrument designed for the application of OOmCFP has been validated regarding its precision. However, having a valid design of the OOmCFP measurement procedure do not implies that the results obtained by the application of OOmCFP be accurate. Thus, this chapter presents the evaluation of the application of OOmCFP that analyzes in depth the results obtained by this measurement procedure.

The validation of the application of OOmCFP has been carried out according to metrology, and also, it has been carried out in terms of the precision and the accuracy of the measurement results obtained by OOmCFP by using the standard that evaluates the accuracy of measurement methods ISO 5725-2 [ISO 1994].

7.1 Metrology Evaluation of the Application of OOmCFP

Metrology is defined in the VIM [ISO 2004] as “*field of knowledge concerned with measurement*”. As has been stated before, metrology includes theoretical and practical aspects of measurements. Theoretical aspects have been used to validate the design of OOmCFP. Thus, to validate the application of OOmCFP, practical aspects of the measurement must be taken into account. To do this, we use a high-level model of the terms and categories of VIM presented by [Abran and Sellami 2002] (see Figure 7.1).

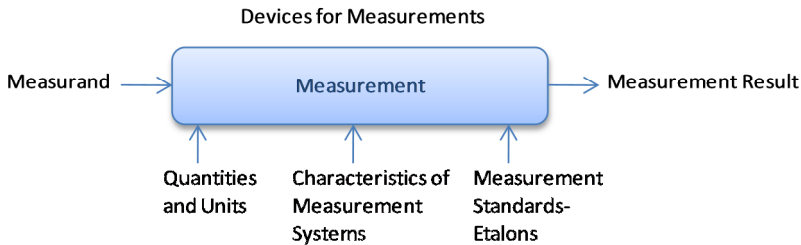


Figure 7.1 High-level model of categories of metrology terms [Abran and Sellami 2002].

Taking into account these sets of metrology concepts, the quality criteria that must be accomplished for the application and the results of a measurement method have been specified in the book *Software Metrics and Software Metrology* [Abran 2010]. Thus, the application and the results of a measurement method are considered ‘good’ when they refer to these quality criteria. Even though the set of measurement concepts is made up by the measurement foundation and the measurement procedure, Table 23 shows that to validate the application of OOmCFP, only the measurement procedure must be taken into account. Following these set of concepts, the

analysis of the application of OOmCFP according to metrology is presented next.

Table 23. Quality criteria for the application of a measurement method according to metrology concepts.

Step in Process Model	Quality Criteria
Application of the Measurement Method	Measurement Procedure
	Devices for Measurement
	Operations with Devices
	Properties of Measuring Devices

7.1.1 Measurement Procedure

A measurement procedure is a detailed description of a measurement according to one or more measurement principles and to a given measurement method [ISO 2004]. The application of a measurement procedure involves the following concepts (see Figure 7.2): Measurand, Operator, Measurement Procedure, Measurement Principle, Measurement Method, Measurement Model, Measured Quantity Value, Influence Quantity, and Measurement Result.

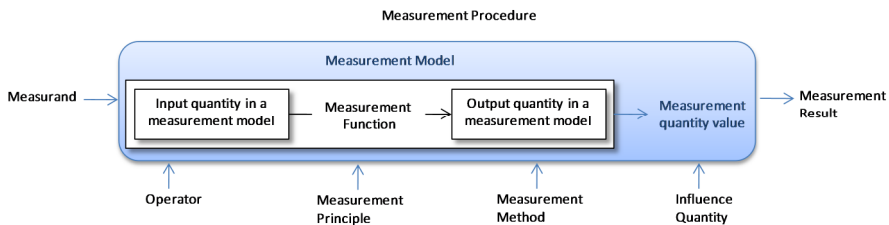


Figure 7.2 Topology of measurement procedure.

The *measurand* corresponds to the object to be measured. In OOmCFP this object correspond to the OO-Method conceptual model that is used as input in the measurement.

The *operator* is a subject that uses OOmCFP to carry out a specific measurement. The operator can apply the OOmCFP FSM procedure in a manual way by using the OOmCFP measurement guide (see Appendix A), as well as an automated way using the OOmCFP tool. In both applications manual and automated, the operators must know the constructs of the OO-Method conceptual model.

The *measurement principle* specifies that the functional size is directly proportional to the number of data movements that occur in the system according to the ISO/IEC 19761 [ISO/IEC 2003a]. The measurement method corresponds to the COSMIC FSM method version 3.0 [Abran *et al.* 2007].

The *influence quantity* corresponds to a quantity, whose change affects the OOmCFP measurement process. Thus, since some quality attributes of the OO-Method conceptual models used to perform the measurement affect positively or negatively the measurement (such as traceability, functional completeness, modifiability, etc.), the quality of the OO-Method conceptual model specifications was considered as the influence quantity.

The instantiation of a measurement procedure handles a measurement model that represents a mathematical relation among the quantities involved in a measurement. Therefore, the *measurement model* for OOmCFP corresponds to the model obtained from the application of the rules defined by OOmCFP to the conceptual constructs of the OO-Method conceptual model that contribute to the functionality of the final applications.

The *measurement quantity value* corresponds to the set of data movements identified in the construction of the measurement model. Finally, the *measurement result* is related to the attribute to be measured (i.e.; the functional size), which is quantitatively expressed in CFP units.

In summary, from the analysis of OOmCFP according to the measurement procedure, we can state that the application of the OOmCFP measurement procedure is valid according to the metrology concepts.

7.1.2 Devices for Measurement

A measuring instrument is a device or a combination of devices that are designed for the measurement of quantities. The OOmCFP procedure has been implemented in a tool, which corresponds to the measuring instrument of OOmCFP. A measurement instrument can be related to the following characteristics (see Figure 7.3): an indicating measuring instrument, a displaying measuring instrument, a scale of a displaying measuring instrument, and a material measure.

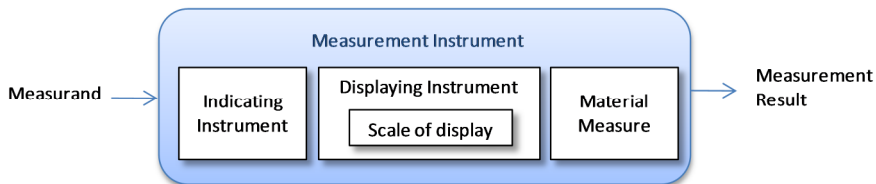


Figure 7.3 Topology of measurement instrument.

An *indicating measuring instrument* corresponds to a measuring instrument providing an output signal carrying information about the value of the quantity to be measured [ISO 2004]. The OOmCFP tool provides the COSMIC Functional Size of the functional processes of the client, the functional processes of the server layer, or the functional processes of the entire MDD application generated from the corresponding conceptual model. The last user interface of the OOmCFP tool shows the value of the quantity measured according to the elements selected for the measurement in the first user interface of OOmCFP. Thus, the OOmCFP tool corresponds to an indicating measurement instrument.

A *displaying measuring instrument* corresponds to a measuring device that shows the results of the measurements to observers. The OOmCFP tool displays a summary of the measurement results and it also store detailed information about the measurement results in an XML file. This XML file is also used by the OOmCFP tool to generate a web page or an Excel sheet with a friendly format to explore the measurement results. Therefore, the OOmCFP tool also displays the detailed information of the measurement results obtained.

A *scale of a displaying measuring instrument* corresponds to the graduation of the information displayed by the tool. The OOmCFP tool displays the measurement results in three levels: the first level displays the COSMIC function points of the functional processes of each layer of an application, the second level displays the data movements that occur in each functional process, and the third level displays the data groups related to each data movement.

A *material measure* corresponds to an instrument that reproduces or supplies quantities of one or more kinds, each one with a value assigned. As we stated in Chapter 5, for OOmCFP, all the quantities correspond to the functional size kind. Moreover, the quantity value corresponds to a numerical discrete value, which its minimum value is 1 data movement and no fixed maximum values. The OOmCFP tool obtains the functional size and its associated value for each functional process of the OO-Method applications.

In summary, from the analysis of OOmCFP according to the devices for measurement, we can state that the application of the OOmCFP measurement device is valid according to the metrology concepts.

7.1.3 Operations with Devices

The use of a complex measuring system might require some operations, such as adjustments of the measuring system [Abran 2010]. The adjustments on a

measurement device are carried out in order to provide prescribed indications corresponding to given values of the quantities to be measured [ISO 2004].

The adjustments of the measuring system must not be confused with the calibration of the measuring system. For instance, zero adjustment of a measuring system provides a null indication corresponding to a null value of the quantity to be measured. This kind of adjustments has been implemented in the OOmCFP tool to verify the structure of the conceptual model to be measured. Thus, if the input XML file doesn't correspond to an OO-Method conceptual model, the OOmCFP tool shows an alert message to the user. The OOmCFP tool also has implemented zero adjustments for the directory where the measurement results will be recorded. Thus, if the user doesn't have permissions to write the selected directory or even the directory doesn't exist, the OOmCFP tool shows an alert message to the user.

In the strategy phase, there is no selection of the functional users or the triggering events in the OOmCFP tool. Both functional users and triggering events are identified by the OOmCFP tool automatically from the conceptual models. Thus, the OOmCFP tool doesn't have zero adjustments for the strategy phase of the measurement. Nevertheless, if it necessary that the user select only some functional users or triggering events of the generated application, it can be implemented for future versions of the OOmCFP tool.

Regarding to the mapping and the measurement phases, since we have specified that the quantity values provided by OOmCFP correspond to numerical discrete values have 1 as the minimal value assigned, applying the OOmCFP rules there is no null values related to the quantities to be measured. Thus, the OOmCFP tool does not require zero adjustments for these phases.

In summary, from the analysis of OOmCFP according to the operations with devices, we can state that the application of the OOmCFP measurement device is valid according to the metrology concepts.

7.1.4 Properties of Measuring Devices

The properties of a measurement instrument correspond to qualitative and quantitative characteristics that affect the quality of the measures obtained by the instrument, such as the stability.

The *stability* of a measuring system corresponds to its ability to maintain its metrological characteristics constant in time [ISO 2004]. The design and the application of OOmCFP have been carefully carried out and validated according to metrology. Although the OOmCFP measurement procedure is stable, the manual application of the procedure depends on human actions; therefore, it might not be always stable. However, the automatic application of OOmCFP ensures the stability. Thus, since the OOmCFP tool implements the OOmCFP measurement procedure considering all the characteristics of OOmCFP, we can infer that the OOmCFP tool is stable.

Finally, from the analysis of OOmCFP according to the measurement procedure, devices for measurement, operations with devices, and properties of measuring devices; we can state that the application of the OOmCFP measurement procedure is valid according to metrology concepts.

7.2 Precision Evaluation of the Application of OOmCFP

To evaluate the precision of the measures obtained by the OOmCFP procedure, we carried out an experiment by applying the precision evaluation method defined in Chapter 5.

The goal of the experiment was defined using the Goal/Question/Metric (GQM) template [Basili and Rombach 1988] as: “To analyze the OOmCFP measurement procedure and the instruments used for the measurement

exercise for the purpose of evaluating its precision from the viewpoint of the researcher in the context of Computer Science students measuring OO-Method conceptual models with OOmCFP”.

The experiment correspond to a laboratory experiment [Wohlin *et al.* 2000]. The following research question was addressed by the experiment:

RQ1: Can OOmCFP obtain precise measurement results?

In order to answer this question, one independent variable and two dependent variables have been considered. The *independent variables* are those whose values are controlled or selected to determine their relationships to an observed phenomenon. Thus, the independent variable of this experiment corresponds to the OOmCFP measurement procedure. The *dependent variables* are the events studied and expected to change when the independent variable is changed. There are two dependent variables used to evaluate the precision of OOmCFP:

REPE, which corresponds to the repeatability of the measurement results obtained by applying OOmCFP.

REPRO, which corresponds to the reproducibility of the measurement results obtained by OOmCFP.

Repeatability conditions means that the same measurement results are obtained using the same measurement procedure, the same subject, the same measuring system, the same operating conditions and the same location [ISO 2004]. Reproducibility conditions means that the same measurement results are obtained changing some conditions mentioned before, for instance, changing the subject [ISO 2004].

Taking into account these variables, the following hypotheses have been formulated:

H_{REPE}: OOmCFP obtains repeatable measurement results.

H_{REPRO}: OOmCFP obtains reproducible measurement results.

7.2.1 The Definition Phase of the Experiment

The subjects were characterized as people with some knowledge of OOmCFP but with knowledge of the OO-Method approach and the Olivanova Modeler tool [CARE-Technologies 2011]. The place was characterized as a room with sufficient computers for the subjects. The computers must have Windows as Operative System, the Olivanova Modeler tool to work with OO-Method conceptual models, and Microsoft Office installed.

Since the subjects have got some knowledge of the OOmCFP procedure and the instruments to perform the measurement, training of subjects was not necessary. Consequently, training instruments were not prepared.

The instruments for the measurement exercise were prepared in this phase. To do this, all the lessons learned from the pilot study (see Chapter 5) were taken into account to improve the OOmCFP measurement instruments. Thus, the instruments for the measurement exercise were the following: three OO-Method conceptual models with different levels of functional size (small, medium, and large), the instructions for the measurement exercise, the OOmCFP measurement guide, and the results recording sheet.

The conceptual models used for the measurement exercise were the following: a Flight Reservation application (small – three classes); a Publishing application (medium – 5 classes); and a Rent-a-Car application (large – 8 classes).

The instructions were structured in two parts: (1) the first part contains 10 instructions for each model to open the model, the measurement instruments, and to save the measurement results; (2) the second part contains the same 10 instructions for each model for the repetition of the measurement process.

There were six results sheets, one for each measurement task. All the experimental instruments (conceptual models, OOmCFP measurement

guide, instructions and results sheets) were validated by two experts in OO-Method and one expert in OOmCFP.

7.2.2 The Measurement Phase of the Experiment

The subjects were selected from the students enrolled in the “Master’s Degree in Software Engineering, Formal Methods, and Information Systems” at the Technical University of Valencia from September 2006 to September 2008. The group of students was made up of 6 students with some knowledge of the OO-Method conceptual model, the Olivanova tool, and the OOmCFP procedure.

The place selected was Room 0S02 of the Department of Information Systems and Computation of the Technical University of Valencia. This room has twenty identical computers with the same programs installed. Each computer has the Windows OS, the Olivanova Modeler tool, and the Microsoft Office software already installed. These computers have also the same versions of the software installed.

The installation of the instruments in the classroom consisted in copying the measurement instruments in six computers of this room. Also in this activity the measurement guide and the instructions were printed and located close to these computers.

The measurement exercise activity was planned in 3 hours each part (measurement itself and repetition of the measurement): 20 minutes for the measurement of the small model, 60 minutes for the medium model, and 100 minutes for the large model.

The entire measurement exercise was carried out two days of the same week: in the first day, the measurement process was carried out, and in the second day, the repetition of the measurement process was carried out.

7.2.3 The Evaluation Phase of the Experiment

The results obtained from the measurement exercise in each results sheet of each subject was recorded in Table 24. The small, medium, and large level corresponds to the small (Flight Reservation application), medium (Publishing application), and large (Rent-a-Car application) models, respectively. The measurement results presented in Table 24 are expressed in COSMIC Function Points (CFP).

Table 24. Results of the measurement exercise.

Subject	Level		
	Small	Medium	Large
1	25	95	98
	26	94	98
2	27	94	97
	27	95	96
3	26	95	95
	25	96	96
4	24	93	99
	25	93	97
5	27	96	99
	27	95	98
6	25	95	96
	26	95	96

Since there are no outliers or outlying subjects, all the measurement results registered in Table 24 are considered valid. Table 25 shows the mean of the cells of Table 24, which has been calculated using Formula 4 (i.e.,

$$\text{Cell}_{ij} = \frac{1}{n_{ij}} \sum_{k=0}^{n_{ij}} \text{Measure}_{ijk} .$$

Table 25. Arithmetic means of cells.

Subject	Level		
	Small	Medium	Large
1	25.5	94.5	98
2	27	94.5	96.5
3	25.5	95.5	95.5
4	24.5	93	98
5	27	95.5	98.5
6	25.5	95	96

Table 26 shows the spread of cells, which is calculated by applying Formula 5 of the precision evaluation method (i.e.,

$$Spread_{ij} = \sqrt{\frac{1}{n_{ij} - 1} \sum_{k=0}^{n_{ij}} (Measure_{ijk} - Cell_{ij})^2}.$$

Table 26. Spread of cells.

Subject	Level		
	Small	Medium	Large
1	0.7	0.7	0
2	0	0.7	0.7
3	0.7	0.7	0.7
4	0.7	0	1.4
5	0	0.7	0.7
6	0.7	0	0

After that, the precision is calculated for each level (small, medium, and large) using Formula 6 (i.e., $S^2_{rj} = \frac{\sum_{i=1}^p (n_{ij} - 1) Spread_{ij}^2}{\sum_{i=1}^p (n_{ij} - 1)}$) for the repeatability

variance and Formula 7 (i.e., $S^2_{Rj} = S^2_{rj} + S^2_{Lj}$) for the reproducibility

variance. Table 27 shows the repeatability variance and the reproducibility variance calculated for each level.

Table 27. Repeatability variance and reproducibility variance of each level.

Variance	Level		
	Small	Medium	Large
Repeatability Variance (S^2_{rj})	0.3	0.3	0.5
Reproducibility Variance (S^2_{Rj})	1.1	1	1.8

Using the formulae defined in the ISO 5725 standard for accuracy measures [ISO 1994], we have obtained the values for the repeatability variance and the reproducibility variance in models of three different size (small, medium, and large). These values represent the magnitudes of the expected measurement error within measurement results or between measurement results, respectively.

The repeatability variance of each model is minor than one CFP² (see Table 27): 0.3 CFP² for small and medium models; and 0.5 CFP² for large models. We use the standard deviation to interpret these results. The standard deviation of measures is obtained by applying square root to the variance. Thus, we obtained the following deviations: 0.5 CFP for small and medium models, and 0.7 CFP for large models. Taking into account that each data movement identified by OOmCFP correspond to 1 CFP, the expected measurement error within measurement results obtained in the same conditions (same subject, same measurement procedure, same instruments, and same instructions) is minimal. Thus, we considered that the data obtained satisfy the hypothesis H_{REPE} : OOmCFP obtains repeatable measurement results. These results also indicate that the measurement instruments used in the laboratory experiment are well-defined since it is possible to obtain repeatable measurement results.

The reproducibility variance of each model is around one CFP² (see Table 27): 1.1 CFP² for small models, 1 CFP² for medium models, and 1.8 CFP² for large models. To interpret these results, we calculate the standard deviation. Thus, we obtained the following standard deviations: 1 CFP for small models; 1 CFP for medium models; and 1.3 CFP for large models. This means that changing the subjects, the expected measurement error between the measurement results is close to 1 CFP. Thus, we considered that the data obtained satisfy the hypothesis **H_{REPRO}**: OOmCFP obtains reproducible measurement results. These results also indicate that the measurement procedure has been understood by the subjects since it is possible to reproduce the measurement results.

Taking into account that high precision is obtained by low repeatability variance and low reproducibility variance values (see Table 27), we can state that the OOmCFP obtains precise measurement results.

7.2.4 Validity of Experimental Results

Although the experiment results are valid for the target population, there are some threats to the validity of the results that we identified. There are three main aspects to describe the validity of a study: construct validity, internal validity, and external validity.

The *construct validity* reflects to what extent the variables that are studied really represent what the researchers have in mind and what is investigated according to the research questions. The following threats to the construct validity were identified in the laboratory experiment:

- Precision have different meanings. In the literature we can find different meanings for precision, and some authors even confuse precision with accuracy. To mitigate this threat, we have adopted the definition of precision of the ISO 5725 standard for the evaluation of the accuracy of measures [ISO 1994], which is widely used in other sciences.

- There no formulae to calculate precision of measures. The ISO 5725 standard [ISO 1994] defines precision according to the repeatability and the reproducibility of measures. Thus, to mitigate this threat, we have defined a method to evaluate the precision of measurement results taking into account the formulae for repeatability and reproducibility defined in the ISO 5725 standard.

The *internal validity* expresses the extent to which the design and analysis may have been compromised by the existence of confounding variables and other unexpected sources of bias. The following threats to the internal validity were identified in the laboratory experiment:

- Subjects cannot be suitable for the experiment. The experience of people with the OO-Method approach and its modeling tool affects the application of the OOmCFP measurement procedure. Thus, to mitigate this risk, we selected people with some knowledge of OO-Method and of its modeling tool.
- The OOmCFP measurement guide, the instructions, and the results sheet can be difficult to understand. To mitigate this threat, we conducted a pilot study to improve the instrumentation of the experiment, and also, we verified these instruments with experts of OO-Method and OOmCFP.
- The OOmCFP measurement procedure cannot be applied to real models. To mitigate this risk, we selected models of different size (small, medium, and large), the large model being similar to models of a real project.

The *external validity* is concerned with to what extent it is possible to generalize the findings, and to what extent the findings are of interest to other people outside the study. Taking into account these aspects of the

validity, the following threats to the external validity were identified in the laboratory experiment:

- The OOmCFP procedure cannot be applied to other MDD approaches. Even though the OOmCFP measurement procedure has been designed to measure OO-Method models, many conceptual constructs of the OO-Method MDD approach can be found in other MDD approaches based in UML diagrams. However, repeating this laboratory experiment with other MDD approach can give more information about the generalizations of the results.

7.3 Accuracy Evaluation of the Application of OOmCFP

In order to evaluate the accuracy of the OOmCFP measurement procedure (related to the closeness to the ‘true value’), precise measurement results must first be obtained.

Although we have evaluated the precision of the OOmCFP measurement procedure in a laboratory experiment, in general, it is not possible to ensure precision in manual measurements of real projects since people can make mistakes. For instance, people can make mistakes in the identification of the functional process, in the application of the measurement rules, or even in the application of the formulae.

In contrast, when a tool performs the measurement, it can ensure the precision of the results because it is an automated measurement where a precise procedure will always produce the same result in any measurement task. Consequently, the OOmCFP tool avoids the errors of the manual measurements and assures the precision of the measurements.

Taking into account that OOmCFP can obtain precise measurement results, we evaluate the accuracy of the results obtained by the OOmCFP tool. To do this, we considered the measurement results obtained by experts as the ‘true value’ of the measurement results.

Given that the OO-Method conceptual model allows the complete specification of the final applications in an abstract way, to evaluate the accuracy of OOmCFP we compare the measurement of six conceptual models using the OOmCFP tool with the measurement results of the respective generated applications that were obtained directly applying the COSMIC measurement method by three experts.

The OO-Method conceptual models represent the functionality of the following systems: airport, invoice, rent-a-car, publishing, photography agency, and expense report. The first four conceptual models were fully measured by experts. However, since the remaining two models correspond to real projects, they were only partially measured by experts. Table 28 shows the results obtained.

Table 28. Results obtained by the OOmCFP tool.

Model	Number of Classes	Functional Processes	Experts Functional Size (CFP)	OOmCFP Functional Size (CFP)
Airport	3	All	26	26
Invoice	4	All	108	108
Rent-a-Car	8	All	97	97
Publishing	5	All	95	95
Photography agency	17	Creation of a report	43	43
Expense Report	17	Expense management	46	46

Taking into account that the results obtained by the OOmCFP tool are the same than the results obtained by experts, which represent the ‘true value’ of the functional size of OO-Method applications, we can state that the OOmCFP tool obtain accurate measurement results for these applications from the corresponding conceptual models.

7.4 Conclusions

In this chapter, the evaluation of the application of OOmCFP has been presented.

A metrological analysis of the application of OOmCFP has been performed using VIM [ISO 2004]. This analysis shows that the concepts that were used in the application of OOmCFP are aligned with metrology terms.

In order to evaluate the precision of the OOmCFP measurement procedure, we applied a method defined according to ISO 5725 standard [ISO 1994] by means of a laboratory experiment. The results of the laboratory experiment confirmed the reproducibility and the repeatability of the OOmCFP measurement results, which means that OOmCFP can obtain precise measurement results.

In order to evaluate the accuracy of the OOmCFP measurement procedure (related to the closeness to the ‘true value’), results obtained by the OOmCFP tool were compared with results obtained by COSMIC experts. Results show that the OOmCFP tool obtains the same measurement results as experts, which means that OOmCFP can obtain accurate measurement results.

Thus, assuming that the conceptual model is of high quality (that is, the conceptual model is correct and complete), the OOmCFP procedure can obtain accurate measurement results of final applications from the corresponding conceptual models in a completely automated way, providing the measurement results in few minutes and using minimal resources. Thus,

7. Evaluation of the Application of OOmCFP

it is essential that the models used to perform the measurement do not present defects.

Chapter 8

Defect Detection in MDD

Conceptual Models

Software development methods are being continuously improved by researchers aiming at producing software at lower costs, in a faster way, and with a higher level of quality by reusing resources. *Model-Driven Development* (MDD) methods are targeting the same objectives [Hailpern and Tarr 2006]. MDD methods separate the business logic from the platform technologies in order to allow automatic generation of software through well-defined model transformations [Selic 2003]. To do this, MDD methods combine *Domain-Specific Modeling Languages* (DSMLs) [Selic 2007] and tools for model transformations and code generation to express domain concepts effectively and to alleviate the complexity of implementation platforms [Schmidt 2006].

In order to produce high quality software by using MDD methods, quality assurance techniques must be developed [Mohagheghi and Aagedal 2007] for the three main components of a MDD method: the models, the transformation engines, and the code generators. To guarantee the quality of final applications, quality assurance of transformation engines and

generators is very important. But, even more important is the quality assurance of models, since it directly affects in the model transformations and code generation.

Nevertheless, in the literature there is no consensus for the definition of quality of conceptual models. There are several proposals that use different terminologies to refer to the same concepts. There are also many proposals that do not even define what they mean by quality of conceptual models. In this thesis, we have adopted the definition of quality of conceptual models proposed by Moody [Moody 2005]. This definition is based on the definition of quality of a product or a service in the ISO 9000 standard [ISO 2000]. Thus, in this thesis, we understand the quality of a conceptual model to be “*The total of features and characteristics of a conceptual model that bear on its ability to satisfy stated or implied needs*”.

To evaluate the quality of conceptual models, many proposals have been developed from different perspectives [Moody 2005]: there are proposals that are based on theory [Lindland *et al.* 1994], experience [Davenport and Prusak 1998], the observation of defects in the conceptual models in order to induce quality characteristics [Neuman 2000], the evaluation of the quality characteristics defined in the ISO 9126 standard [ISO/IEC 2001] in conceptual models by means of measures [Genero *et al.* 2005] [Marín *et al.* 2007], a synthesis approach [Cherfi *et al.* 2002], etc.

Taking into account the advantages and disadvantages of each development perspective for quality frameworks [Moody 2005], defect detection is considered as a suitable approach because it provides a high level of empirical validity provided by the variety of conceptual models that are observed. However, it is interesting to note that this approach is not broadly used in the software engineering discipline, even though defect detection is the most common quality evaluation approach used by other disciplines such as health care [Wilson *et al.* 1995].

It has been asserted by Boehm [Boehm 1981] that the cost of fault correction increases exponentially over the development life cycle. Thus, it

is of paramount importance to discover faults as early as possible: this means detecting defects in the models of MDD approaches.

To develop an effective quality assurance technique, it is necessary to know what defects types may occur in conceptual models related to MDD approaches. Currently, there are some approaches that detect defects in conceptual models (such as [Lange and Chaudron 2007] and [Bellur and Vallieswaran 2006]), which are mostly focused on the detection of defects that come from either the data perspective (data models) or the process perspective (process models). However, defect detection has not been clearly accomplished from the interaction perspective (interaction models) even though all of these perspectives (data, process, and interaction modeling) are essential to specify a correct conceptual schema used in a MDD context [Vanderdonckt 2008].

Defect detection approaches are usually applied using reading techniques or rules (heuristics) defined from the experience of the researchers. However, the use of a single approach to find defects does not guarantee that all the defects will be found [Trudel and Abran 2008]. Thus, the use of several approaches is recommended in order to find as many defects as possible.

A Functional Size Measurement (FSM) method defines a set of rules to measure the size of software by quantifying the Functional User Requirements [ISO 1998]. To apply an FSM method to models, FSM procedures must be defined to provide a detailed description of the application of the FSM method [ISO 2004]. We advocate that a FSM procedure can be used to identify defects in the conceptual models of a MDD approach because it systematically analyzes all the conceptual constructs that participate in the system functionality. Thus, a FSM procedure corresponds to a new approach to detect defects in conceptual models, which can be combined with other defect detection approaches in order to improve the quality of models.

To formalize the defect detection in MDD conceptual models, the next sections present an approach that allows the automated verification of the conceptual models used in MDD environments with respect to defect types from the data, process, and interaction perspectives. To do that, a metamodel that formalizes the elements involved in the identification of the different defect types is defined using current metamodeling standards. Also, the use of the OOmCFP functional size measurement procedure to detect defects is presented.

8.1 A Metamodel for Defect Detection

In general terms, a metamodel is the artifact used to specify the abstract syntax of a modeling language and includes: the structural definition of the involved conceptual constructs with their properties, the definition of relationships among the different constructs, and the definition of a set of rules to control the interaction among the different constructs specified [Selic 2007].

In EMOF, a metamodel is represented by means of a class diagram, where each class of the diagram corresponds to a construct of the modeling language involved. We use the OCL specification [OMG 2006b] for the definition of the controlling rules of the metamodel since it is part of the OMG standards and it works with EMOF metamodels. Also, OCL rules provide a computable language for rule specification, which allows the defined rules to be automatically evaluated by existent tools such as Eclipse OCL tools [Eclipse 2011a].

Since MDD proposals select a set of conceptual constructs and aggregate others to specify the conceptual models, it is important to note that a great number of conceptual constructs increases the complexity of the specification of the models and may cause the introduction of more defects into the conceptual models. For this reason, the conceptual constructs of an

MDD proposal must be carefully selected so that the number of constructs that allow the complete specification of software applications at the conceptual abstraction level is as low as possible.

A metamodel can specify the constructs involved in the different types of defects as well as the properties that must be present in the different conceptual constructs for the detection of defects. In addition, the OCL language used for the metamodeling rules can be used to define specific rules to automate defect detection. Thus, we define a quality model that is comprised of two main elements: 1) a metamodel for the description of the conceptual constructs that are used in MDD environments (which includes all the properties involved in defect detection) and 2) a set of OCL rules that allows the automated detection of defects according to the list of defects detected using OOmCFP.

Figure 8.1 presents the metamodel for defect detection in MDD conceptual models. As this figure shows, a generic *ConceptualModel* of an MDD approach consists of a structural model (class *StructuralModel*), a behaviour model (class *BehaviourModel*), and an interaction model (class *PresentationModel*).

The structural model has a set of classes (class *Class*). Each class has several features (class *ClassFeature*), which can be services (class *Service*) or properties (class *Property*). In turn, the properties can be typed properties (class *TypedProperty*) or association ends (class *AssociationEnd*). The typed properties correspond to the attributes of a class, which must have a data type (class *DataType*) specified (attribute *Kind*). The typed properties can be derived (class *DerivedAttribute*) or not derived (class *Attribute*).

The services can be events (class *Event*), transactions (class *Transaction*), or operations (class *Operation*). The events have valuations (class *Valuation*) to change the value of the attributes of a class. Each service has a set of arguments (class *Argument*) with their corresponding types (class *Type*), and it can also have a set of preconditions (association *precondition*).

There are relationships between the classes of the model (represented by the class *RelationShip*), which can be associations (class *Association*), generalizations (class *Generalization*), and agents (class *Agent*). The agent definition is oriented to state the visibility and execution permissions over the classes of the defined model (association *agent*). The associations can be aggregations, compositions, or normal associations (attribute aggregation of the class *AssociationEnd*). Each class has a set of integrity constraints (association *integrityConstraint*). The classes, class features, arguments, and relationships must have a name (class *NamedElement*).

The derived attributes, services, preconditions, and integrity constraints require the specification of the functionality that they perform. This functionality is specified by means of the behaviour model. The behaviour model has elements (class *BehaviourElement*) that can be conditional elements (*ConditionalBehaviourElement*) or constraint elements (*Constraint*). The conditional elements correspond to formulae (class *Formula*) with a condition (association *condition*) and an effect (association *effect*). The constraint elements correspond to formulae (class *Formula*) with an error message (attribute *errorMsg*).

The formulae are defined (attribute *value*) by means of a particular language called OASIS, which is similar to the OCL language. Thus, the valuations and the specification of the derived attributes (class *ValueSpecification*) correspond to conditional behaviour elements, and the transactions and operations correspond to behaviour elements. The preconditions and the integrity constraints correspond to constraint behaviour elements.

The interaction model has a set of interaction units (class *InteractionUnit*) and a set of auxiliary patterns (class *AuxPattern*) that allow the specification of the graphical user interface at an abstract level. The interaction units can be instances (*InstanceIU*), set of instances (*PopulationIU*), services (*ServiceIU*), and composite units (*MasterDetailIU*).

The master-detail interaction units correspond to composite interaction units (class *DependentIU*), which are comprised of a master part (class *IndependentIU*) and a set of detail interaction units (class *DependentIU*). In the master part, only instances or populations can be used. In the detail part, instances, populations and other master detail interaction units can be used. Since, the instance interaction units and the population interaction units can be used independently of other interaction units, we classify them in the class *IndependentIU*. However, these interaction units can also be used inside the detail part of master detail interaction units, so we classify them in the class *DependentIU*.

The independent interaction units have display sets (class *DisplaySet*) to present the data. Each display pattern has a set of attributes (association *relatedattribute*) that are specified in the structural model, from which the data will be recovered to show the users of the application.

The independent interaction units can have actions (class *ActionSet*) to present the set of services (throw a *ServiceIU*) that can be executed by the users over the instances shown in the interaction units. In addition, the independent interaction units can have navigations (class *NavigationSet*) to present the interaction units that can be accessed.

The population interaction units can also have filters (class *Filter*) to search for information in a set of instances, which must be specified with the corresponding formula (class *Formula*). The service interaction units have entry (class *EntryPattern*) and selection patterns (class *SelectionPattern*), which have associated formulae composed of a condition (association *condition*) and an effect (association *effect*).

Since the metamodel has been specified using the standards of metamodeling, this metamodel eliminates redundancy of the elements defined and can be implemented using open-source modeling tools.

8. Defect Detection in MDD Conceptual Models

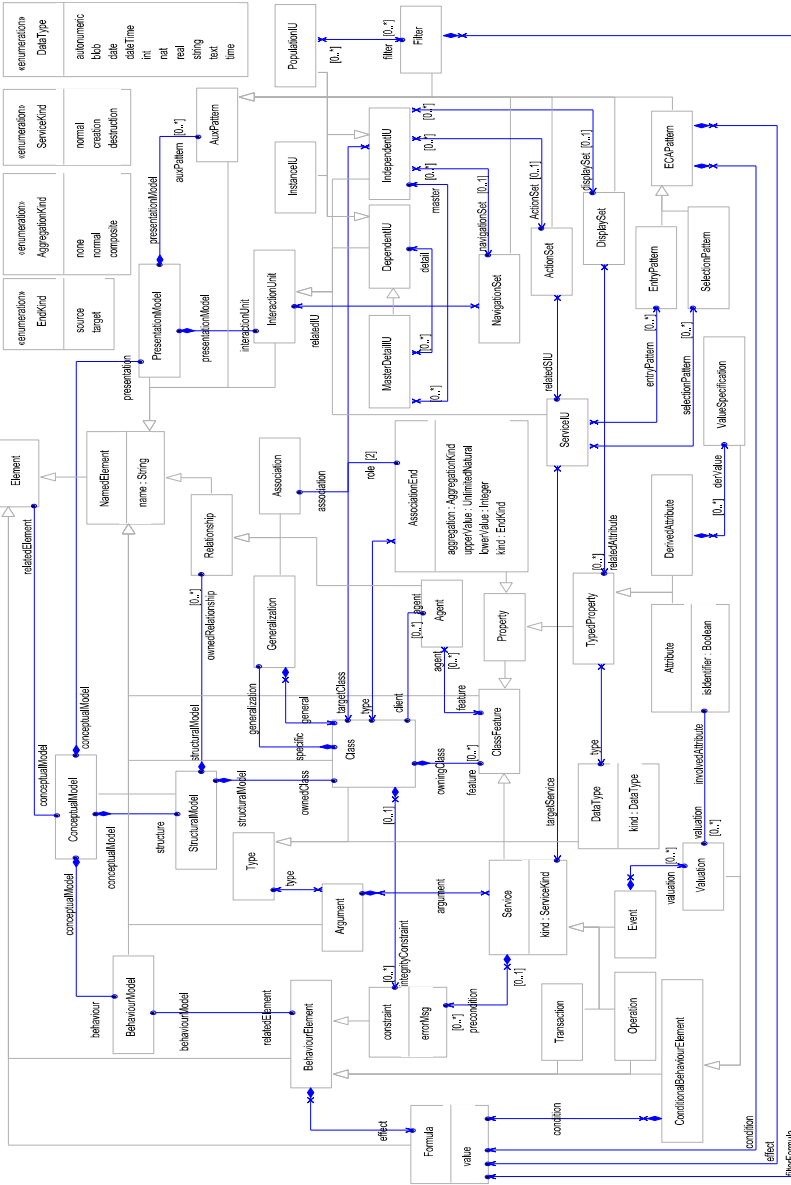


Figure 8.1 A metamodel for defects detection in conceptual models

8.2 Using OOmCFP to Detect Defects

Since the measurement of the functional size using the OOmCFP approach has defined rules to perform the mapping between the concepts of COSMIC and OO-Method, and rules to identify the data movements of the final application in the conceptual model; it is possible to identify some defects that impede the compilation of the conceptual model or that cause faults in the generated application.

In order to determine the defect types of MDD conceptual models, the proposed metamodel and the OOmCFP procedure were applied to three conceptual models of different functional sizes: a Publishing application (a small model); a Photography Agency application (a medium model); and an Expense Report application (a large model). In summary, we identified 39 different defects in these models, and we grouped them into 24 defect types. Table 29 presents a set of rules of the OOmCFP measurement procedure that are related to the mapping between COSMIC and OO-Method, and the defect types which can be found using these rules.

Based on the Conradi et al. proposal [Conradi *et al.* 2003], we classify the defect types into:

- Omission: missing item,
- Extraneous information: information that should not be in the model,
- Incorrect fact: misrepresentation of a fact,
- Ambiguity: unclear concept,
- Inconsistency: disagreement between representations of a concept.

Thus, Defects 1, 2, 3, 4, and 7 correspond to *omissions*; Defect 5 corresponds to an *incorrect fact*; and Defects 6 and 8 correspond to *ambiguities*.

Table 29. Defects related to mapping rules of OOmCFP.

COSMIC	OOmCFP	Defects
Functional User	<u>Rule 1:</u> Identify 1 functional user for each agent in the OO-Method object model.	<u>Defect 1:</u> An object model without a specification of an agent class.
Functional Process	<u>Rule 5:</u> Identify 1 functional process for each interaction unit that can be directly accessed in the menu of the OO-Method presentation model.	<u>Defect 2:</u> An OO-Method Conceptual Model without a definition of the presentation model. <u>Defect 3:</u> A presentation model without the specification of one or more interaction units.
Data Group	<u>Rule 6:</u> Identify 1 data group for each class defined in the OO-Method object model, which does not participate in an inheritance hierarchy.	<u>Defect 4:</u> An object model without the specifications of one or more classes. <u>Defect 5:</u> A class without a name. <u>Defect 6:</u> Classes with a repeated name.
Attributes	<u>Rule 9:</u> Identify the set attributes of the classes defined in the OO-Method object model.	<u>Defect 7:</u> A class without the definition of one or more attributes. <u>Defect 8:</u> A class with attributes with repeated names.

Table 30 presents a set of rules of the OOmCFP measurement procedure that are related to the identification of data movements in display patterns and filter patterns. This table also presents the defect types which can be found using the OOmCFP rules.

Table 30. Defects related to display and filter patterns OOmCFP rules.

OO-Method Conceptual Element	OOmCFP Rules	Defects
Display Pattern	<u>Rule 10:</u> Identify 1X data movement for the client piece of software for each display pattern in the interaction units that participate in a functional process.	<u>Defect 9:</u> An instance interaction unit without display pattern. <u>Defect 10:</u> A population interaction unit without display pattern.
	<u>Rule 11:</u> Identify 1E data movement for the client piece of software, and 1X and 1R data movements for the server piece of software for each different class that contributes with attributes to the display pattern.	<u>Defect 11:</u> A display pattern without attributes.
	<u>Rule 13:</u> Identify 1R data movement for the server piece of software for each different class that is used in the effect of the derivation formula of derivate attributes that appear in the display pattern.	<u>Defect 12:</u> Derived attributes without a derivation formula.
Filter Pattern	<u>Rule 16:</u> Identify 1R data movement for the server piece of software for each different class that is used in the filter formula of the filter patterns of the interaction units that participate in a functional process.	<u>Defect 13:</u> A filer without a filter formula.

Table 31 presents a set of rules of the OOmCFP measurement procedure that are related to the identification of data movements in services. This table also presents the defect types that can be found using these OOmCFP rules.

Table 31. Rules to identify the data movements of OOmCFP.

OO-Method Conceptual Element	OOmCFP Rules	Defects
Service	<p><u>Rule 20:</u> Identify 1R data movement for the server piece of software for each different class that is used in the effect of the valuation formula of events that participate in the interaction units contained in a functional process.</p>	<p><u>Defect 14:</u> An event of a class of the object diagram without valuations.</p>
	<p><u>Rule 21:</u> Identify 1W data movement for the server piece of software for each create event, destroy event, or event that has valuations (represented by the class that contains the service) that participate in the interaction units contained in a functional process.</p>	<p><u>Defect 15:</u> A class without a creation event.</p>
	<p><u>Rule 22:</u> Identify 1R data movement for the server piece of software for each different class that is used in the service formula of transactions, operations, or global services that participate in the interaction units contained in a functional process.</p>	<p><u>Defect 16:</u> Transactions without a specification of a sequence of services (service formula). <u>Defect 17:</u> Operations without a specification of a sequence of services (service formula). <u>Defect 18:</u> Global services without a specification of a sequence of services (service formula).</p>

Table 32 presents a set of rules of OOmCFP that are related to the identification of data movements in the arguments of a service and the defect types that can be found using these rules.

Table 32. Rules to identify the data movements of OOmCFP.

OO-Method Conceptual Element	OOmCFP Rules	Defects
Service	<p><u>Rule 23:</u> Identify 1E data movement and 1X data movement for the client piece of software, and 1E data movement for the server piece of software for the set of data-valued arguments of the services (represented by the class that contains the service) that participate in the interaction units contained in a functional process.</p> <p><u>Rule 24:</u> Identify 1E data movement and 1X data movement for the client piece of software, and 1E data movement for the server piece of software for each different object-valued argument of the services that participate in the interaction units contained in a functional process.</p>	<p><u>Defect 19:</u> A service without arguments.</p> <p><u>Defect 20:</u> A service with arguments with repeated names.</p>

Table 33 presents a set of rules of OOmCFP that are related to the identification of data movements in the preconditions defined for a service and the integrity constraints defined in the class that contain the service. This table also presents the defect types that can be found using these OOmCFP rules.

Table 33. Rules to identify the data movements of OOmCFP.

OO-Method Conceptual Element	OOmCFP Rules	Defects
Service	<u>Rule 31:</u> Identify 1R data movement for the server piece of software for each different class that is used in the precondition formulae of the services that participate in the interaction units contained in a functional process.	<u>Defect 21:</u> A precondition without the specification of the precondition formula.
	<u>Rule 32:</u> Identify 1X data movement for the client piece of software for all error messages of the precondition formulae of the services that participate in the interaction units contained in a functional process.	<u>Defect 22:</u> A precondition without an error message.
	<u>Rule 34:</u> Identify 1R data movement for the server piece of software for each different class that is used in the integrity constraint formulae of the class that contains each service that participates in the interaction units contained in a functional process.	<u>Defect 23:</u> An integrity constraint without the specification of the integrity formula.
	<u>Rule 35:</u> Identify 1X data movement for the client piece of software for all error messages of the integrity constraint formula of the class that contains each service that participates in the interaction units contained in a functional process.	<u>Defect 24:</u> An integrity constraint without an error message.

The list of defect types presented in Tables 30, 31, 32, and 33 have also been classified using the Conradi et al. [Conradi *et al.* 2003] classification. Thus, Defects 9, 10, 15, 19, 22, and 24 correspond to *omissions*; Defects 11, 12, 13, 14, 16, 17, 18, 21, and 23 correspond to an *incorrect fact*; and Defect 20 corresponds to an *ambiguity*. Therefore, we can state that the OOmCFP measurement procedure helps in the identification of defects types that are related to omissions, incorrect facts, and ambiguities of conceptual models.

It is important to note that Defects 1, 2, 4, 5, 6, 7, 8, 9, 10, 12, 13, 14, 16, 17, 18, 19, 21, and 23 allow the definition of measures that contribute to the evaluation of the sub-characteristic of *compliance* of the conceptual models (in accordance with the ISO 9126 standard), because it is possible to determine if the conceptual model adhered to the rules and conventions of the model compiler. In the same way, Defects 3, 11, 15, 20, 22, and 24 allow the definition of measures that contribute to the evaluation of the sub-characteristic of *analyzability* of software products (in accordance with the ISO 9126 standard), because it is possible to diagnostic the possible faults of the final application in the conceptual models.

The presented defect types are related to structural models and interaction models. This is one interesting contribution of using the OOmCFP measurement procedure to detect defects since, to the best of our knowledge, there are no reported findings of defect types related to interaction models in the published literature.

8.3 Formalization of Defect Detection Rules

In order to formalize the defect detection rules in the metamodel presented in Figure 8.1, we defined OCL rules to prevent the occurrence of the identified defects in the conceptual models. Table 34 shows the defect types presented in Table 29, and the corresponding OCL rules of our approach.

Table 34. 8 Defect Types of Conceptual Models found using OOmCFP and OCL Rules.

Defect Types found using OOmCFP	OCL Rules
Defect: An object model without a specification of an agent class.	context Agent inv: body self.allInstances->size()>0
Defect: An OO-Method Conceptual Model without a definition of the presentation model.	context ConceptualModel inv: body self.presentation->size()>0
Defect: A presentation model without the specification of one or more interaction units.	context PresentationModel inv: body self.interactionUnit->size()>0
Defect: An object model without the specifications of one or more classes.	context StructuralModel inv: body self.ownedClass->size()>0
Defect: A class without a name.	context Class inv: body Class.allInstances()->select(c c.name.isEmpty())->isEmpty()
Defect: Classes with a repeated name.	context Class inv: body self.allInstances()->forAll(c1, c2 c1 <> c2 implies c1.name <> c2.name)
Defect: A class without the definition of one or more attributes.	context Class inv: body self.features->select(t t.oclIsKindOf(TypedProperty))->collect(t t.oclAsType(TypedProperty))->size()>0
Defect: A class with attributes with repeated names.	context Class inv: body self.features->select(t t.oclIsKindOf(TypedProperty))->collect(t t.oclAsType(TypedProperty))->forAll(a1, a2 a1 <> a2 implies a1.name <> a2.name)

Table 35 shows the defect types presented in Tables 30, and 31; and the corresponding OCL rules of our approach.

Table 35. 9 Defect Types of Conceptual Models found using OOmCFP and OCL Rules.

Defect Types found using OOmCFP	OCL Rules
Defect: An instance interaction unit without a display pattern.	context InstanceIU inv: body self.displaySet->size(>0)
Defect: A population interaction unit without a display pattern.	context PopulationIU inv: body self.displaySet->size(>0)
Defect: A display pattern without attributes.	context DisplaySet inv: body self.relatedAttribute->size(>0)
Defect: Derived attributes without a derivation formula.	context DerivedAttribute inv: body self.derValue.effect->select(f f.value.isEmpty())->isEmpty()
Defect: A filter without a filter formula.	context Filter inv: body self.filterFormula->select(f f.value.isEmpty())->isEmpty()
Defect: An event of a class of the object diagram without valuations (excluding creation or destruction events).	context Event inv: body self.allInstances->select(e (e.kind <> ServiceKind::creation and e.kind <> ServiceKind::destruction) implies e.valuation.size() > 0)
Defect: A class without a creation event.	context Class inv: body self.features->select(s s.ocIsKindOf(Service))->collect(s s.ocAsType (Service))->select(s s.kind = ServiceKind::creation)->notEmpty()
Defect: Transactions without a specification of a sequence of services (service formula).	context Transaction inv: body self.effect->select(f f.value.isEmpty())->isEmpty()
Defect: Operations without a specification of a sequence of services (service formula).	context Operarion inv: body self.effect->select(f f.value.isEmpty())->isEmpty()

Table 36 shows the defect types presented in Tables 32, and 33; and the corresponding OCL rules of our approach.

Table 36. 6 Defect Types of Conceptual Models found using OOmCFP and OCL Rules.

Defect Types found using OOmCFP	OCL Rules
Defect: A service without arguments.	context Service inv : body self.argument->size()>0
Defect: A service with arguments with repeated names.	context Service inv : body self.argument->forall(a1, a2 a1 <> a2 implies a1.name <> a2.name)
Defect: A precondition without the specification of the precondition formula.	context Service inv : body self.precondition.effect->select(f f.value.isEmpty()->isEmpty())
Defect: A precondition without an error message.	context Service inv : body self.precondition->select (c c.errorMsg.isEmpty()->isEmpty())
Defect: An integrity constraint without the specification of the integrity formula.	context Constraint inv : body self.effect->select(f f.value.isEmpty()->isEmpty())
Defect: An integrity constraint without an error message.	context Constraint inv : body self.allInstances->select (c c.errorMsg.isEmpty()->isEmpty())

In order to identify the maximum number of defects using the proposed quality model, we aggregated the defect types already found in the literature (see Chapter 3) with the corresponding OCL rules (see Table 37). We selected the defect types related to the class model, which is a diagram commonly used by several MDD proposals. We ruled out the defect types of the literature that were also identified using the OOmCFP measurement procedure.

Table 37. 5 Defect Types of Conceptual Models found in the literature and OCL Rules.

Defects Types found in the literature	OCL Rules
Defect: An attribute of a class without the specification of the type.	context Class inv : body self.features->select(t t.oclIsKindOf(TypedProperty))->collect(t t.oclAsType(TypedProperty))->select(a a.type.isEmpty()->isEmpty())
Defect: An argument of a service without the specification of the type.	context Service inv : body self.features->select(s s.oclIsKindOf(Service))->collect(s s.oclAsType(Service)).argument.type->size<1->isEmpty()
Defect: Associations replicated at sub-classes.	Classifier::parents (): Set(Classifier); parents = generalization.general Classifier::allParents (): Set(Classifier); allParents = self.parents()->union(self.parents()->collect(p p.allParents())) context AssociationEnd inv : body self.allInstances->forAll(r1, r2 r1.name = r2.name and r1.owningClass.allParents()->select(c c.name = r2.name)->isEmpty())
Defect: Associations with a repeated name.	context Relationship inv : body self.allInstances()->forAll(r1, r2 r1 <> r2 implies r1.name <> r2.name)
Defect: An association without a source and target class.	context Association inv : body self.role->select(e1,e2 e1.role.kind = EndKind::source and e2.role.kind = EndKind::target)

In the three empirical studies performed using the proposed quality model, the conceptual models did not achieve the characteristics of

consistency and correctness due to the defect types presented in our approach. Thus, the OCL rules presented in Tables 34, 35, 36, and 37 can be implemented for the model compilers of MDD proposals in order to automatically verify the conceptual models with regard to these characteristics.

8.4 A Defect Detection Tool

Defect detection in conceptual models is often manually performed by an inspection team using reading techniques [Conradi *et al.* 2003] [Laitenberger *et al.* 2000] [Travassos *et al.* 1999]. However, the manual inspection of models takes a lot of time, which increases the costs and the delivery date of software products.

Since the cost of removing defects and enhancing designs increases with the stages of the life-cycle [Boehm 1981], for industrial MDD developments is essential to find defects in a quick and precise way. Thus, a tool that automates the defect detection in the models is needed to avoid the excessive time and the precision errors involved in a manual detection process.

There are three main quality characteristics that a model must satisfy: completeness¹[Lindland *et al.* 1994], correctness² [IEEE 1990], and consistency³ [IEEE 1990]. Since the completeness is related to the semantics of the model, it needs to be evaluated by an inspection team taking into account the requirements of the system. However, the correctness and the

¹ Completeness is defined by Lindland *et al.* as the degree to which a model contains all the statements about the domain that are correct and relevant.

² Correctness is defined in the IEEE 610 standard as the degree to which a system or component is free from faults in its specification, design, and implementation.

³ Consistency is defined in the IEEE 610 standard as the degree of uniformity, standardization, and freedom from contradiction among the documents or parts of a system or component.

consistency of models can be automatically evaluated by a tool, reducing the number of defects that must be found manually by the inspection team.

The defect types that a tool can detect depend on the DSML used to define the involved models and the process applied to perform the defect detection. The procedure that we have chosen for the identification of defects in OO-Method models is the Functional Size Measurement (FSM) procedure called OOmCFP.

8.4.1 The OOmCFP Tool

The OOmCFP tool has been updated to detect defects in OO-Method Models. Thus, the process of the OOmCFP tool is presented in Figure 8.2.

To start (Step 1 of Figure 8.2), an XML file that represents the defined OO-Method model is loaded into the defect detection tool. Then, the defect detection tool identifies the functional users and the functional processes by applying rules 1-10 of the OOmCFP measurement procedure. If one or more problems arise during the application of these rules, then, one defect is stored for each detected problem. The defects that may arise in this stage correspond to defect types 1-3 of Table 29.

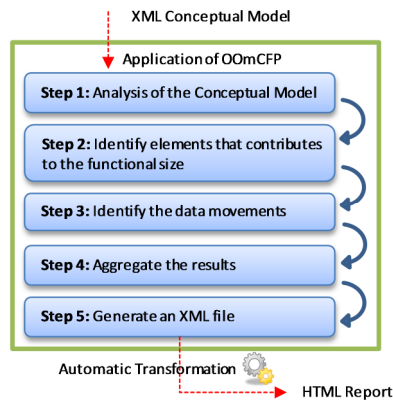


Figure 8.2 Process of the defect detection tool

In the next step (Step 2 of Figure 8.2), the tool identifies the elements that contribute to the functionality of the final system. As the same as the first step, one defect is stored for each problem produced during the application of the OOmCFP rules 11-16. The defects that can be found in this stage correspond to defect types 4-8 of Table 29.

Later, the data movements are identified (Step 3 of Figure 8.2). To do this, all 74 rules of OOmCFP are applied. To reduce the coupling in the identification of data movements, each rule has been implemented according to the conceptual element involved in the data movements. If these rules cannot be applied, the tool stores the defects related to the involved modeling elements in order to present detailed information in the final report. These defects correspond to defect types 9-24 in Table 30.

In the next step (Step 4 of Figure 8.2), the tool aggregates the results obtained by analyzing the stored defects. However, if the model does not present any defect (there are not stored defects), the tool applies the measurement rules to calculate the functional size of the application that will be generated from this model.

When the step 4 finishes, the tool generates an XML file with the result obtained (Step 5 of Figure 8.2). This file contains (1) the list of defects identified, which has information of the involved modeling elements and the corresponding defects; or (2) the COSMIC functional size obtained by the application of the OOmCFP measurement procedure.

Finally, the tool transforms the generated XML file in a friendly format for the user. To do this, the tool applies XSLT transformations to obtain an HTML page or and Excel sheet from the generated XML file. Figure 8.3 shows a screenshot of the tool and the HTML report generated for a model with defects.

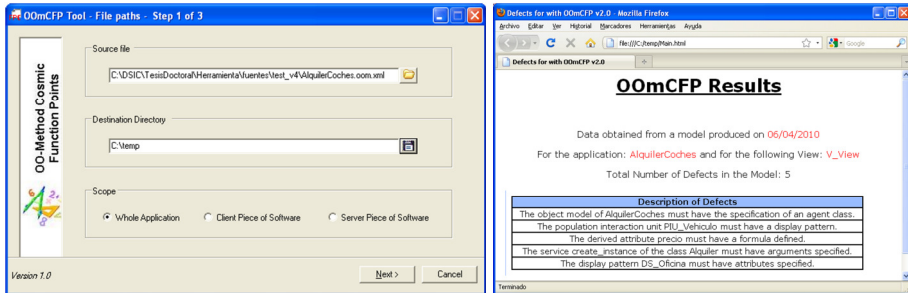


Figure 8.3 Screenshot of the tool and HTML defects report

8.5 Conclusions

For many years, software industry has applied different techniques for the requirement modeling and definition of conceptual models in order to identify and correct software defects. Otherwise, these defects could propagate to later development phases, which imply an extra cost to fix them. This situation is also present in the new software production processes, such as MDD methods. Therefore, it is very important to use different techniques in order to found defects in the conceptual models, avoiding their propagation to the final application.

Since in MDD approaches the quality of conceptual models has a direct impact in the quality of generated applications, the use of the OOmCFP measurement procedure for defects detection provides a new technique to improve the quality of conceptual models, and hence, the quality of final applications.

The defect types presented in this chapter were identified by applying the OOmCFP FSM procedure to different case studies of the OO-Method approach, which have been selected because they (all together) cover all the modeling possibilities of the OO-Method approach.

We take advantage of modeling, metamodeling, and transformation techniques to avoid having to manually identify defects in the conceptual models, which is an error-prone activity. Thus, a quality model (metamodel + OCL rules) has been designed for its easy application to other MDD proposals. This is feasible because the EMOF standard is used to define the metamodel, which is supported by existent open-source tools [Eclipse 2011a] [Eclipse 2011b] and is also used by other MDD proposals for the specification of their modeling languages. Therefore, we can state that the quality model proposed here contributes substantially to improving the MDD processes and the quality of software products generated in this context.

Chapter 9

Defect Detection Case Study

In the literature, there is no consensus about the definition of case study [Runeson and Host 2009]. There are several proposals that use the term case study to refer to well-organized studies in the field or even to refer to small toy examples. There are also many software engineering proposals that use the case study methodology following the guidelines of social sciences [Robson 2002] [Yin 2003] or the guidelines of information systems [Benbasat *et al.* 1987].

In this work, we use the definition provided for software engineering case studies by [Runeson and Host 2009], which states that a case study is an empirical method aimed at investigating contemporary phenomena in their context. Thus, in this case, the phenomena correspond to the defects of conceptual models used to generate final applications, and the context corresponds to the industrial MDD approach with the corresponding industrial modeling tool that allows the generation of final applications from the conceptual models.

In this chapter, we present a case study that aims to evaluate the use of an FSM procedure to detect defects in models of an MDD approach. To do this, the study compares the defects detected by an inspection team and the defects detected by an FSM procedure. Moreover, we determine the types of

the defects found and the implications that these defects have on the quality of the involved models.

We plan, conduct and report the case study following the guidelines presented by Runeson et al. [Runeson and Host 2009]. These guidelines indicate that to planning a case study it is necessary to define the objective (what to achieve?), the case (what is studied?), the theory (or frame of reference), the research questions (what to know?), the methods (how to collect data?), and the selection strategy (where to seek data?). These guidelines also indicate that to conducting a case study it is necessary to have the design of the case study, prepare the data collection procedures, collect evidence by executing the studied case, analyze the collected data, and report the study. Finally, these guidelines indicate that to reporting a case study it is necessary to report the related work (context and early studies), the design of the case study (which includes the research questions, case and subjects selection, data collection procedures, analysis procedures, and validity procedures), the results of the case study (which includes case and subjects description, covering execution, analysis and interpretation issues), and the conclusions and limitations of the study. We have rigorously followed these guidelines, presenting the design and the results of the case study in this chapter.

9.1 Design of the Case Study

This section presents the design of a case study, which includes the objective of the study, the research questions, the case and subject selection, the data collection procedures, the analysis procedures, and the validity procedures.

The case study corresponds to an exploratory study about the defects and the defect types that are found in the models of the OO-Method MDD approach by a reviewer's team and by the OOmCFP FSM procedure.

The objective of this study aims to evaluate the usefulness of the OOmCFP FSM procedure to detect defects by comparing these defects and defect types with the defects and defect types found by the reviewers, and by determining the quality characteristics that the models do not achieve due to these defects. Thus, we have formulated the following research questions:

RQ1: Are the defect types found by the inspection team the same as the defect types found by the OOmCFP FSM Procedure?

RQ2: Are the quality characteristics related to the defects found by the inspection team the same as the quality characteristics related to the defects found by the OOmCFP tool?

RQ3: Is the OOmCFP FSM procedure efficient at finding defects related to a defect type?

RQ4: Is the OOmCFP FSM procedure useful at finding defects in models of an MDD environment?

9.1.1 Case and Subjects Selection

The case of the study is a software development project of a Management Information System that has been developed in an MDD environment. To ensure that the case study is performed in a real-world context, we selected a software project from an industrial partner that is using a tool that implements the OO-Method approach.

We are aware that it is possible to define different model versions for the intended software system depending on the vision of different analysts. Consequently, in order to find defects that represent real-world defects introduced by analysts in their daily work, we selected a software project

that has several model versions that have been developed by different analysts. Therefore, the defects that the models may have are not biased by the researchers. In other words, we select a software project that has several versions of design models, which do not correspond to consecutive models nor even correspond to models of evolving software.

Considering these selection criteria, we selected the Photography Agency project. This project was modeled by several groups of analysts in order to find the best way to represent a solution to the problem. Then, final applications were automatically generated from the models in the OO-Method MDD environment. Therefore, the Photography Agency can choose the system that best fits its organization. From this project, we selected five versions of the conceptual model, each of which was developed by groups of three different novice analysts. Therefore, the defects in these models were not introduced on purpose. Nevertheless, since these five versions of the conceptual model correspond to the same project, they are similar among them. For ethical considerations, the names of the analysts are kept confidential to avoid repercussions within the enterprise. Also, as a consent agreement, the versions of the conceptual model that were selected for the case study do not correspond to the final versions of the model used to generate the Photography Agency application. In order to understand the defects found, a brief description of the operation of the Photography Agency is presented next:

The photography agency is dedicated to the management of photo reports and their distribution to publishing houses. This agency operates with freelance photographers, who must present a request to the production department of the photography agency. This request contains: the photographer's personal information, a description about the equipment owned, a brief curriculum vitae, and a book showing the photographic projects performed by the photographer. An accepted photographer is

classified in one of three possible levels for which minimum photography equipment is required. For this, the technical department creates a new record for the photographer and saves it in the photographer's file. A new record with a sequential code is created for each photo project presented by a photographer. This record has the price that the publishing houses must pay to the agency, which is established according to the number of photos and the level of the photographer. This record also contains a descriptive annotation about the content of the project. Depending on the level of photographer, the sales department establishes the price that will be paid to the photographer and the price that will be charged to the publishing house for each photo.

The subjects of the study correspond to people with knowledge of the OO-Method MDD approach and the OO-Method modeling tool having different levels of expertise: expert, intermediate, and novice. Expert analysts can correctly model and generate the final software system using the OO-Method approach; Intermediate analysts can produce a complete model, but the model is not correct; and Novice analysts cannot produce a complete model of a system, and, hence, they cannot generate the final application. The subjects were selected from the PROS Research Center. This group of subjects was made up of 16 researchers with different levels of expertise on the OO-Method approach and its modeling tool: 5 were considered as experts, 5 as intermediates, and 6 as novices. In order to maintain the confidentiality of the subjects who participated in the study, we assigned the code E1, E2, E3, E4, and E5 to the experts; I1, I2, I3, I4, and I5 to the intermediates; and N1, N2, N3, N4, N5, and N6 to the novices. This group of subjects does not have expertise using the OOmCFP measurement procedure. This is not necessary due to the OOmCFP procedure is applied by the researchers using the OOmCFP tool. Regarding the inspection technique,

subjects have some knowledge reading design models and requirements specifications.

9.1.2 Data Collection Procedure

The data collection procedure was defined taking into account the triangulation that will be used to analyze the results obtained in the study. Triangulation means taking different angles towards the studied object in order to provide a broader picture [Runeson and Host 2009]. In this study, two types of triangulation were considered: data triangulation and theory triangulation.

Data triangulation refers to using more than one data source or collecting the same data on different occasions. In this study, data triangulation is taken into account since there are five versions of the Photography Agency conceptual model that are reviewed.

Theory triangulation refers to using alternative theories in the study. This type of triangulation is taken into account since the models will be reviewed using two different techniques: analysis of the models performed by the selected subjects and analysis of the models performed by the tool that automates the application of the OOmCFP FSM procedure.

In summary, the following common steps were defined to collect data in the study for the five models:

- Each model was reviewed by an 8-person inspection team of analysts with different levels of expertise (three experts, three intermediates, and two novices) for 30 minutes. Work diaries with the defect founds were completed by each subject for each model reviewed.
- Each model was loaded into the OOmCFP tool by means of its XML representation generated by the tool that implements the OO-Method

approach. The OOmCFP tool delivered an Excel sheet with the defects found for each model, the corresponding defect types, and the time used in its analysis.

9.1.3 Analysis Procedure

Since the Photography Agency models versions have been developed by different analysts, the defects in the models are not known in advance. Thus, it is necessary to define a procedure to analyze the defects reported and distinguish between valid defects and invalid defects. Therefore, in the Photography Agency case study, a qualitative analysis procedure [Seaman 1999] was conducted in several steps. In the first step, the defects found were investigated in order to find the valid defects. Then, valid defects were classified into defect types. The defect types were coded using an editing approach (i.e., including a set of a priori codes that were extended and modified during the analysis). Each code was composed of the conceptual model where the defect was found and the conceptual construct involved in the defect. If a defect involves several views, the code is composed of all the views related to such defect. Afterwards, defects (valid or not) and defect types with the corresponding codes constituted a body of knowledge that was used to answer the research questions formulated in this study.

In addition, two independent variables and seven quantitative dependent variables were considered to investigate the research questions.

Independent Variables

1. Industrial models with their intrinsic complexity.
2. The techniques used to review the models:
 - a. A horizontal reading technique and a vertical reading technique [Travassos et al. 1999] that was used by the

reviewers in the models selected for the study. The horizontal reading technique refers to reading software artifacts that are built in the same software lifecycle phase. The vertical reading technique refers to reading software artifacts that are built in different software lifecycle phase. In our case, the software artifacts correspond to design models. Thus, the horizontal review correspond to the following: class models with respect to state transitions diagrams, class models with respect to functional models, and class models with respect to presentation models. The vertical review corresponds to class models with respect to textual requirements specification.

- b. An automated measuring system called OOmCFP that implements an ISO standard for Functional Size Measurement (i.e., COSMIC) as well as a FSM procedure defined to apply the standard to the conceptual model of an MDD approach. This automated measuring system also has a feature to find and to display defects to the users.

Dependent Variables

1. Number of defects found by the inspection teams in each model, which corresponds to the total amount of issues detected and classified as defects by the inspection teams.
2. Number of invalid defects found by the inspection teams in each model.

3. Number of defect types found by the inspection teams in each model, which corresponds to the classifiers that group similar defects according to the conceptual constructs.
4. Number of defects found by the OOmCFP tool in each model, which corresponds to the total amount of defects detected by the OOmCFP tool.
5. Number of defect types found by the OOmCFP tool in each model, which corresponds to the classifiers of the defects found.
6. Time used to find defects by the inspection teams in each model, with time measured in minutes.
7. Time used to find defects by the OOmCFP tool in each model, with time measured in minutes.

9.1.4 Validity Procedure

The validity of a study denotes the trustworthiness of the results, i.e., to what extent the results are true and not biased by the researchers' point of view [Runeson and Host 2009]. Even though the results obtained are valid for our study, we have identified some threats that might affect these results. There are three main aspects to describe the validity of a study: construct validity, internal validity, and external validity.

The following threats to the *construct validity* were identified in the case study:

- The defects found by people depend on the experience of each person involved with the MDD approach and the OO-Method modeling tool. Expert people look for defects in complex conceptual constructs in contrast to novice people who look for defects in the most frequently used and basic conceptual constructs. To mitigate this threat, we selected people with different levels of experience (expert, intermediate, and novice).

- The defects found may not correspond to a real defect in the model. To mitigate this risk, we investigated the defects found and the amount of invalid defects detected in the dependent variables a and b .

The following threats to the *internal validity* were identified in the case study:

- The experience of people with the OO-Method approach and its modeling tool affects the results of the study. People with experience can find real defects in the models quicker than people with lack of experience. These are two things that are measured and can be influenced by the experience of people, and not only by changing the independent variables. To mitigate this risk, we selected people with at least a basic knowledge of OO-Method and its modeling tool.
- The experience of people with the reading techniques. People with experience performing horizontal and vertical readings of OO-Method models can focus in common valid defect types. Thus, experience of people regarding reading techniques can influence the number of invalid defect types identified. Training the subjects regarding the reading techniques can help to diminish this threat.
- The models selected for the case study do not represent all the conceptual constructs of the OO-Method MDD approach since they do not have a presentation view. Thus, other conceptual constructs may have different defect types related.
- The model of a system represents only one way to model the system. To mitigate this risk, we selected alternative models of

the system performed by different groups of analysts in order to take into account different modeling possibilities.

- The learning effect of selected subjects during the inspections of models. Several persons work on several versions of design models of the same system. Thus, the first time that they analyze the models, they need to understand the selected case, the concepts involved, and how the model satisfy the requirements of the system. Thus, the time that they took to find defects is supposed to be longer in the first model analyzed. To mitigate this threat, we selected models that were developed by different groups of analysts. Thus, since different analysts take different decisions to better represent the system, persons need to understand how the model satisfy the system's requirements for each model.

Finally, the following threats to the *external validity* were identified in the case study:

- The representativeness of the selected case study models since all the models correspond to a specific MDD approach. This can cause the findings to be valid only in this industrial context. Repeating the case study with other MDD approaches can give more information about the generalizations of the results.
- The representativeness of the selected subject population since wrong people can be involved in the case study. To mitigate this threat, we have recruiting with different expertise levels with the MDD approach. Therefore, results can be generalizable to other people.
- The representativeness of the inspection technique. We choose the inspection techniques that better fit with the goal of the case

study. Repeating the case study with other inspection techniques would obtain different results.

9.2 Results

This section presents the execution of the case study, and the analysis and interpretation of the collected data.

At this point, it is important to mention that the MDD modeling tool already has detected some defects according to the OO-Method metamodel. These defects are related to the structural relationships among the conceptual constructs of the OO-Method metamodel, for instance, it is not possible to specify a precondition of an attribute due to the preconditions can only be related to services in the OO-Method metamodel.

Since the OO-Method approach has a well-defined metamodel, the tool that implements OO-Method prevents analysts from performing some actions that infringe the structural properties of the metamodel when they are designing a model. Therefore, the modeling tool alleviates the difficult task of detecting defects in the models. However, defects related to the semantics that represent the conceptual constructs used in the models, or defects related to the values that are assigned or not to the conceptual constructs used in the models are not addressed by the MDD tool. Precisely, these defects are found by the inspectors and the OOmCFP tool, which correspond to defects that are not detected by the OO-Method modeling tool.

Therefore, if the conceptual models have some defects related to structural relationships, the MDD tool will detect them before the generation of the final application. Nevertheless, if the conceptual models have defects related to semantics or syntactical correctness, the MDD tool will generate a

final application that has faults. To prevent these faults, inspections and OOmCFP are used to detect other kind of defects.

9.2.1 Execution Description

The five selected models were inspected by both the inspection team using reading techniques and the tool that implements the OOmCFP FSM procedure.

For the reading technique, each model was inspected by a group of eight subjects (three experts, three intermediates, and two novices). Table 38 shows how the 16 subjects were distributed across the 8-person inspection teams for each of the five models.

Table 38. Distribution of subjects in the inspection teams for the five models.

Models	Subjects
Model1	E1, E3, E4, I1, I3, I4, N1, N3
Model2	E2, E3, E4, I1, I4, I5, N2, N3
Model3	E2, E3, E4, I1, I2, I3, N2, N5
Model4	E2, E3, E4, I1, I2, I3, N2, N6
Model5	E2, E4, E5, I2, I3, I4, N3, N4

To perform the inspections, the team was located in a room that had one computer for each inspector. Each inspection team received the models they were assigned to find defects for. Also, each person received a set of instructions to perform inspections and a template that had to be completed during the inspection with the defects found for each inspected model and the time that had passed.

For the tool that implements the OOmCFP measurement procedure, each model was loaded in the tool, which delivers an excel sheet with the defects

that have been detected in each model, the related defect types, and the time used to find these defects.

9.2.2 Analysis and Interpretation Issues

In the first step, the defects detected by the inspection team in the five models were analyzed. To do this, we put all the defects found by the inspection team for each model in an Excel sheet.

If a defect was detected by more than one person on the inspection team in each model, it was considered only once for the analysis since we are interested in the defects found and not in the skills of the inspectors. For instance, in Model2, the defect ‘service edit_instance of class Photographer does not have a valuation⁴’ was found by E3, E4, I4, N2, N3. Figure 9.1 presents the number of defects detected by each person on the inspection team in Model2 (for instance, inspector E2 found 5 defects), which in total corresponds to 50 defects. In Figure 9.1, it is clear that some defects were detected by more than one inspector. For instance, inspectors M4 and M5 found the same identical defect, so we counted it only once for our dependent variable *a: number of defects found by the inspection teams in each model*. Therefore, there are only 36 distinct defects found by the inspection team in Model2.

Figure 9.1 shows that novice inspectors found more defects than expert inspectors; i.e., novice inspectors (N2 and N3) detected a total of 24 defects, and expert inspectors (E2, E3, and E4) detected a total of 17 defects. This occurs because novice inspectors identified several invalid defects (i.e., 10 defects) in contrast to expert inspectors. The reason for this is that novice inspectors do not have enough knowledge of the modeling approach, which

⁴ Valuations are formulae that are used to assign values to the attributes of class using a formal language called OASIS.

may impede them to properly identify when a modeling specification is incorrect or missing. Once the defects were reorganized to consider them only once for each model, it can be observed that N3 detected 13 different defects compared to the other inspectors. However, 10 of these defects were misinterpretations of the novice inspector N3.

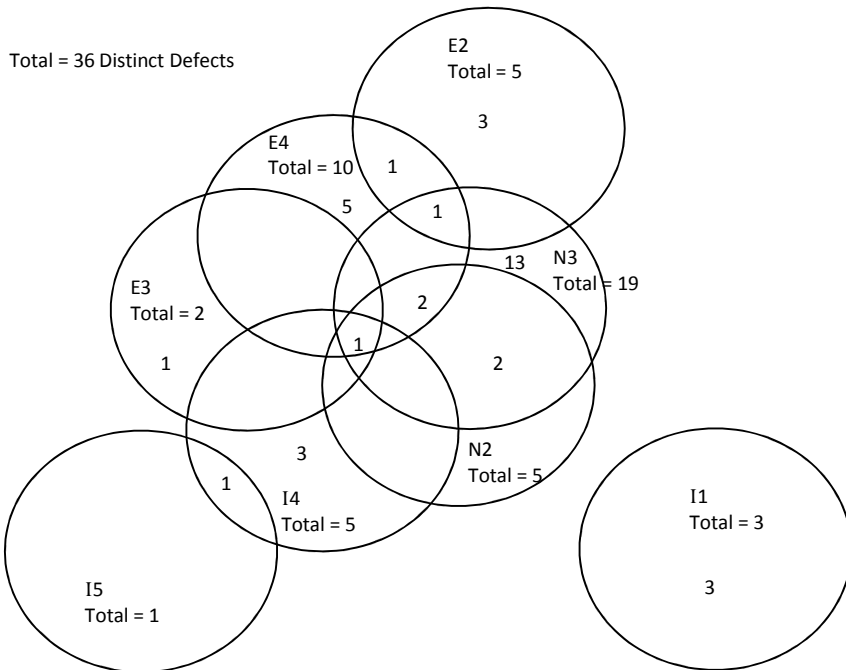


Figure 9.1 Defects found by each inspector in Model2

For this reason, we analyzed the different defects found in each model in order to leave out the issues identified by the inspection team that are invalid defects that arose due to misinterpretations by the inspection team. For instance, in Model4, the identified issue: ‘the derivation formula of the attribute total of class DeliveryNoteEx is missing’ is not a defect since this formula is actually specified for this attribute. Figure 9.2 presents the defects

found by the inspection team in Model4, which is up to 52 defects. However, there are 13 defects that do not really correspond to a defect. Thus, Table 34 presents the real defects of Model4 (39 defects) and the issues that do not correspond to a defect (13 invalid defects). In order to illustrate the defects that were found by the inspectors in Model4, we listed the defects found with D1, D2, D3, etc.

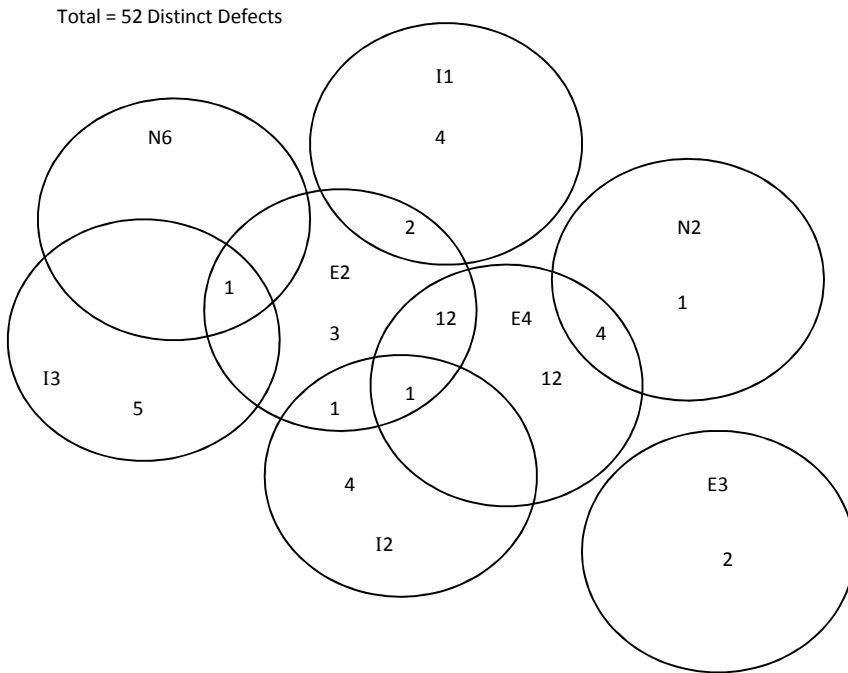


Figure 9.2 Distinct defects found in Model4

Many times novice and intermediate inspectors identified invalid defects; however, Table 39 shows that even the expert inspectors identified invalid defects (for instance E3). In this case, the expert inspector E3 interpreted that to fulfill the requirements it was necessary to use complex

derivation formulae for the price that the publishing houses must pay to the photography agency. Since this inspector did not find this specification in Model4, E3 detected defects D8 and D9. Nevertheless, these issues were not considered as defects by us since the specifications to fulfill the requirements were done with simple derivation formulae in Model4.

Table 39. Analysis of Defects found in Model4.

Subjects	Number of Defects	Valid Defects	Invalid Defects
E2, N6, I3	1	D1	
E2	2	D2, D7	D3
E2, E4, I2	1	D4	
E2, I2	1	D5	
E2, I1	1	D6	D42
E3	0		D8, D9
E4, E2	12	D10, D11, D12, D13, D14, D15, D16, D17, D18, D19, D20, D21	
E4	12	D22, D23, D24, D25, D26, D27, D28, D29, D30, D31, D32, D33	
E4, N2	4	D34, D35, D36, D37	
I1	0		D38, D39, D40, D41
I2	4	D43, D44, D45, D46	D43, D44, D45, D46
I3	1	D47	D48, D49, D50, D51
N2	0		D52
	Total	39 Valid Defects	13 Invalid Defects

Next, we analyzed each defect and assigned a code to classify the defect types found by the inspection team for later analysis. For instance, in Model1 the following defects corresponded to the same defect type:

DM_SReach: A state of the STD of a class that is not reachable: the state signed of the DeliveryNote class in its STD is not reachable (found by E4, I3, N1, N3); the state charged of the DeliveryNote class in its STD is not reachable (found by E4, I3); the state charged of the Exclusive class in its STD is not reachable (found by I3). Figure 9.3 shows the codes assigned to the defect types related to the defects found in Model1.

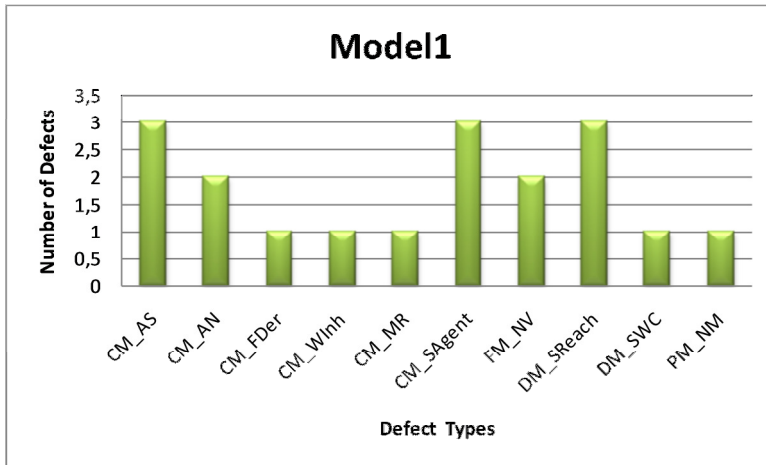


Figure 9.3 Defect Types found by the inspection team in Model1

Once the defect types were assigned, we noticed that expert inspectors were mainly focused on the specifications of the relationships in the models (such as agent relationships, dynamic relationships, inheritance relationships, etc). These are more complex constructs than the ones that novice inspectors were mainly focused on (such as attributes, valuations, etc). This situation occurred in the five models of our study, i.e., experts focused on the complex constructs leaving out the analysis of the basic constructs. Thus, we conclude that is not enough to have an inspection team made up of only expert inspectors, because, in a limited period of time, they only focused on the

complex constructs. For that reason, it is very important to have an inspection team made up of inspectors with different expertise in the MDD approach and its modeling tool.

In summary, 24 defect type codes were assigned to the defects found by the inspection teams in the five models of the study: 19 for the structural view, 1 for the functional view, 2 for the dynamic view, and 2 for the presentation view. These defects were related to the consistency among the views of the models (see Table 40), the syntactical correctness of each view of the models (see Table 40), and the completeness of the models regarding the requirements (see Table 41).

Table 40. Description of Defect types related to the correctness and consistency found by the inspection teams.

Defect Type	Description	Quality Characteristic
CM_RServ	Missing specification of dynamic services in a dynamic relationship.	Correctness
CM_SAgent	A service without a defined agent.	Correctness
CM_AAgent	An attribute without a defined agent.	Correctness
CM_RAgent	A role without a defined agent.	Correctness
FM_NV	A service without the specification of a valuation.	Consistency
DM_SReach	A state of the STD of a class that is not reachable	Correctness
DM_SWC	A state of the STD is reachable without the creation of an object.	Correctness
PM_NM	A model without the specification of a presentation view.	Consistency
PM_IP	Empty introduction pattern.	Correctness

Table 41. Description of Defect types related to the completeness found by the inspection teams.

Defect Type	Description	Quality Characteristic
CM_Id	A class with more than one identifier.	Completeness
CM_AS	Attribute of string data type with small size for the requirements.	Completeness
CM_AN	Attribute that allows a null value with a default value specified.	Completeness
CM_FDer	Wrong derivation formula for the requirements.	Completeness
CM_WArg	Wrong argument in a service.	Completeness
CM_MArg	Missing argument in a service.	Completeness
CM_DCServ	Duplicated creation event in a class.	Completeness
CM_SEditC	An edit service in a class that only has constant and derived attributes.	Completeness
CM_InhLib	Inheritance hierarchy between two classes without a liberator event.	Completeness
CM_InhId	An identifier defined in a child of an inheritance hierarchy.	Completeness
CM_WInh	Wrong definition of an inheritance hierarchy between two classes for the requirements.	Completeness
CM_InhServ	Duplicated service in the children of an inheritance hierarchy.	Completeness
CM_MR	Missing relationship in the class model.	Completeness
CM_WCard	Wrong cardinality in a relationship.	Completeness
CM_FIC	Wrong integrity constraint formula.	Completeness

Figure 9.4 shows the defect types found by the inspection teams in the five models of the study. This figure shows that Model5 had 7 defects related to the CM_SEditC defect type, Model2 had only 1 defect related to

the same defect type, and the remainder models did not have defects related to this defect type.

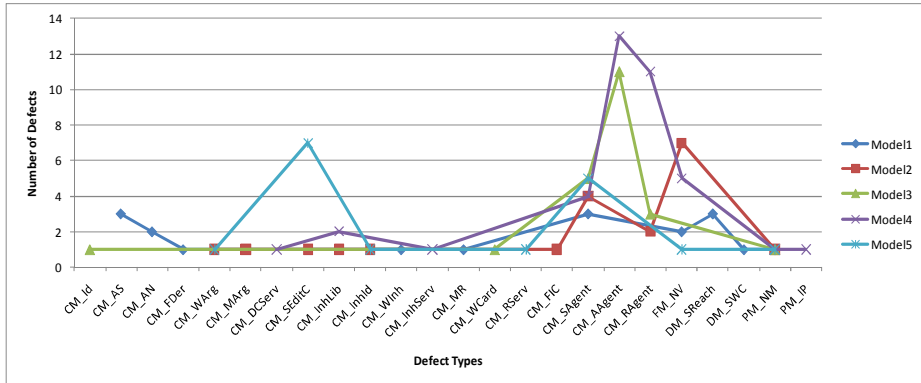


Figure 9.4 Defect Types found by the inspection teams

In contrast, the defects found by the OOmCFP tool were detected only once, that is, the tool does not have misinterpretations in the identification of defects since it applies specific rules that analyze the entire functionality of a system that is represented in the model in a systematic way. Also, for each defect, the OOmCFP tool presents the corresponding defect type.

It is important to mention that the OO-Method model compiler detects some defects when it transforms the conceptual model to an execution model and then generates the corresponding final application. These defects are related to the structure of the XML representation of the conceptual model. Nevertheless, the OO-Method model compiler does not analyze the functionality specified with the conceptual constructs. OOmCFP analyzes the specification of the functionality of final applications in the conceptual constructs instead of its XML representation. Hence, the defects found by the OOmCFP are not found by the model compiler. Therefore, it is very important to rely on a technique that can find defects that the model compiler does not detect.

Even though the OOmCFP tool can detect defects related to 24 defect types in the OO-Method conceptual models, the defects found in the five models of the study were related to only three defect types (1 for the structural view, 1 for the functional view, and 1 for the presentation view). This occurs because several defect types detected by the OOmCFP tool are related to the presentation view of a model, and the selected models do not have this view specified. Table 42 shows the description of the 3 defect types found and the related quality characteristics.

Table 42. Description of Defect types found by the OOmCFP tool.

Defect Type	Description	Quality Characteristic
CM_NA	A class without the definition of one or more attributes.	Correctness
FM_NV	A service without the specification of a valuation.	Consistency
PM_NM	A model without the specification of a presentation view.	Consistency

The difference between the defect types found by the inspection teams and the OOmCFP tool is not surprising due to the following:

- Some defect types found by the inspection teams are related to the semantics of the model, i.e., CM_Id, CM_AS, CM_AN, CM_FDer, CM_WArg, CM_MArg, CM_DCServ, CM_SEditC, CM_InhLib, CM_InhId, CM_WInh, CM_InhServ, CM_MR, CM_WCard, and CM_FIC. These 15 defect types are detected by the inspection teams since they have the requirements specification of the Photography Agency system. Since the OOmCFP tool applies the OOmCFP FSM procedure to analyze the functionality of design models by a systematic analysis of

the models, there is no knowledge process that the tool can perform in order to analyze the semantics of a model for the requirements.

- Another defect type found by the inspection teams is related to the correctness of conceptual constructs that do not contribute to the functionality of the applications, i.e., PM_IP. This is not surprising since the OOmCFP FSM procedure takes into account the conceptual constructs that contribute to the functionality of the applications leaving out aesthetic aspects that do not represent data movements.
- Other defect types found by the inspection teams are related to the relationships between classes in the structural view of the model (i.e., CM_RServ, CM_SAgent, CM_AAgent, and CM_RAgent) and the transitions among states in the dynamic view of the model (i.e., DM_SReach, and DM_SWC). One limitation of the OOmCFP FSM procedure is that it does not analyze the relationships and the cardinalities of the conceptual constructs because they do not represent data movements.
- The remaining defects (FM_NV, PM_NM) correspond to the defect types found by the tool. In addition, the OOmCFP tool identified a defect type (i.e., CM_NA) that was not identified by the inspection teams.

Figure 9.5 shows the defect types found by the OOmCFP tool in the models. This figure shows that the five models had one defect related to the defect type PM_NM, Model4 had four defects related to defect type CM_NA, Model2 had eight defects related to defect type FM_NM, etc.

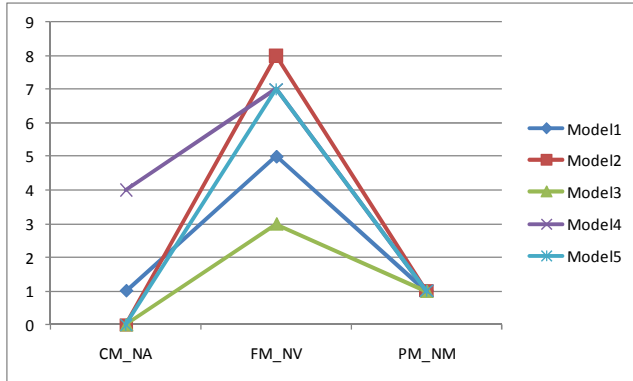


Figure 9.5 Defect Types found by the OOmCFP tool.

Table 43 presents the number of defects, defect types, and times collected for the dependent variables from the inspections performed by the inspection teams and the OOmCFP tool. Based on the dependent variables, we try to provide an answer to the underlying research questions.

Table 43. Defects found by the inspection teams (IT) and OOmCFP.

Models	Number of defects by IT	Number of invalid defects by IT	Number of valid defects by IT	Number of defect types by IT	Number of defects by OOmCFP	Number of defect types by OOmCFP	Time IT	Time OOmCFP
Model1	36	18	18	10	7	3	229	0,31
Model2	36	16	20	10	9	2	233	0,43
Model3	39	17	22	7	4	2	198	0,95
Model4	52	13	39	9	12	3	214	0,76
Model5	38	21	17	7	8	2	249	0,38

RQ1: Are the defect types found by the inspection team the same as the defect types found by the OOmCFP FSM Procedure?

The inspection team found defects related to 24 defect types in the five models of the study (see Table 35). These defect types are related to the four views (structural view, functional view, dynamic view, and presentation view) of the OO-Method conceptual model. In contrast, the OOmCFP tool found defects related to only 3 defect types in the five models (see Table 36), which are related to the structural view, the functional view, and the presentation view of the OO-Method conceptual model. This does not mean that the tool cannot find defects related to the dynamic view of a model; however, the tool does not identify defects related to the transitions between the states of the state-transition diagram. Two of the defect types found by OOmCFP were also found by the inspection team (i.e., FM_NV and PM_NM). However, it is important to note that defect type CM_NA was not detected by the inspection team.

Thus, it can be observed that some of the defect types found by the inspection team are the same as the defect types found by the OOmCFP tool; also, there are defect types found by the inspection team that were not found by the tool, and vice versa.

RQ2: Are the quality characteristics related to the defects found by the inspection team the same as the quality characteristics related to the defects found by the OOmCFP tool?

Focusing on the defect types found by the inspection teams in the five models, the great majority of defect types are related to the semantics of a model. Since the inspectors had the requirement specifications, they were able to inspect the models according to the requirements; thus, they detected defects related to the completeness of the models. The remaining defect types found by the inspectors were related to the consistency among the views of a model (e.g. FM_NV) and the syntactical correctness of a model (e.g. CM_SAgent).

In the same models, the defect types found by the OOmCFP tool were related to the consistency among the views of a model (e.g. FM_NV) and the syntactical correctness of the model (e.g. CM_NA). Understanding the completeness of a model as being a model that contains all the statements according to the requirements, this quality characteristic cannot be achieved by the OOmCFP tool since the tool does not have the requirements specification of the system.

In summary, the quality characteristics related to the defects found by the inspection team are completeness, consistency, and correctness; and the quality characteristics related to the defects found by the OOmCFP tool are consistency and correctness.

RQ3: Is the OOmCFP FSM procedure efficient at finding defects related to a defect type?

Focusing on the defect types found in Model5, the inspection team found 38 defects. However, 21 were misinterpretations of the team and did not correspond to real defects. Therefore, there were only 17 defects found in Model5 by the inspection team. These defects were related to 7 defect types. In the same model, 8 defects were found by the OOmCFP tool, which were related to 2 defect types (FM_NV and PM_NM). However, it is important to note that for defect type FM_NM, the inspection team found 1 defect and the OOmCFP tool found 7 defects. This occurs because the OOmCFP tool analyzes the model completely in a systematic way by applying the OOmCFP rules, while the people of the inspection team often forget to inspect some parts of the model. Also, the OOmCFP tool took 23 seconds (0.38 minutes) to completely analyze Model5, in contrast to the inspection team which took 4 1/4 hours (249 minutes) to partially analyze the same model.

In the remaining models evaluated (i.e., Model1, Model2, Model3, and Model4), the situation was very similar to Model5. Taking these results into

account, it is clear that OOmCFP is more efficient than an inspection team in finding defects related to a defect type.

RQ4: Is the OOmCFP FSM procedure useful at finding defects in models of an MDD environment?

Taking into account all the results obtained for the dependent variables, it is clear that the OOmCFP FSM procedure detected fewer defects than traditional inspections, although there are also advantages: it found the total number of defects related to a defect type; it found different defect types than the inspection team did (i.e., CM_NA); and it found defects using less resources (time, effort, etc). Thus, we consider that the OOmCFP FSM procedure is useful at finding defects in models of the OO-Method MDD approach.

9.3 Conclusions

In this chapter, we have reported a case study that was conducted to find out the usefulness of a FSM procedure to detect defects in the conceptual models of an MDD environment. The results indicate that the FSM is useful since it found all the defects related to a specific defect type and also found different defect types than an inspection team. However, the inspection team is also necessary to find defects that the FSM cannot find. Thus, the combination of these two techniques can be an interesting approach to evaluate the quality of conceptual models.

There are successful studies of the combination of FSM methods and inspections to find defects manually in textual representations of requirements [Trudel and Abran 2008; Trudel and Abran 2010]. However, to the best of our knowledge, this study corresponds to the first study of the usefulness of an FSM procedure to detect defects in conceptual models of

MDD environments. Thus, further empirical research is necessary to establish greater external validity for these results. Other researchers are invited to replicate our study in other MDD contexts.

One limitation of this study is that none of the analyzed models had a presentation view specified; still the OOmCFP finds defect types that are related to conceptual constructs of the presentation view.

Chapter 10

Conclusions

In this thesis, we have investigated the use of a standard Functional Size Measurement (FSM) method in the measurement of conceptual models of a Model-Driven Development (MDD) approach in order to achieve accurate functional size measurement results. To reach this goal, we have designed a FSM procedure based in the COSMIC FSM method that allows the accurate measurement of applications generated by the conceptual models of the OO-Method MDD approach. Furthermore, in this thesis, we have also investigated defect detection at early stages of the software development process, which in MDD approaches corresponds to the design of conceptual models. Thus, we have demonstrated the usefulness of the FSM procedure designed to detect defects in OO-Method conceptual models.

This last chapter introduces the conclusions of the work presented in this thesis. First, we list the main contributions of this thesis. Next, we present the publications that have been produced throughout the development of this work. Finally, we explain the work that is currently being performed as well as future work that has emerged from this thesis.

10.1 Contributions

The main contribution of this work is a functional size measurement procedure that allows the application of COSMIC to OO-Method conceptual models in order to obtain the accurate functional size of the applications generated in the OO-Method MDD environment. The development of this procedure comprises its design, its application, its automation, and also the verification of OO-Method conceptual models in order to be defect-free. Thus, this work provides the following contributions:

- 1 An *accurate FSM procedure* has been systematically designed to measure the functional size of applications generated in MDD environments from their conceptual models. We have designed this FSM procedure according to the last version (v. 3.0) of the standard COSMIC FSM method since it allows the measurement of multi-layer applications in contrast to other standard FSM methods. The design of the FSM procedure includes the following: a set of rules that allows the identification of the COSMIC concepts related to the measurement of the conceptual constructs of the OO-Method conceptual models; a set of rules that avoids the duplicated measurement of the conceptual constructs used in the conceptual models; and a set of rules that allows the measurement of the functional size of the generated applications taking into account different granularities: each functional process, each piece of software, or the entire application.
- 2 A *method to evaluate the precision of functional size measurement results* has been defined. This method is based on the ISO 5725 standard, which evaluates precision according to the repeatability and the reproducibility of the measurement results.

- 3 *A tool that automates the application of the FSM procedure to MDD applications has been developed.* This tool measures the functional size of applications generated in the OO-Method MDD environment using the XML specification of the conceptual models involved. Thus, this tool can totally automatically obtain the functional size of these applications, and it can be easily generalized to other MDD applications.
- 4 *A metamodel with the minimal set conceptual constructs that an MDD approach must have* in order to allow the generation of fully working applications has been defined. This metamodel has been designed based on the conceptual constructs of the OO-Method approach using the MOF metamodeling standard [OMG 2006a] and the Eclipse open-source tools.
- 5 *A novel approach to detect defects in object-oriented conceptual models* has been defined. Since a FSM procedure analyzes all the conceptual constructs of a conceptual model and their relationships, we apply the FSM procedure to conceptual models that have defects in order to find these defects. Therefore, the FSM procedure does not allow the measurement of the functional size, but it obtains the defects of the model that are related to 24 defect types that impede the correct compilation of the conceptual models in order to generate final applications in MDD environments.
- 6 *A tool that automates the defect detection in conceptual models of MDD applications* has been developed. This tool detects defects in the OO-Method conceptual models using their XML specification as a starting point and returns a report with the conceptual constructs related to each defect found in the conceptual model.

10.2 Publications

Research in different areas of software engineering has been carried out to perform the work presented in this thesis, such as conceptual modeling methods, quality measures for conceptual models, standard measurement methods, standard validation methods, software quality, software testing, functional size methods, and functional size measurement tools. Thus, the research activity related to this work has resulted in **25 publications**, which correspond to 4 international journals, 2 national journals, 13 international conferences, 4 international workshops, and 2 national conferences.

International Journals:

- Giovanni Giachetti, **Beatriz Marín**, Oscar Pastor. *Integration of Domain-Specific Modeling Languages and UML through UML Profile Extension Mechanism*. International Journal of Computer Science & Applications (**IJCSA 2009**) - vol. 6, n° 5, pp. 145-174, 2009.
- **Beatriz Marín**, Oscar Pastor, Alain Abran. *Towards an accurate functional size measurement procedure for conceptual models in an MDA environment*. Data & Knowledge Engineering (**DKE 2010**) -. vol. 69, n° 5, pp. 472-490. 2010. (DKE is included in the top quartile of JCR: 1.745 impact factor 2010, position 23/116 in the "Computer Science, Information System" category).
- **Beatriz Marín**, Giovanni Giachetti, Oscar Pastor, Alain Abran. *A Quality Model for Conceptual Models of MDD Environments*. Advances in Software Engineering - Special Issue: New Generation of Software Metrics (**ASE 2010**). Article ID 307391, 17 pages, 2010.

- Giovanni Giachetti, Manoli Albert, **Beatriz Marín**, Oscar Pastor. *Linking UML and MDD Through UML Profiles: A Practical Approach based on the UML Association*. Journal of Universal Computer Science (**JUCS 2010**) - vol. 16, no. 17, PP. 2353-2373, 2010. (JUCS is included in the third quartile of JCR: 0.669 impact factor 2010, position 68/93 in the "Computer Science, Software Engineering" category).

National Journals:

- **Beatriz Marín**, Nelly Condori-Fernández, Oscar Pastor. *Calidad en Modelos Conceptuales: Un Análisis Multidimensional de Modelos Cuantitativos basados en la ISO 9126*. Revista de Procesos y Métricas de las Tecnologías de la Información (**RPM**), vol. 4, pp. 153-167, 2007.
- Antonio de Rojas, Tanja Vos, **Beatriz Marín**. *Experiencias de una PYME en la mejora de procesos de prueba*. Revista Española de Innovación, Calidad e Ingeniería del Software (**REICIS**), vol. 5 n° 2, pp. 63-69, 2009.

International Conferences:

- Giovanni Giachetti, **Beatriz Marín**, Nelly Condori-Fernández, Juan Carlos Molina. *Updating OO-Method Function Points*. 6th IEEE International Conference on the Quality of Information and Communications Technology (**QUATIC 2007**). Lisbon, Portugal, September 12-14, 2007. IEEE Computer Society Press, pp. 55-64.
- **Beatriz Marín**, Giovanni Giachetti, Oscar Pastor. *Una Herramienta Industrial para la Medición del Tamaño Funcional de Aplicaciones Desarrolladas en Entornos MDA*. XI Conferencia Iberoamericana de Software Engineering (**CIBSE 2008**). Recife, Brazil, February 13-17, 2008. pp. 357-362.

- **Beatriz Marín**, Oscar Pastor, Giovanni Giachetti. *Automating the Measurement of Functional Size of Conceptual Models in a MDA Environment*. 9th International Conference on Product Focused Software Process Improvement (**PROFES 2008**). Rome, Italy, June 23-25, 2008. Springer, LNCS 5089, pp. 215-229.
- **Beatriz Marín**, Nelly Condori-Fernández, Oscar Pastor. *Towards a Method for Evaluating the Precision of Software Measures*. 8th International Conference on Quality Software (**QSIC 2008**). Oxford, UK, August 12-13, 2008. IEEE Computer Society Press, pp. 305-310.
- **Beatriz Marín**, Giovanni Giachetti, Oscar Pastor. *Measurement of Functional Size in Conceptual Models: A Survey of Measurement Procedures based on COSMIC*. 3rd International Conference on Software Process and Product Measurement (**MENSURA 2008**). Munich, Germany, November 17-19, 2008. Springer, LNCS 5338, pp. 170-183.
- **Beatriz Marín**, Giovanni Giachetti, Oscar Pastor, Alain Abran. *Identificación de Defectos en Modelos Conceptuales utilizados en Entornos MDA*. XII Conferencia Iberoamericana de Software Engineering (**CIbSE 2009**), Medellín, Colombia, 2009. pp. 109-114.
- Giovanni Giachetti, **Beatriz Marín**, Oscar Pastor. *Integración de UML y DSMLs en Entornos de Desarrollo Dirigido por Modelos*. XII Conferencia Iberoamericana de Software Engineering (**CIbSE 2009**). Medellín, Colombia, 2009. pp. 103-108.
- Giovanni Giachetti, **Beatriz Marín**, Oscar Pastor. *Using UML Profiles to Interchange DSML and UML Models*. 3rd International Conference on Research Challenges in Information Science (**RCIS 2009**). Fès, Morocco, 22-24 April 2009. IEEE Computer Society, pp. 385-394.

- Giovanni Giachetti, **Beatriz Marín**, Oscar Pastor. Using UML as a Domain-Specific Modeling Language: A Proposal for Automatic Generation of UML Profiles. 21st International Conference of Advanced Information Systems Engineering (**CAiSE 2009**). Amsterdam, The Netherlands, June 8-12, 2009. Springer, LNCS 5565, pp. 110-124.
- **Beatriz Marín**, Giovanni Giachetti, Oscar Pastor. *Applying a Functional Size Measurement Procedure for Defect Detection in MDD Environments*. 16th European Conference on Software Process Improvement and Innovation (**EUROSPI 2009**), Alcalá (Madrid), Spain, 2009. Springer-Verlag, CCIS 42, pp. 57-68.
- **Beatriz Marín**, Giovanni Giachetti, Oscar Pastor, Tanja Vos, Alain Abran. *Evaluating the Usefulness of a Functional Size Measurement Procedure to Detect Defects in MDD Models*. 4th International Symposium on Empirical Software Engineering and Measurement (**ESEM 2010**), Bolzano-Bozen, Italy, 2010. ACM.
- **Beatriz Marín**, Giovanni Giachetti, Oscar Pastor, Tanja Vos. *A Tool for Automatic Defect Detection in Models used in Model-Driven Engineering*. 7th International Conference on the Quality of Information and Communications Technology (**QUATIC 2010**), Oporto, Portugal, 2010. IEEE, pp. 242-247.
- **Beatriz Marín**, Tanja Vos, Giovanni Giachetti, Arthur Baars, Paolo Tonella. *Towards Testing Future Web Applications*. 5th International Conference on Research Challenges in Information Science (**RCIS 2011**). Guadeloupe, France, 19-21 May 2011. IEEE Computer Society, pp. 226-237.

International Workshops:

- **Beatriz Marín**, Giovanni Giachetti, Oscar Pastor. *Intercambio de Modelos UML y OO-Method*. X Workshop Iberoamericano de

Ingeniería de Requisitos y Ambientes Software (**IDEAS 2007**). Isla Margarita, Venezuela, Mayo 07-11, 2007. pp. 283–296.

- **Beatriz Marín**, Nelly Condori-Fernández, Oscar Pastor, Alain Abran. *Measuring the Functional Size of Conceptual Models in an MDA Environment*. 20th International Conference on Advanced Information Systems Engineering Forum (**CAiSE Forum 2008**). Montpellier, France, June 18-20, 2008. pp. 33-36.
- **Beatriz Marín**, Nelly Condori-Fernández, Oscar Pastor. *Design of a Functional Size Measurement Procedure for a Model-Driven Software Development Method*. Accepted in the 3rd Workshop on Quality in Modeling (**MODELS Workshops 2008**). Toulouse, France, September 28-30, 2008.
- Fernanda Alencar, **Beatriz Marín**, Giovanni Giachetti, Oscar Pastor, Jaelson Castro, Xavier Franch, Joao Pimentel. *From i* to OO-Method: Problems and Solutions*. 4th International i* Workshop (**iStar 2010**) - CAiSE Workshops, vol. 586. CEUR Workshop Proceedings. 2010.

National Conferences:

- Giovanni Giachetti, **Beatriz Marín**, Oscar Pastor. *Perfiles UML y Desarrollo Dirigido por Modelos: Desafíos y Soluciones para Utilizar UML como Lenguaje de Modelado Específico de Dominio*. V Taller sobre Desarrollo de Software Dirigido por Modelos (**DSDM 2008**) – Taller JISDB. 2008.
- Tanja Vos, Arthur Baars, **Beatriz Marín**. *Pruebas Evolutivas en la Industria*. V Taller sobre Pruebas en Ingeniería del Software (**PRIS 2010**) – Taller de las Jornadas de Ingeniería del Software y Bases de Datos (JISBD), Vol. 4, n° 5, 2010. pp. 59-66.

Table 44 summarizes and classifies these publications according to the place of publication.

Table 44. Summary of publications related to this thesis.

Category	Number	Acronyms
International Journal indexed by JCR	2	DKE, JUCS
International Journal	2	IJCSA, ASE
National Journal	2	RPM, REICIS
International Conferences	13	QUATIC (2), CibSE(3), QSIC, PROFES, MENSURA, RCIS (2), CAiSE, EUROSPI, ESEM
International Workshops	4	IDEAS, CAiSE Forum, MODELS Workshops, ISTAR
National Conferences	2	DSDM, PRIS
Total		25 publications

10.3 Future Work

The research presented here is not a closed work, and there are several interesting directions that can still be pursued in the field of functional size measurement and defect detection of MDD applications. The following list summarizes the research activities that are planned in order to continue this work:

- Application of the OOmCFP procedure to new approaches of requirements models, such as [España *et al.* 2009] [de la Vara *et al.* 2008], which has enough conceptual constructs to completely define conceptual models. By applying OOmCFP to these requirements approaches, accurate measurement results of the functional size of final applications can be provided in very early stages of a MDD software development process.
- Incorporation of more rules to detect defects in conceptual models of MDD approaches. In particular, we plan to incorporate rules that are related to the interaction models of MDD approaches focusing on new characteristics defined for models of this kind, such as [Aquino *et al.* 2010] [Panach *et al.* 2008]. Taking into account that there are several MDD approaches that do not have a defined interaction model defined, our quality model for defect detection can be used as a reference to improve these approaches.
- Definition of a generic model for functional size measurement, which can be used for other application domains of MDD approaches, such as pervasive models, automotive models, etc. To do this, we plan to define the model using lightweight extension mechanisms, and also to develop an open-source tool that allows the application of the model to different MDD approaches independently of the implementation platforms of these approaches.
- Definition of a model to predict the cost of MDD applications. Since MDD applications are generated automatically from conceptual models by means of a model compiler, the effort used to produce these applications is different than the effort used to produce applications by means of human programmers. Thus, current models to predict the cost of software applications (e.g. COCOMO) should

not be used to predict the cost of MDD applications because they assign a high value to the human effort that is used to produce the software. To define a cost prediction model, we plan to develop a repository with the measures obtained in several MDD projects and then calculate the cost by taking into account the effort required to produce the model compiler and also the effort required to develop the conceptual models.

- Definition of techniques to perform Model-Driven Testing. We are currently working on testing the Future Internet using search-based software testing. We expect that Future Internet applications will be developed using Model-Driven methods. Thus, with the knowledge obtained in this work about Model-Driven Methods and our knowledge related to software testing, we plan to develop a methodology that allows conceptual models to be tested using search-based testing techniques. Then, test cases can be automatically generated.

10. Conclusions

References

Abrahão, S., Mendes, E., Gomez, J. AND Insfrán, E. 2007. *A Model-Driven Measurement Procedure for Sizing Web Applications: Design, Automation and Validation*. In Proceedings of the ACM/IEEE 10th International Conference On Model Driven Engineering Languages and Systems (MoDELS), Nashville, USA2007, 467-481.

Abrahao, S., Poels, G. AND Pastor, O. 2004. *Assessing the Reproducibility and Accuracy of Functional Size Measurement Methods through Experimentation*. In Proceedings of the International Symposium on Empirical Software Engineering (ISESE)2004 IEEE Computer Society, 189-198.

Abrahão, S., Poels, G. AND Pastor, O. 2006. *A Functional Size Measurement Method for Object-Oriented Conceptual Schemas: Design and Evaluation Issues*. Journal of Software and System Modeling 5(1), 48-71.

Abran, A. 2010. *Software Metrics & Software Metrology*. Wiley-IEEE Computer Society Press.

Abran, A., Desharnais, J., Lesterhuis, A., Londeix, B., Meli, R., Morris, P., Oligny, S., O'Neil, M., Rollo, T., Rule, G., Santillo, L., Symons, C. AND Toivonen, H. 2007. *The COSMIC Functional Size Measurement Method - Version 3.0*. GELOG web site www.gelog.etsmtl.ca.

Abran, A., Desharnais, J., Oligny, S., St-Pierre, D. AND Symons, C. 1999. *COSMIC-FFP Measurement Manual - Version 2.0*.

Abran, A., Desharnais, J., Oligny, S., St-Pierre, D. AND Symons, C. 2001. *COSMIC-FFP Measurement Manual - Version 2.1*. <http://www.cosmicon.com/>.

Abran, A., Desharnais, J.M., Oligny, S., St-Pierre, D. AND Symons, C. 2003. *COSMIC-FFP Measurement Manual - Version 2.2, The COSMIC Implementation Guide for ISO/IEC 19761*. <http://www.cosmicon.com/>.

Abran, A. AND Sellami, A. 2002. *Initial Modeling of the Measurement Concepts in the ISO Vocabulary of Terms in Metrology*. In Proceedings of the 12th International Workshop on Software Measurement - IWSM, Magdeburg (Germany), Oct. 7-9 2002 Shaker-Verlag.

Albrecht, A. 1979. *Measuring Application Development Productivity*. In Proceedings of the IBM Applications Development Symposium 1979, 83-92

Aquino, N., Vanderdonckt, J. AND Pastor, O. 2010. *Transformation Templates: Adding Flexibility to Model-Driven Engineering of User Interfaces*. In Proceedings of the 25th ACM Symposium on Applied Computing, SAC 2010 Sierre, Switzerland 2010, S.Y. Shin, S. Ossowski, M. Schumacher, M.J. Palakal AND C.-C. Hung Eds. ACM Press, 1195–1202.

Azzouz, S. AND Abran, A. 2004. *A proposed measurement role in the Rational Unified Process (RUP) and its implementation with ISO 19761: COSMIC FFP*. In Proceedings of the Software Measurement European Forum (SMEF), Rome, Italy, January 28-30 2004, 1-12.

Basili, V.R. AND Rombach, H.D. 1988. *The TAME Project: Towards Improvement Oriented Software Environments*. IEEE Transactions on Software Engineering 14(6), 758-773.

Basili, V.R., S., G., Laitenberger, O., Lanubile, F., Shull, F., Sorumgard, S. AND Zelkowitz, M.V. 1996. *The Empirical Investigation of Perspective-Based Reading*. Empirical Software Engineering Journal I, 133-164.

Bellur, U. AND Vallieswaran, V. 2006. *On OO Design Consistency in Iterative Development*. In Proceedings of the 3rd Int. Conf. on Information Technology: New Generations (ITNG), April 10-12 2006 IEEE, 46-51.

Benbasat, I., Goldstein, D. AND Mead, M. 1987. *The case research strategy in studies of information systems*. MIS Q 11(3), 369-386.

Berenbach, B. 2004. *The Evaluation of Large, Complex UML Analysis and Design Models*. In Proceedings of the 26th ICSE, May 23-28 2004 IEEE Computer Society, 232-241.

Berkenkötter, K. 2008. *Reliable UML Models and Profiles*. Electronic Notes in Theoretical Computer Science 217, 203-220.

Bevo, V. 2005. *Analyse et Formalisation Ontologique des Procédures de Mesure Associées aux Méthodes de Mesure de la Taille Fonctionnelle des Logiciels: de Nouvelles Perspectives Pour la Mesure* Université du Québec à Montréal - UQAM, Montréal.

Bévo, V., Lévesque, G. AND Abran, A. 1999. *Application de la méthode FFP à partir d'une spécification selon la notation UML: compte rendu des premiers essais d'application et questions*. In Proceedings of the 9th International Workshop on Software Measurement (IWSM), Canada1999, 230-242.

Boehm, B. 1981. *Software Engineering Economics*. Prentice Hall.

CARE-Technologies 2011. Web site. <http://www.care-t.com/>. (Last accessed May 2011).

Condori-Fernández, N. 2007. *Un procedimiento de medición de tamaño funcional a partir de especificaciones de requisitos*. Doctoral thesis. In

Departamento de Sistemas Informáticos y Computación Universidad
Politécnica de Valencia, Valencia.

Condori-Fernández, N., Abrahão, S. AND Pastor, O. 2007. *On the Estimation of Software Functional Size from Requirements Specifications*. Journal of Computer Science and Technology 22(3), 358-370.

Condori-Fernández, N. AND Pastor, O. 2006. *Evaluating the Productivity and Reproducibility of a Measurement Procedure*. In Proceedings of the ER Workshops 2006, 352-361.

Condori-Fernández, N.A., S.; Pastor, O. 2004. *Towards a Functional Size Measure for Object-Oriented Systems from Requirements Specifications*. In Proceedings of the 4th IEEE International Conference on Quality Software (QSIC), Germany2004, 94-101.

Conradi, R., Mohagheghi, P., Arif, T., Hegde, L.C., Bunde, G.A. AND Pedersen, A. 2003. *Object-Oriented Reading Techniques for Inspection of UML Models – An Industrial Experiment*. In Proceedings of the 17th ECOOP, July 2003 2003 Springer, 483-501.

COSMIC_Group 2003. *Rice Cooker – Cosmic Group Case Study*.

Cherfi, S.S.-S., Akoka, J. AND Comyn-Wattiau, I. 2002. *Conceptual modeling quality—from EER to UML schemas evaluation*. In Proceedings of the 21st International Conference on Conceptual Modeling (ER 2002), Tampere, Finland2002, S.T.M. S. Spaccapietra, Y. Kambayashi Ed.

Davenport, T.H. AND Prusak, L. 1998. *Working Knowledge: How Organisations Manage What They Know*. Business School Press, Boston, Massachusetts.

de la Vara, J.L., Sánchez, J. AND Pastor, O. 2008. *Business Process Modelling and Purpose Analysis for Requirements Analysis of Information Systems*. In Proceedings of the CAiSE2008, Z. Bellahsene AND M. Léonard Eds. Springer, 213-227.

Dedene, G. AND Snoeck, M. 1994. *M.E.R.O.DE.: A Model-driven Entity-Relationship Object-oriented Development Method*. ACM SIGSOFT Software Engineering Notes 19(3), 51-61

DeMarco, T. 1982. *Controlling Software Projects*. Prentice Hall.

Diab, H., Frappier, M. AND St-Denis, R. 2001. *Formalizing COSMIC-FFP Using ROOM*. In Proceedings of the ACS/IEEE Int. Conf. on Computer Systems and Applications (AICCSA)2001, 312-318.

Diab, H., Koukane, F., Frappier, M. AND St-Denis, R. 2005. *μcROSE: Automated measurement of COSMIC-FFP for Rational Rose Real Time*. Information and Software Technology 47(3), 151-166.

Diaz, I., Sanchez, J. AND Pastor, O. 2005. *Metamorfosis: Un marco para el análisis de requisitos funcionales*. In Proceedings of the Workshop on Requirements Engineering (WER), Porto, Portugal2005, 233-244.

Eclipse 2011a. *Model Development Tools*. <http://www.eclipse.org/modeling/mdt/>. (Last accessed May 2011).

Eclipse 2011b. *Modeling Project*. <http://www.eclipse.org/modeling/>. (Last accessed May 2011).

Egyed, A. 2006. *Instant Consistency Checking for the UML*. In Proceedings of the 28th ICSE, Shangai, China, May 20-28 2006 ACM, 381-390.

España, S., González, A. AND Pastor, O. 2009. *Communication Analysis: A Requirements Engineering Method for Information Systems*. In Proceedings of the 21st International Conference on Advanced Information Systems Engineering (CAiSE 2009), Amsterdam, The Netherlands2009, P. Van Eck, J. Gordijn AND R. Wieringa Eds. Springer, 530-545.

Fenton, N. AND Pfleeger, S. 1996. *Software Metrics: A Rigorous and Practical Approach (2nd edition)*. International Thomson Computer Press.

Fenton, N.E. AND Neil, M. 1999. *A Critique of Software Defect Prediction Models*. IEEE Transactions on Software Engineering 25(5), 675-689.

Fink, T., Koch, M. AND Pauls, K. 2006. *An MDA approach to Access Control Specifications Using MOF and UML Profiles*. Electronic Notes in Theoretical Computer Science 142, 161-179.

France, R.B., Ghosh, S., Dinh-Trong, T. AND Solberg, A. 2006. *Model-driven development using uml 2.0: Promises and pitfalls*. IEEE Computer 39(2), 59-66.

García, F., Bertoa, M.F., Calero, C., Vallecillo, A., Ruíz, F., Piattini, M. AND Genero, M. 2006. *Towards a consistent terminology for software measurement*. Information and Software Technology 48(8), 631-644.

Gartner 2011. *Gartner Says Worldwide IT Spending to Grow 5.1 Percent in 2011*. <http://www.gartner.com/it/page.jsp?id=1513614>. (Last accessed May 2011).

Genero, M., Piattini, M. AND Calero, C. 2005. *A Survey of Metrics for UML Class Diagrams*. Journal of Object Technology 4(9), 59-92.

Giachetti, G., Marín, B., Condori-Fernández, N. AND Molina, J.C. 2007. *Updating OO-Method Function Points*. In Proceedings of the 6th IEEE International Conference on the Quality of Information and Communications Technology (QUATIC 2007), Lisboa, Portugal2007, 55-64.

Gomaa, H. 2000. *Designing Concurrent, Distributed, and Real-Time Applications with UML*. Addison-Wesley.

Gomaa, H. AND Wijesekera, D. 2003. *Consistency in Multiple-View UML Models: A Case Study*. In Proceedings of the Workshop on Consistency Problems in UML-based Software Development II, San Francisco, USA, October 20 2003 IEEE, 1-8.

Gómez, J., Insfrán, E., Pelechano, V. AND Pastor, O. 1998. *The Execution Model: a component-based architecture to generate software components from conceptual models*. In Proceedings of the Workshop on Component-based Information Systems Engineering 1998.

Grau, G. AND Franch, X. 2007a. *ReeF: Defining a Customizable Reengineering Framework*. In Proceedings of the CAiSE Trondheim 2007a Springer, 485-500.

Grau, G. AND Franch, X. 2007b. *Using the PRiM method to Evaluate Requirements Model with COSMIC-FFP*. In Proceedings of the International Conference on Software Process and Product Measurement (IWSM-MENSURA), Mallorca, Spain, November 2007b, 110-120.

Habela, P., Glowacki, E., Serafinski, T. AND Subieta, K. 2005. *Adapting Use Case Model for COSMIC-FFP Based Measurement*. In Proceedings of the 15th International Workshop on Software Measurement (IWSM), Montréal 2005, 195-207.

Habra, N., Abran, A., Lopez, M. AND Sellami, A. 2008. *A framework for the design and verification of software measurement methods*. Journal of Systems and Software 81(5), 633-648.

Hailpern, B. AND Tarr, P. 2006. *Model-driven development: The good, the bad, and the ugly*. IBM Systems Journal 45(3), 451-461.

IEEE 1990. *IEEE 610 Standard Computer Dictionary. A Compilation of IEEE Standard Computer Glossaries*.

IEEE 2004. *IEEE 1012 Standard for Software Verification and Validation*.

IEEE 2009. *IEEE 1044 Standard Classification for Software Anomalies*.

Insfrán, E., Pastor, O. AND Wieringa, R. 2002. *Requirements Engineering-Based Conceptual Modelling*. Journal Requirements Engineering (RE) 61-72.

ISO 1992. *ISO 31 - Quantities and units.*

ISO 1994. *ISO 5725-2 – Accuracy (trueness and precision) of Measurements Methods and Results – Part 2: Basic Method for the Determination of the Repeatability and Reproducibility of a Standard Measurement Method.*

ISO 1998. *ISO/IEC 14143-1 – Information Technology – Software Measurement – Functional Size Measurement – Part 1: Definition of Concepts*

ISO 2000. *ISO Standard 9000-2000: Quality Management Systems: Fundamentals and Vocabulary.*

ISO 2002. *ISO/IEC 14143-2 – Information Technology – Software Measurement – Functional Size Measurement – Part 2: Conformity Evaluation of Software Size Measurement Methods to ISO/IEC 14143-1:1998*

ISO 2003. *ISO/IEC 14143-3 – Information Technology –Software measurement – Functional size measurement –Verification of functional size measurement methods.*

ISO 2004. *International vocabulary of basic and general terms in metrology (VIM).*

ISO/IEC 2001. *ISO/IEC 9126-1, Software Eng. – Product Quality – Part 1: Quality model.*

ISO/IEC 2002. *ISO/IEC 20968, Software Engineering – Mk II Function Point Analysis – Counting Practices Manual*

ISO/IEC 2003a. *ISO/IEC 19761, Software Engineering – COSMIC-FFP – A Functional Size Measurement Method.*

ISO/IEC 2003b. *ISO/IEC 20926, Software Engineering – IFPUG 4.1 Unadjusted Functional Size Measurement Method – Counting Practices Manual.*

ISO/IEC 2005. *ISO/IEC 24570, Software Engineering – NESMA Functional Size Measurement Method version 2.1 – Definitions and Counting Guidelines for the application of Function Point Analysis.*

ISO/IEC 2008. *ISO/IEC 29881, Software Engineering – FiSMA Functional Size Measurement Method version 1.1.*

ISO/IEC 2011. *ISO/IEC 19761, Software Engineering – COSMIC – A Functional Size Measurement Method.*

Jacquet, J.P. AND Abran, A. 1997. *From Software Metrics to Software Measurement Methods: A Process Model.* In Proceedings of the 3rd International Standard Symposium and Forum on Software Engineering Standards (ISESS), Walnut Creek, USA1997, 1-12.

Jenner, M.S. 2001. *COSMIC-FFP and UML: Estimation of the Size of a System Specified in UML – Problems of Granularity.* In Proceedings of the 4th European Conf. Soft. Measurement and ICT Control2001, 173-184

Jenner, M.S. 2002. *Automation of Counting of Functional Size Using COSMIC-FFP in UML.* In Proceedings of the 12th International Workshop Software Measurement2002, 43-51.

Kemerer, C.F. 1993. *Reliability of Function Points Measurement.* Communications of the ACM 36(2), 85-97.

Khelifi, A. 2005. *A Set of References for Software Measurement with ISO 19761 (COSMIC-FFP): an Exploratory Study.*Phd Thesis. In Département du génie logiciel École de Technologie Supérieure, Montréal, Canada, 495.

Khelifi, A. AND Abran, A. 2007. *Software Measurement Standard Etalons: A Design Process*. INTERNATIONAL JOURNAL OF COMPUTERS 1(3), 41-48.

Khelifi, A., Abran, A., Symons, C., Desharnais, J.M., Machado, F., Jayakumar, J. AND Leterthuis, A. 2003. *The C-Registration System Case Study with ISO 19761*. http://www.gelog.etsmtl.ca/cosmic-ffp/casestudies_with_ISO_19761_2003.html.

Kim, D.-K., France, R. AND Ghosh, S. 2004. *A UML-based language for specifying domain-specific patterns*. Journal of Visual Languages & Computing 15(3-4), 265-289.

Kitchenham, B. 1997. *Counterpoint: The Problem with Function Points*. IEEE Software Status Report 14(2), 29-31.

Kitchenham, B., Pfleeger, S.L. AND Fenton, N. 1995. *Towards a Framework for Software Measurement Validation*. IEEE Transactions on Software Engineering 21(12), 929 -944.

Kruchten, P. 2000. *The Rational Unified Process: An Introduction*. Addison Wesley.

Kuzniarz, L. 2003. *Inconsistencies in Student Designs*. In Proceedings of the Workshop on Consistency Problems in UML-based Software Development II, San Francisco, USA, October 20 2003 IEEE, 9-17.

Laitenberger, O., Atkinson, C., Schlich, M. AND Emam, K.E. 2000. *An experimental comparison of reading techniques for defect detection in UML design documents*. Journal of Systems & Software 53(2), 183-204.

Lange, C. AND Chaudron, M. 2004. *An Empirical Assessment of Completeness in UML Designs*. In Proceedings of the 8th Conf. on Empirical Assessment in Software Eng. (EASE), May 2004 2004 IEEE, 111-121.

Lange, C. AND Chaudron, M. 2006. *Effects of Defects in UML Models – An Experimental Investigation*. In Proceedings of the 28th Int. Conf. on Software Eng. (ICSE), Shanghai, China, May 20–28 2006 ACM, 401-410.

Lange, C. AND Chaudron, M. 2007. *Defects in Industrial UML Models – A Multiple Case Study*. In Proceedings of the 2nd Workshop on Quality in Modeling (QiM) of MODELS, Nashville, TN, USA2007, 50-79.

Lange, C., Wijins, M. AND Chaudron, M. 2007. *Metric View Evolution: UML-based Views for Monitoring Model Evolution and Quality*. In Proceedings of the 11th European Conference on Software maintenance and Reengineering (CSMR'07), Amsterdam, The Netherlands, March 2007 2007 IEEE, 327-328.

Lehne, A. 1997. *Experience Report: Function Points Counting of Object-Oriented Analysis and Design based on the OOram method*. In Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Atlanta, Georgia1997.

Leung, F. AND Bolloju, N. 2005. *Analyzing the Quality of Domain Models Developed by Novice Systems Analysts*. In Proceedings of the 38th Hawaii International Conference on System Sciences2005 IEEE, 1-7.

Levesque, G., Bevo, V. AND Cao, D.T. 2008. *Estimating software size with UML models*. In Proceedings of the C3S2E Conference, Montreal2008, 81-87.

Lindland, O.I., Sindre, G. AND Solvberg, A. 1994. *Understanding Quality in Conceptual Modeling*. IEEE Software 11(2), 42-49.

Lother, M. AND Dumke, R. 2001. *Point Metrics-Comparison and Analysis*. In Proceedings of the Current Trends in Software Measurement, Aachen, Germany2001 Shaker Publ, 228-267.

March, S.T. AND Smith, G.F. 1995. *Design and Natural Science Research on Information Technology*. Decision Support Systems 15, 251-266.

Marín, B., Condori-Fernández, N. AND Pastor, O. 2007. *Calidad en Modelos Conceptuales: Un Análisis Multidimensional de Modelos Cuantitativos basados en la ISO 9126*. Revista de Procesos y Métricas de las Tecnologías de la Información 4, 153-167.

Marín, B., Giachetti, G. AND Pastor, O. 2008. *Measurement of Functional Size in Conceptual Models: A Survey of Measurement Procedures Based on COSMIC*. In Proceedings of the IWSM/Metrikon/Mensura, Munich, Germany 2008, R.D.E. Al. Ed. Springer, 170-183.

Meli, R., Abran, A., Ho Vinh, T. AND Oligny, S. 2000. *On the Applicability of COSMIC-FFP for Measuring Software Throughout its Life Cycle*. In Proceedings of the 11th European Software Control and Metrics Conference Munich April 18-20 2000, 1-10.

Mellor, S. AND Balcer, J. 2002. *Executable UML: A Foundation for Model-Driven Architecture*. Addison Wesley

Mellor, S.J., Clark, A.N. AND Futagami, T. 2003. *Guest Editors' Introduction: Model-Driven Development*. IEEE Software 20, 14-18.

Mohagheghi, P. AND Aagedal, J. 2007. *Evaluating Quality in Model-Driven Engineering*. In Proceedings of the International Workshop on Modeling in Software Engineering (MISE'07) 2007 IEEE Computer Society.

Molina, P. 2003. *Especificación de interfaz de usuario: De los requisitos a la generación automática*. Doctoral thesis Universidad Politécnica de Valencia, Valencia, España.

Moody, D.L. 2005. *Theoretical and practical issues in evaluating the quality of conceptual models: current state and future directions*. Data & Knowledge Engineering 55(3), 243-276.

Nagano, S. AND Ajisaka, T. 2003. *Functional metrics using COSMIC-FFP for object-oriented real-time systems*. In Proceedings of the 13th

International Workshop on Software Measurement (IWSM), Montreal, Canada, September 23-25 2003, 1-7.

Neuman, W.L. 2000. *Social Research Methods—Qualitative and Quantitative Approaches*. Needham Heights, MA, USA.

OECD 2010. *Glossary of Statistical Terms*. <http://stats.oecd.org/glossary/index.htm>. (Last accessed November 2010).

Olivé, A. AND Raventós, R. 2006. *Modeling events as entities in object-oriented conceptual modeling languages*. *Data and Knowledge Engineering* 58(3), 243-262.

OMG 2006a. *MOF 2.0 Core Specification*.

OMG 2006b. *Object Constraint Language 2.0 Specification*.

OMG 2010. *UML 2.3 Superstructure Specification*. www.omg.org/spec/UML/2.1.2/.

Opdahl, A.L. AND Henderson-Sellers, B. 2005. *A Unified Modelling Language without referential redundancy*. *Data & Knowledge Engineering* 55(3), 277-300.

Panach, J.I., España, S., Moreno, A. AND Pastor, O. 2008. *Dealing with Usability in Model Transformation Technologies*. In *Proceedings of the ER 2008, Barcelona2008* Springer 498-511.

Pastor, O., Gómez, J., Insfrán, E. AND Pelechano, V. 2001. *The OO-Method Approach for Information Systems Modelling: From Object-Oriented Conceptual Modeling to Automated Programming*. *Information Systems* 26(7), 507–534.

Pastor, O., Hayes, F. AND Bear, S. 1992. *OASIS: An Object-Oriented Specification Language*. In *Proceedings of the Int. Conference on Advanced Information Systems Engineering (CAiSE)*, Manchester, UK1992, 348–363.

Pastor, O. AND Molina, J.C. 2007. *Model-Driven Architecture in Practice: A Software Production Environment Based on Conceptual Modeling*. Springer, New York.

Poels, G. 2002. *A Functional Size Measurement Method for Event-Based Object-oriented Enterprise Models*. In Proceedings of the Int. Conf. on Enterprise Inf. Systems (ICEIS)2002, 667-675

Poels, G. 2003a. *Definition and Validation of a COSMIC-FFP Functional Size Measure for Object-Oriented Systems*. In Proceedings of the 7th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE), Darmstadt, Germany, July 2003a, 1-6.

Poels, G. 2003b. *Functional Size Measurement of Multi-Layer Object-Oriented Conceptual Models*. In Lecture Notes in Computer Science 2817 Springer Berlin / Heidelberg, 334-345.

Poels, G. AND Dedene, G. 2000. *Distance-based software measurement: necessary and sufficient properties for software measures*. Information and Software Technology 42, 35-46.

Robson, C. 2002. *Real World Research: A resource for social scientists and practitioner-researchers*. Blackwell (2nd Edition).

Runeson, P. AND Host, M. 2009. *Guidelines for conducting and reporting case study research in software engineering*. Empirical Software Engineering Journal 14(2), 131-164.

Schmidt, D. 2006. *Model Driven Engineering*. IEEE Computer 39(2), 25-31.

Schneidewind, N.E. 1992. *Methodology for Validating Software Metrics*. IEEE Transactions on Software Engineering 18(5), 410-422.

Seaman, C. 1999. *Qualitative methods in empirical studies of software engineering*. IEEE Transactions on Software Engineering 25(4), 557-572.

Selic, B. 2003. *The Pragmatics of Model-Driven Development*. IEEE Software 20(5), 19–25.

Selic, B. 2007. *A Systematic Approach to Domain-Specific Language Design Using UML*. In Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)2007, 2–9.

Selic, B., Gullekson, G. AND Ward, P.T. 1994. *Real-time Object Oriented Modelling*. Wiley.

Sellami, A. AND Abran, A. 2003. *The Contribution of Metrology Concepts to Understanding and Clarifying a Proposed Framework for Software Measurement Validation*. In Proceedings of the 13th International Workshop on Software Measurement – IWSM, Montreal (Canada), September 23–25 2003 Shaker Verlag ISBN: 3-8322-1880-7, 18-40.

Shlaer, S. AND Mellor, S. 1992. *Object Lifecycles: Modelling the World in States*. Yourdon Press, Prentice-Hall.

St-Pierre, D., Maya, M., Abran, A., Desharnais, J.-M. AND Bourque, P. 1997. *Full Function Points: Counting Practices Manual*.

Standish_Group 2010. *CHAOS Summary 2010*.

Tavares, H., Carvalho, A. AND Castro, J. 2002. *Medicao de Pontos por Funcao a partir da Especificacao de Requisitos*. In Proceedings of the Workshop on Requirements Engineering (WER), Valencia, Spain2002, 278-298.

Teijlingen, E.v. AND Hundley, V. 2001. *The importance of pilot studies*. Social Research Update 35.

Tran-Cao, D., Levesque, G. AND Abran, A. 2002. *Measuring Software Functional Size: Towards an Effective Measurement of Complexity*. In

Proceedings of the International Conference on Software Maintenance2002
IEEE Computer Society, 370-376.

Travassos, G., Shull, F., Fredericks, M. AND Basili, V. 1999. *Detecting Defects in Object-Oriented Designs: Using Reading Techniques to Increase Software Quality*. In Proceedings of the OOPSLA' 99, Denver, CO, USA1999, 47-56.

Trudel, S. AND Abran, A. 2008. *Improving Quality of Functional Requirements by Measuring Their Functional Size*. In Proceedings of the IWSM/Metrikon/Mensura, Munich, Germany2008, R.D.E. Al. Ed. Springer, 287-231.

Trudel, S. AND Abran, A. 2010. *Functional Requirements Improvements through Size Measurement: A Case Study with Inexperienced Measurers*. In Proceedings of the 8th ACIS International Conference on Software Engineering Research, Management and Applications - SERA 2010, Montreal, May 24-26 2010 IEEE-CS Press, 181-189.

Uemura, T., Kusumoto, S. AND Inoue, K. 1999. *Function Point Measurement Tool for UML Design Specification*. In Proceedings of the 5th IEEE International Software Metrics Symposium (METRICS), Florida, USA1999, 62-71.

Vaishnavi, V. AND Kuechler, W. 2007. *Design Research in Information Systems*. <http://www.isworld.org/Researchdesign/drisISworld.htm>. (Last accessed May 2009).

Vanderdonckt, J. 2008. *Model-Driven Engineering of User Interfaces: Promises, Successes, and Failures*. In Proceedings of the 5th Annual Romanian Conf. on Human-Computer Interaction ROCHI'2008, Iasi, 18-19 September 2008, S. Buraga AND I. Juvina Eds. Matrix ROM, 1-10.

Wilson, R.M., Runciman, W.B., Gibberd, R.W., Harrison, B.T., Newby, L. AND Hamilton, J.D. 1995. *The quality in Australian health care study*. The Medical Journal of Australia.

Wohlin, C., Runeson, P., Host, M., Ohlsson, M., Regnell, B. AND Wesslén, A. 2000. *Experimentation in Software Engineering - An Introduction*. Kluwer Academic Publishers.

Yin, R. 2003. *Case study research. Design and methods*. Sage (3rd Edition), London.

Zuse, H. 1998. *A Framework for Software Measurement*. Walter de Gruyter, Germany, Berlin.

Appendix A

OOmCFP Measurement Guide

The OOmCFP (OO-Method COSMIC Function Points) procedure has been developed to measure the functional size of the applications generated in the OO-Method MDD environment from their conceptual models. The OOmCFP measurement procedure was defined in accordance with the COSMIC measurement manual version 3.0.

The application of the OOmCFP measurement procedure is composed by the following steps:

1. Determinate the purpose of the measurement.
2. Determinate the scope of the measurement.
3. Identify the functional users and the boundaries between the layers.
4. Determinate the granularity level.
5. Identify the functional process.
6. Eliminate the duplicity of the functional process.
7. Identify the data groups.
8. Identify the data attributes (optional).
9. Identify the data movements.
10. Apply the measurement function.
11. Calculate the functional size.

12. Report the measurement results.

Step 1: Determinate the purpose of the measurement.

- Purpose: Measuring the accurate functional size of the OO-Method applications generated in an MDD environment from the involved conceptual models to estimate the cost of these applications specifically generated by the OlivaNova tool from the developer's viewpoint.

Step 2: Determinate the scope of the measurement.

The OOmCFP measurement procedure uses the OO-Method conceptual model as the input artefact for the measurement of the functional size of applications generated in MDD environments. This conceptual model formally and unambiguously specifies the functional requirements of the applications independently of the technological characteristics that the generated applications will have.

- Scope: OO-Method conceptual model, which is comprised of four models (Object, Dynamic, Functional, and Presentation).

The OO-Method software applications are generated according to a three-tier software architecture: presentation, logical, and database. These tiers correspond with the layer definition of the COSMIC Measurement Manual [Abran *et al.* 2007]. Thus, we distinguish three layers in an OO-Method application: a client layer, which contains the graphical user interface; a server layer, which contains the business logic of the application; and a database layer, which contains the persistence of the applications.

In each layer of an OO-Method application, there is a piece of software that can interchange data with the pieces of software of the other layers. Thus, we distinguish, respectively, three pieces of software in an OO-

Method application: the client piece of software, the server piece of software, and the database piece of software. Finally, there are no peer components in each piece of software of the OO-Method applications.

Step 3: Identify the functional users and the boundaries between the pieces of software.

- Rule 1: Identify a human functional user for each agent class in the OO-Method object model.
- Rule 2: Identify one boundary between the human functional user and the client layer.
- Rule 3: Identify a client functional user for the client component of an OO-Method application.
- Rule 4: Identify one boundary between the client functional user and the server layer.
- Rule 5: Identify a server functional user for the server component of an OO-Method application.
- Rule 6: Identify one boundary between the server functional user and the database layer.
- Rule 7: Identify a legacy functional user for each legacy view in the OO-Method object model.
- Rule 8: If exists a legacy functional user, identify one boundary between the legacy functional user and the server layer.

Step 4: Determinate the granularity level.

- Granularity level: Low

Step 5: Identify the functional process.

A functional process starts with an entry data movement carried out by a functional user given that an event (triggering event) has happened. A functional process ends when all the data movements needed to generate the answer to this event have been executed. Thus, a functional process has at least two data movements (1 entry/read data movement + 1 exit/write data movement). In OOmCFP, the ‘human functional user’, the ‘client functional user’, the ‘server functional user’, and the ‘legacy functional user’ start functional processes.

Rule 9: Identify a functional process in the client layer for each Population Interaction Unit (PIU), Service Interaction Unit (SIU) or Master Detail Interaction Unit (MDIU) that is a direct child of the hierarchy action tree (HAT) of the presentation model of the OO-Method conceptual model.

Rule 9.1: For each PIU, identify the Display Set, Action Set, Navigation Set, Filter, Order Criteria, and the interaction units that are contained in these patterns.

Rule 9.2: For each SIU, identify the arguments, the Conditional Navigations, and the interaction units that are contained in these patterns.

Rule 9.3: For each MDIU, identify the master part and the detail part, and the interaction units that are contained in these patterns.

Rule 9.4: For each IIU, identify the Display Set, Action Set, Navigation Set, and the interaction units that are contained in these patterns.

Rule 10: A functional process corresponds to the set of formulae (derivations, default values, filters, valuations, integrity constraints, triggers, transactions, preconditions, dependency rules, control conditions, and conditional navigation) that solve the Server layer in response to the events that occur in the functional processes of the Client layer.

Rule 10.1: For each functional process in the server layer, name it using the name of the client functional process that started it.

Step 6: Eliminate the duplicity of the functional process.

Rule 11: Do not consider in the functional size of a functional process FP_B the functional size of a functional process FP_A that is contained in the functional process FP_B.

Rule 12: Only consider one time the functional size of an IIU, PIU, MDIU or SIU that is auto contained.

Step 7: Identify the data groups.

Rule 13: Identify a data group for each class that is not part of an inheritance hierarchy in the object model that participates in a functional process.

Rule 14: Identify a data group for the parent class of an inheritance hierarchy in the object model of a class that participates in a functional process belongs to.

Rule 15: Identify a data group for each child class that has different attributes than his parent of an inheritance hierarchy in the object model of a class that participates in a functional process belongs to.

Step 8: Identify the data attributes (optional).

Rule 16: Identify a data attribute for each attribute of the classes in the object model that are identified as data groups.

Step 9: Identify the data movements.

To identify the data movements of every functional process identified in the step 5, the following counting rules must be applied. Since all the data movements of the generated applications can be identified focusing in three main conceptual constructs (display sets, filters, and services), the counting rules are grouped by these constructs:

Display Sets:

- Counting Rule 1: 1 **read** data movement for each different *class* that contributes with *attributes* to the display set of a PIU or IIU that participates in a functional process of the **server layer**.
- Counting Rule 2: 1 **entry** data movement for each different *legacy view* that contributes with *attributes* to the display set of a PIU or IIU that participates in a functional process in the **server layer**.
- Counting Rule 3: 1 **exit** data movement for each different *class* or *legacy view* that contributes with *attributes* to the display set of a PIU or IIU that participates in a functional process in the **server layer**.
- Counting Rule 4: 1 **entry** data movement for each different *class* or *legacy view* that contributes with *attributes* to the display set of a PIU or IIU that participates in a functional process of the **client layer**.
- Counting Rule 5: 1 **exit** data movement for *all* the *attributes* that are shown in a display set of a PIU or IIU that participates in a functional process of the **client layer**.
- Counting Rule 6: 1 **read** data movement for each different *class* that is used in the *derivation formula* of the derived attributes of the display set of a PIU or IIU that

participates in a functional process in the **server layer**.

Counting Rule 7: 1 **read** data movement for each different *class* that is used in the *condition of the derivation formula* of the derived attributes of the display set of a PIU or IIU that participates in a functional process in the **server layer**.

Counting Rule 8: 1 **entry** data movement for each different *legacy view* that is used in the *derivation formula* of an attribute of a display set of a PIU or IIU that participates in a functional process in the **server layer**.

Counting Rule 9: 1 **entry** data movement for each different *legacy view* that is used in the *condition of a derivation formula* of an attribute of a display set of a PIU or IIU that participates in a functional process in the **server layer**.

Filters:

Counting Rule 10: 1 **entry** data movement (represented by the class that contains the filter) for the *set of data-valued variables* that are associated to a filter of a PIU that participates in a functional process of the **client layer**.

Counting Rule 11: 1 **entry** data movement for each different *object-valued variables* that is associated to a filter of a PIU

that participates in a functional process in the **client layer**.

Counting Rule 12: 1 **exit** data movement (represented by the class that contains the filter) for the *set of data-valued variables* that are associated to a filter of a PIU that participates in a functional process in the **client layer**.

Counting Rule 13: 1 **exit** data movement for each different *object-valued variable* that is associated to a filter of a PIU that participates in a functional process in the **client layer**.

Counting Rule 14: 1 **entry** data movement (represented by the class that contains the filter) for the *set of data-valued variables* that are associated to a filter of a PIU that participates in a functional process in the **server layer**.

Counting Rule 15: 1 **entry** data movement for each different *object-valued variable* that is associated to a filter of a PIU that participates in a functional process in the **server layer**.

Counting Rule 16: 1 **read** data movement for each different *class* that is used in the *filter formula* of a filter of a PIU that participates in a functional process in the **server layer**.

Counting Rule 17: 1 **entry** data movement for each different *legacy view* that is used in the *filter formula* of a filter of a PIU that participates in a functional process in the **server layer**.

Counting Rule 18: 1 **exit** data movement for each different *class* that is used in the formula of the *default value* of an *object-valued variable* that is associated to a filter of a PIU that participates in a functional process in the **server layer**.

Counting Rule 19: 1 **exit** data movement (represented by the class that contains the filter) for the *set of data-valued variables* that has a *default value*, and that are associated to a filter of a PIU that participates in a functional process in the **server layer**.

Counting Rule 20: 1 **entry** data movement for the *default value* of an *object-valued variable* that is associated to a filter of a PIU that participates in a functional process in the **client layer**.

Counting Rule 21: 1 **entry** data movement (represented by the class that contains the filter) for the *set of data-valued variables* that has a *default value*, and that are associated to a filter of a PIU that participates in a functional process in the **client layer**.

Counting Rule 22: 1 **exit** data movement (represented by the class that contains the filter) for the *set of data-valued*

variables that has a *default value*, and that are associated to a filter of a PIU that participates in a functional process in the **client layer**.

Counting Rule 23: 1 **exit** data movement for the *default value* of an *object-valued variable* that is associated to a filter of a PIU that participates in a functional process in the **client layer**.

Services:

Counting Rule 24: 1 **read** data movement for each different *class* that is used in the *formula of the preconditions* of a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 25: 1 **entry** data movement for each different *legacy view* that is used in the *formula of the preconditions* of a SIU that participates in a functional process in the **server layer**.

Counting Rule 26: 1 **read** data movement for each different *class* that is used in the *formula of the error messages* associated to the *preconditions* of a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 27: 1 **entry** data movement for each different *legacy view* that is used in the formula of the *error messages* associated to the *preconditions* of a SIU that

participates in a functional process in the **server layer**.

Counting Rule 28: 1 **exit** data movement for each different *class* or *legacy view* that is used in the formula of the *error messages* associated to the *preconditions* of a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 29: 1 **entry** data movement for each different *class* or *legacy view* that is used in the *formula of the error messages* associated to the *preconditions* of a service related to a SIU that participates in a functional process in the **client layer**.

Counting Rule 30: 1 **exit** data movement for all the *error messages* associated to the *preconditions* of a service related to a SIU that participates in a functional process in the **client layer**.

Counting Rule 31: 1 **entry** data movement (represented by the class that contains the SIU) for the set of *data-valued arguments* of a SIU that participates in a functional process in the **client layer**.

Counting Rule 32: 1 **entry** data movement for each different *class* that corresponds to an *object-valued argument* of a SIU that participates in a functional process in the **client layer**.

Counting Rule 33: 1 **exit** data movement (represented by the class that contains the SIU) for the set of *data-valued arguments* of a SIU that participates in a functional process in the **client layer**.

Counting Rule 34: 1 **exit** data movement for each different *class* that corresponds to an *object-valued argument* of a SIU that participates in a functional process in the **client layer**.

Counting Rule 35: 1 **entry** data movement (represented by the class that contains the SIU) for the set of *data-valued arguments* of a SIU that participate in a functional process in the **server layer**.

Counting Rule 36: 1 **entry** data movement for each different *class* that corresponds to an *object-valued argument* of a SIU that participates in a functional process in the **server layer**.

Counting Rule 37: 1 **read** data movement for each different *class* that is used in the *condition of the valuation formula* of the event related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 38: 1 **read** data movement for each different *class* that is used in the *valuation formula* of the event related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 39: 1 **read** data movement for each different *class* that is used in the *formula of the transaction, operation or global service* related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 40: 1 **entry** data movement for each different *legacy view* that is used in the *condition of the valuation formula* of a SIU that participates in a functional process in the **server layer**.

Counting Rule 41: 1 **entry** data movement for each different *legacy view* that is used in the *valuation formula* of a SIU that participates in a functional process in the **server layer**.

Counting Rule 42: 1 **entry** data movement for each different *legacy view* that is used in the *formula of the transaction, operation or global service* related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 43: 1 **write** data movement for the *class* that contains a *destroy event* related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 44: 1 **write** data movement for the *class* that contains a *creation event* related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 45: 1 **write** data movement for the *class* that contains an *event that has valuations* and that is related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 46: 1 **exit** data movement for each different *class* that is used in the formula of the *default value* of an *object-valued argument* that is associated to a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 47: 1 **exit** data movement (represented by the class that contains the SIU) for the set of *data-valued argument* that has a *default value* and that are associated to a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 48: 1 **entry** data movement for each different *class* that is used in the formula of the *default value* of an *object-valued argument* that is associated to a service related to a SIU that participates in a functional process in the **client layer**.

Counting Rule 49: 1 **entry** data movement (represented by the class that contains the SIU) for the set of *data-valued argument* that has a *default value* and that are associated to a service related to a SIU that participates in a functional process in the **client layer**.

Counting Rule 50: 1 **exit** data movement (represented by the class that contains the SIU) for the set of *data-valued argument* that has a *default value* and that are associated to a service related to a SIU that participates in a functional process in the **client layer**.

Counting Rule 51: 1 **exit** data movement for the *default value of an object-valued argument* that is associated to a service related to a SIU that participates in a functional process in the **client layer**.

Counting Rule 52: 1 **read** data movement for each different *class* that is used in the *formula of the integrity constraints* of a class that contains a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 53: 1 **entry** data movement for each different *legacy view* that is used in the *formula of the integrity constraints* of a class that contains a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 54: 1 **read** data movement for each different *class* that is used in the *formula of the error messages* associated to the *integrity constraints* of a class that contains a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 55: 1 **entry** data movement for each different *legacy view* that is used in the formula of the *error messages* associated to the *integrity constraints* of a class that contains a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 56: 1 **exit** data movement for each different *class* or *legacy view* that is used in the formula of the *error messages* associated to the *integrity constraints* of a class that contains a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 57: 1 **entry** data movement for each different *class* or *legacy view* that is used in the *formula of the error messages* associated to the *integrity constraints* of a class that contains a service related to a SIU that participates in a functional process in the **client layer**.

Counting Rule 58: 1 **exit** data movement for all the *error messages* associated to the *integrity constraints* of a class that contains a service related to a SIU that participates in a functional process in the **client layer**.

Counting Rule 59: 1 **read** data movement for each different *class* that is used in the *formula of the control condition of a service* related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 60: 1 **entry** data movement for each different *legacy view* that is used in the *formula of the control condition* of a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 61: 1 **read** data movement for each different *class* that is used in the *condition formula of the triggers* of the class that contains a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 62: 1 **entry** data movement for each different *legacy view* that is used in the *condition formula of the triggers* of the class that contains a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 63: 1 **entry** data movement for each different *legacy view* that is used in the *action formulae of the dependency rules* of the arguments of a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 64: 1 **entry** data movement for each different *legacy view* that is used in the *condition formulae of the dependency rules* of the arguments of a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 65: 1 **read** data movement for each different class that is used in the *formulae of the dependency rules of the arguments* of a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 66: 1 **read** data movement for each different class that is used in the *condition formulae of the dependency rules of the arguments* of a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 67: 1 **read** data movement for each different *class* that is used in the *conditional navigation formula* of a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 68: 1 **entry** data movement for each different *legacy view* that is used in the *conditional navigation formula* of a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 69: 1 **read** data movement for each different *class* that is used in the *condition of the formula of the arguments initialization* of a SIU associated to the conditional navigation of a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 70: 1 **entry** data movement for each different *legacy view* that is used in the *condition of the formula of the arguments initialization* of a SIU associated to the conditional navigation of a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 71: 1 **read** data movement for each different *class* that is used in the *formula of the arguments initialization* of a SIU associated to the conditional navigation of a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 72: 1 **entry** data movement for each different *legacy view* that is used in the *formula of the arguments initialization* of a SIU associated to the conditional navigation of a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 73: 1 **read** data movement for each different *class* that is used in the *formula of the navigational filtering* of an IIU, PIU, or MDIU associated to the conditional navigation of a service related to a SIU that participates in a functional process in the **server layer**.

Counting Rule 74: 1 **entry** data movement for each different *legacy view* that is used in the *formula of the navigational filtering* of an IIU, PIU, or MDIU associated to the

conditional navigation of a service related to a SIU that participates in a functional process in the **server layer**.

Step 10: Apply the measurement function.

- Measurement Function: OOmCFP assigns *1 CFP* to each data movement.

Step 11: Calculate the functional size.

Measurement Rule 1: Aggregate the data movements of a functional process that occur in the client layer to obtain the functional size of the functional process. Formula (1) shows how to calculate the functional size of a functional process.

Measurement Rule 2: Aggregate the data movements of a functional process that occur in the server layer to obtain the functional size of the functional process. Formula (1) shows how to calculate the functional size of a functional process.

$$\text{SizeFunctionalProcess (CFP v3.0)} = \sum_{i=1}^n \text{DataMovement}_i \quad (2)$$

Measurement Rule 3: Aggregate the functional size of each functional process of the client layer to obtain the functional size of the client layer. Formula

(2) shows how to calculate the functional size of a layer.

Measurement Rule 4: Aggregate the functional size of each functional process of the server layer to obtain the functional size of the server layer. Formula (2) shows how to calculate the functional size of a layer.

$$\text{SizeLayer (CFP v3.0)} = \sum_{i=1}^n \text{SizeFunctionalProcess}_i \quad (2)$$

Measurement Rule 5: Aggregate the functional size of each layer of the OO-Method application to obtain the functional size of the application. Formula (3) shows how to calculate the functional size of the generated application.

$$\text{SizeOOMethodApplication (CFP v3.0)} = \sum_{i=1}^n \text{SizeLayer}_i \quad (3)$$

Step 12: Report the measurement results.

Functional Process	Contained Elements			Client		Server			Total Functional Size (CFP v3.0)
				E	X	E	X	R	
Total Functional Size Layer (CFP v3.0)									
Total Functional Size Application (CFP v3.0)									

Appendix B

Conformity Evaluation

Checklist

The conformity evaluation of OOmCFP FSM procedure regarding to the COSMIC FSM Method version 3.0 has been performed using the checklist presented in this appendix.

The evaluation checklist is comprised of four parts. The first part has questions related to the strategy phase of COSMIC. The second part has questions related to the mapping phase of COSMIC. The third part has questions related to the measurement phase of COSMIC. And, the fourth part has questions related to the presentation of the measurement results.

The evaluation checklist has a table structure in each part. The first column of the checklist presents the evaluation questions. The second column must be filled by the evaluators with the location of the relevant information of OOmCFP that is used to respond the question. The third column must be filled by the evaluators depending on the answer of the question satisfied or not the requirement of COSMIC. If a requirement is marked as not satisfied or there is no information to solve the question, the evaluators must justify their decision. The last column presents the location

of the requirements of COSMIC in its measurement manual version 3.0 [Abran *et al.* 2007].

Strategy Phase

Evaluation Question	Location	Satisfies?	Corresponding requirements
Is the purpose of the measurement defined in the OOmCFP measurement procedure?			2.1
Is the scope of the measurement defined in the OOmCFP measurement procedure?			2.2.1
Can be the FUR derived from the software to be measured by OOmCFP?			1.2
Are the levels of decomposition of the software to be measured identified by the OOmCFP measurement procedure?			2.2.2
Are the layers of the software to be measured identified by the OOmCFP measurement procedure?			2.2.3
Are the pieces of the software to be measured identified by the OOmCFP measurement procedure			2.2.1
Are the peer-components of the software to be measured identified by the OOmCFP measurement procedure?			2.2.4
It is possible to identify the functional users using the OOmCFP measurement procedure?			2.3.2
It is possible to identify the boundaries using the measurement procedure?			2.3.2
Is the granularity level of the functional processes defined in the measurement procedure?			2.4

Justifications

.....

.....

.....

Mapping Phase

Evaluation Question	Location	Satisfies?	Corresponding requirements
It is possible to identify the functional processes using the OOmCFP measurement procedure?			3.2
Are the functional processes identified using OOmCFP triggered by an event?			3.2.1
Is each functional process belonged to only one layer of the software to be evaluated by OOmCFP?			3.2.2
Is each functional process identified using OOmCFP comprised by at least two or more data movements?			3.2.2
It is possible to identify the data groups used by the functional processes using the OOmCFP measurement procedure?			3.3
Are the data groups a unique and distinguishable set of attributes?			3.3.1
It is possible to identify the data attributes used by the functional processes using the OOmCFP measurement procedure?			3.4

Justifications

.....

.....

.....

Measurement Phase

Evaluation Question	Location	Satisfies?	Corresponding requirements
It is possible to identify entry (E) data movements using the OOmCFP measurement procedure?			4.1.2
It is possible to identify exit (X) data movements using the OOmCFP measurement procedure?			4.1.3
It is possible to identify read (R) data movements using the OOmCFP measurement procedure?			4.1.4
It is possible to identify write (W) data movements using the OOmCFP measurement procedure?			4.1.5
It is possible to identify data manipulations not associated with the data movements using OOmCFP?			4.1.6
Is 1 CFP assigned to each data movement by the OOmCFP measurement procedure?			4.2
Is an aggregation function defined to obtain the functional size of each functional process in the OOmCFP measurement procedure?			4.3.1
Is an aggregation function defined to obtain the functional size of each layer in the OOmCFP measurement procedure?			4.3.1
Is an aggregation function defined to obtain the functional size of the data movements' modifications in the OOmCFP measurement procedure?			4.4.1
Is an extension considered by OOmCFP measurement procedure to measure the software size?			4.5

Justifications

.....

.....

.....

Measurement Report

Evaluation Question	Location	Satisfies?	Corresponding requirements
Is the measurement result labeled as the numerical value of the functional size and the symbol CFP (v3.0)?			5.1
There is a template with the needed information to be archived to document the result of a measurement?			5.2

Justifications

.....

.....

.....

Appendix C

Number of Defects in the Case Study

This appendix shows the number of defects related to each defect type found in the models of the case study, whether they were detected by the inspection team or the OOmCFP tool.

Defects detected by the OOmCFP tool in the five models of the case study are the following:

Defect Type	Model1	Model2	Model3	Model4	Model5
CM_NA	1	0	0	4	0
FM_NV	5	8	3	7	7
PM_NM	1	1	1	1	1

Defects detected by the inspection teams in the five models of the case study are the following:

Defect Type	Model1	Model2	Model3	Model4	Model5
CM_Id	0	0	1	0	0
CM_AS	3	0	0	0	0
CM_AN	2	0	0	0	0
CM_FDer	1	0	0	0	0
CM_WArg	0	1	0	0	1
CM_MArg	0	1	0	0	0
CM_DCServ	0	0	0	1	0
CM_SEditC	0	1	0	0	7
CM_InhLib	0	1	0	2	0
CM_InhId	0	1	0	0	1
CM_WInh	1	0	0	0	0
CM_InhServ	0	0	0	1	0
CM_MR	1	0	0	0	0
CM_WCard	0	0	1	0	0
CM_RServ	0	0	0	0	1
CM_FIC	0	1	0	0	0
CM_SAgent	3	4	5	4	5
CM_AAgent	0	0	11	13	0
CM_RAgent	0	2	3	11	0
FM_NV	2	7	0	5	1
DM_SReach	3	0	0	0	0
DM_SWC	1	0	0	0	0
PM_NM	1	1	1	1	1
PM_IP	0	0	0	1	0
