Impact Analysis and Change Management of UML Models

By

Leeshawn L. O'Sullivan

A thesis submitted to

The Faculty of Graduate Studies and Research

In partial fulfilment of the requirements for the degree of

Master of Applied Science

Ottawa-Carleton Institute of Electrical and Computer Engineering

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario, Canada

May 2003

Copyright © 2003 by Leeshawn L. O'Sullivan

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-612-83475-1

Our file Notre référence

ISBN: 0-612-83475-1

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

Canada

The undersigned hereby recommend to
The Faculty of Graduate Studies and Research
Acceptance of the thesis

Impact Analysis and Change Management of UML Models

Submitted by

Leeshawn L. O'Sullivan

In partial fulfilment of the requirements for the degree of
Master of Applied Science

Professor L. C. Briand (Co-Supervisor)

Professor Y. Labiche (Co-Supervisor)

Professor R. A. Goubran (Department Chair)

Carleton University

May 2003

Abstract

The use of UML (Unified Modeling Language) analysis/design models on large projects leads to a large number of interdependent UML diagrams. As software systems evolve, those diagrams undergo changes, for example, to correct errors or address changes in the requirements. These changes can in turn lead to subsequent changes to other elements in the UML diagrams. Impact analysis is then defined as the process of identifying the potential consequences (side-effects) of a change, and estimating what needs to be modified to accomplish a change. Proposed in this thesis is a UML model-based approach to impact analysis that can be applied before any implementation of the changes, thus allowing for early decision-making and change planning. Firstly, each version of a UML model is verified to ensure consistency – consistency verification. After which, changes between two different versions of the model are identified according to a change taxonomy. Model elements that are directly or indirectly impacted (that is, may undergo changes) by these changes are then determined using formally defined impact analysis rules (written in the Object Constraint Language). A measure of distance between a changed element and the potentially impacted ones is also proposed, to prioritize the results of impact analysis according to their likelihood of occurrence. Also presented is a prototype tool that provides automated support for the impact analysis strategy used. The tool is applied to a case study to verify the methodology and assess the practical challenges of its implementation.

Acknowledgements

Firstly, thanks to the Holy Father for without whom nothing is possible.

Thanks to Dr. Briand and Dr. Labiche, for their guidance and advice. In addition, thanks to Telcordia Technologies Inc. for sponsoring this work. Special thanks to my family and friends, especially Kamla, for their help and support – including Jim for his help in proof reading the thesis, and Yihong for her help with OCL. Thanks also to Karin for assisting in the case study.

Table of Contents

Chapter 1 – Introduction	1
1.1 Thesis Contribution.....	3
1.2 Related Works.....	3
1.3 Thesis Organization.....	5
Chapter 2 – Problem Description.....	6
Chapter 3 – Approach Taken	8
Chapter 4 – UML Model Changes	11
4.1 Change Detection Concepts	12
4.1.1 Added/Deleted Model Element.....	15
4.1.2 Changed Model Element.....	16
4.2 UML Metamodel.....	17
4.3 Change Taxonomy	18
4.3.1 Change Taxonomy Model.....	18
4.3.2 Pictorial Representation of the Change Taxonomy.....	21
4.3.3 Change Taxonomy Derivation	21
4.4 Change Detection Criteria.....	24
Chapter 5 – Impact Analysis	27
Chapter 6 – Distance Measure	38
Chapter 7 – Impact Analysis Tool	40
7.1 control Subsystem	40
7.2 umlModel Subsystem.....	43
7.2.1 foundation Package.....	44
7.2.1.1 core Package.....	45
7.2.1.2 dataTypes Package.....	50
7.2.1.3 extensionMechanisms Package.....	51
7.2.2 behaviouralElements Package.....	52
7.2.2.1 commonBehaviour Package.....	53
7.2.2.2 collaborations Package.....	54
7.2.2.3 stateMachines Package.....	57
7.2.3 modelManagement Package	60
7.3 parser Subsystem.....	60
7.4 consistencyVerification Subsystem.....	61
7.5 modelChanges Subsystem	62
7.6 impactAnalysis Subsystem.....	65
7.7 reportGeneration Subsystem.....	67

Chapter 8 – Case Study	69
8.1 Description	69
8.2 Results	71
8.3 Accuracy.....	74
8.4 Conclusions	76
Chapter 9 – Conclusions	78
9.1 Thesis Limitations	79
9.1.1 Practical Limitation.....	79
9.1.2 Limitations of the Methodology.....	80
9.1.3 Implementation Limitations	80
9.2 Future Work	81
References	83
Appendix A – Change Detection	86
A.1 Change Taxonomy	86
A.2 Change Detection Rules	94
Appendix B – Impact Analysis Rules	115
Appendix C – Consistency Rules.....	148
Appendix D – Case Study	152
D.1 Logical Changes	152
D.2 ATM System UML Model (Original Version)	155

List of Figures

Figure 1: Conceptual model outlining the main concepts in this work.....	10
Figure 2: Example metamodel illustrating the concept of property.....	14
Figure 3: Example UML model corresponding to Figure 2.....	15
Figure 4: Example modified version of Figure 3.	17
Figure 5: Example UML metamodel – change taxonomy example.	23
Figure 6: Example change taxonomy – derived from Figure 5.....	24
Figure 7: Example Change Detection Rule – Added Model Elements.....	25
Figure 8: Example Change Detection Rule – Changed Model Elements.	25
Figure 9: Example Change Detection Rule – Deleted Model Elements.	26
Figure 10: Excerpt from IACMTool – impact analysis example.....	32
Figure 11: Example impact analysis rule - CSDVAM.....	34
Figure 12: Class diagram for impact analysis example.....	36
Figure 13: Original sequence diagram for impact analysis example.	36
Figure 14: Changed sequence diagram for impact analysis example.	37
Figure 15: Example impact analysis rule – CCOCpst.....	37
Figure 16: Example of distance between a changed element and the impacted ones.	39
Figure 17: IACMTool subsystem decomposition.	41
Figure 18: control subsystem.....	42
Figure 19: umlModel subsystem.	44
Figure 20: foundation package.	45
Figure 21: foundation::core – main class diagram.	47
Figure 22: foundation::core – classifiers class diagram.	48
Figure 23: foundation::core – relationships class diagram.....	49
Figure 24: foundation::core – class diagram view.	50
Figure 25: foundation::dataTypes package.....	51
Figure 26: foundation::extensionMechanisms package.....	52
Figure 27: behaviouralElements package.....	53
Figure 28: behaviouralElements::commonBehaviour package.....	54
Figure 29: behaviouralElements::collaborations – roles.	55
Figure 30: behaviouralElements::collaborations – interaction.	56
Figure 31: Sequence diagram view class diagram.	57
Figure 32: behaviouralElements::stateMachines – main.....	58
Figure 33: behaviouralElements::stateMachines – events.	59
Figure 34: Statechart diagram view class diagram.....	59
Figure 35: modelManagement package.	60
Figure 36: parser subsystem.	61
Figure 37: consistencyVerification subsystem.	62
Figure 38: Change detection algorithm.....	64
Figure 39: modelChanges subsystem.....	65
Figure 40: Summary of the impact analysis algorithm.	66
Figure 41: impactAnalysis subsystem.	67

Figure 42: reportGeneration subsystem.	68
Figure 43: Cumulative number of all impacted elements vs. distance.....	73
Figure 44: Cumulative number of impacted classes vs. distance.....	74
Figure 45: Cumulative number of impacted operations vs. distance.	74
Figure 46: Change Taxonomy – Changed UMLModel.	86
Figure 47: Change Taxonomy – Changed ClassDiagramView.....	87
Figure 48: Change Taxonomy – Changed SequenceDiagramView.....	88
Figure 49: Change Taxonomy – Changed StatechartDiagramView.....	89
Figure 50: Change Taxonomy – Changed Association.....	90
Figure 51: Change Taxonomy – Changed ClassClassifier.....	91
Figure 52: Change Taxonomy – Changed Attribute.....	91
Figure 53: Change Taxonomy – Changed Operation.....	92
Figure 54: Change Taxonomy – Changed Interface.....	92
Figure 55: Change Taxonomy – Changed Parameter.....	92
Figure 56: Change Taxonomy – Changed State.....	93
Figure 57: Change Taxonomy – Changed StateMachine action.....	93
Figure 58: ATM System – Class Diagram.	155
Figure 59: ATM System – AskingDoAnother Sequence Diagram.	156
Figure 60: ATM System – ATMShutOff Sequence Diagram.	157
Figure 61: ATM System – ATMStartUp Sequence Diagram.	157
Figure 62: ATM System – Cancel Sequence Diagram.....	158
Figure 63: ATM System – CardInsert Sequence Diagram.	158
Figure 64: ATM System – CardNotReadable Sequence Diagram.	159
Figure 65: ATM System – FailedTransaction Sequence Diagram.	159
Figure 66: ATM System – Deposit Sequence Diagram.	160
Figure 67: ATM System – GetPIN Sequence Diagram.....	161
Figure 68: ATM System – Inquiry Sequence Diagram.	161
Figure 69: ATM System – PINInvalid Sequence Diagram.	162
Figure 70: ATM System – PrintReceipt Sequence Diagram.	162
Figure 71: ATM System – Transaction Sequence Diagram.....	163
Figure 72: ATM System – Transfer Sequence Diagram.	164
Figure 73: ATM System – Withdraw Sequence Diagram.....	165

List of Tables

Table 1: Structure of Impact Analysis Rules	31
Table 2: Change Distribution – ATM Case Study	70
Table 3: Code Impacts vs. False-Positives – ATM Case Study	75

Chapter 1 – Introduction

The use of UML (Unified Modeling Language) analysis/design models [15] on large projects leads to a large number of inter-dependent UML diagrams¹. These diagrams undergo changes as software systems are evolving. Such changes to a diagram may lead to subsequent changes to other elements of the same diagram or in other related diagrams. In this context, the (potential) side effects of a change to the unchanged diagrams should be automatically identified to help (1) keep those diagrams up-to-date and consistent, and (2) assess the potential impact of changes in the system. This can in turn help to predict the cost and complexity of changes and help to decide whether to implement them in a new release [7].

In the context of large software development teams, the above problems are even more acute as diagrams may undergo changes in a concurrent manner and different people may be involved in these changes. Support is therefore required to help a team assess the complexity of changes, identify their side effects, and communicate that information to each of the affected team members. In order to address the above issues, the work presented here focuses on *impact analysis* of UML analysis or design models. Impact analysis is defined as the process of identifying the potential consequences (side-effects) of a change, and estimating what needs to be modified to accomplish a change [7].

Most of the research on impact analysis is based on the program code (implementation). However, in the context of UML-based development, it becomes clear

¹ That may also contain Object Constraint Language (OCL) [4] constraints, e.g., operation contracts and guard conditions.

that the complexity of changing analysis and design models is also very high. Therefore, this work seeks to provide automated support to identify changes made to UML model elements and the impact of these changes on other model elements.

While code-based impact analysis methods have the advantage of identifying impacts in the final product – the code, they require the implementation of these changes (or a very precise implementation plan) before impact analysis can be performed. However, a UML model-based approach to impact analysis looks at impacts to the system *before* the implementation of such changes. Then a proper decision can be made earlier, before any change detailed implementation is considered, on whether to implement a particular (set of) change(s) based on what design elements are likely to get impacted, and thus on the likely change cost. Earlier decision-making and change planning is clearly important in the context of rigorous change management. On the other hand, since UML models describe the system at a higher level of abstraction than the code, model-based approaches may provide less precise results than code-based ones. For example, it may be possible that new, unexpected impacts appear at implementation time. This is an issue that requires further investigation but that will not be addressed in this work.

Another assumption made by any model-based impact analysis method is that the model is consistent with the code and up-to-date. This is often an issue in many software development organizations. However, the functionality to manage traceability and consistency between design models and code is now available in some UML CASE tools. For example, Together®, by TogetherSoft™ [5], updates the class diagram when changes

are made to the code and checks some consistency aspects of the updated class diagram with other UML diagrams in the design model.

1.1 Thesis Contribution

The work done for this thesis contributes in several complementary ways to providing support for impact analysis of UML models:

- It defines a methodological framework for (1) detecting changes between two UML model versions, and (2) using these changes to perform impact analysis.
- It provides a set of change detection and impact analysis (side effect) rules, which were derived by systematically analyzing components of UML models (including OCL [4] constraints), and analyzing changes in an actual case study.
- It provides a set of consistency rules that must be satisfied before impact analysis can be performed.
- A prototype tool implements the above principles using a carefully thought-out architecture and an extendable design.
- A case study has been performed to assess the feasibility and practical challenges of the approach taken.

1.2 Related Works

Bohner [8], examines the general issues involved in change impact analysis, and provides structured guidelines to help find solutions to such issues. For example, if one considers both direct and indirect (transitive closure) impacts, the results of impact

analysis show a large number of impacts, thus (possibly) over-estimating the impact. This advocates tool support, as well as the use of semantic (related to the impacts) and structural (for example, distance between a change and an impact) constraints to structure analysis results.

As mentioned in [16], a large number of change impact analysis strategies require source code analysis; where as a few of them are model-based. Kung et al. [17] describes how change impact analysis can be performed from a class diagram, introducing the notion of class firewall (that is, classes that may be impacted by a change in a given class), and discusses the impact of object-oriented characteristics (for example, encapsulation, inheritance, polymorphism, and dynamic binding) on such an analysis. In [18], the authors use a functional model (referred to as "domain model") of the system under consideration to generate test cases, and build a mapping between changes to the domain model and the impact it has on test cases, to classify them. Another method for regression test selection, based on UML models (class and sequence diagrams), is presented in [19]. In this method, rough impact analysis is performed for the sole purpose of classifying the regression test cases as obsolete, re-testable, or reusable. The current work is a significant extension and performs impact analysis at a much more refined level so that it can be applied to a variety of problems, including change effort estimation and supporting the identification of ripple effects. In addition, this approach has been mentioned in [8] but no details were ever proposed or experiments performed. It has been mentioned in [8], that the number of impacts varies exponentially with distance; however, this work shows that the number of impacts varies linearly with distance.

1.3 Thesis Organization

This thesis describes the methodological framework and the fundamental principles underlying the change detection and impact analysis rules, presents an overview of the tool's design, and reports on the case study performed. Consistency verification and other important issues are also discussed. Chapter 2 describes precisely the problem tackled in this work, while the approach taken to solve it is discussed in Chapter 3. The change detection methodology used is described in Chapter 4. Chapter 5 explains the impact analysis methodology adopted. The notion of *distance measure* is addressed in Chapter 6, while Chapter 7 summarizes the design of the tool developed. Chapter 8 presents the case study performed. The main conclusions, limitations of the work, and ideas for future work are discussed in Chapter 9. Appendix A presents the change taxonomy derived in this work as well as the change detection rules defined. The impact analysis rules are defined in Appendix B, followed by the consistency verification rules, which are defined in Appendix C. The details of the case study are then presented in Appendix D.

Chapter 2 – Problem Description

Impact analysis of UML analysis/design models can be decomposed into several sub-problems, as follows.

1. *Verify the consistency of each model version.* The model versions must be self-consistent for any impact analysis algorithm to provide correct results. Since consistency in complex UML models is not always easy to achieve, consistency verification must be supported by tools. Note that this is different from impact analysis since it does not focus on finding (potentially) impacted elements (that is, whose properties or implementation may require change). Instead, consistency verification seeks to determine structural inconsistencies between UML diagrams. For example, in a sequence diagram there exists a classifier role² with a base class C, however C is not defined in any class diagram in the model – this is an inconsistency.
2. *Automatically detect and classify changes* across different versions of UML models. Ideally, a UML model is modified and then the impact analysis tool is used to automatically identify all the changes performed since the last version. It is not desirable for software engineers to specify each change since this would lead to a lot of overhead that would impede the practice of impact analysis. As seen below, changes have to be classified to be able to perform precise impact analysis.

² In the UML standard terminology [1], a *classifier role* identifies an object in a sequence diagram, and the *base class* of the classifier role is the class of this object (the term *base* does not relate to inheritance).

3. *Perform impact analysis* to determine the *potential* side effects of the model changes. In most cases, for reasons described below, side effects cannot be identified with certainty as there is no way to ascertain whether a change is really necessary based on the UML analysis or design model only. As a result, an *impacted element* is a UML model element whose properties or implementation *may* require modification because of a change to another model element (that is, one of its properties may change)³. To clarify the terminology employed, changes to UML diagrams are the result of *logical changes*. A logical change is a high level change (for instance performed by designers) to a UML model that results in a number of model element changes. These high level changes include error corrections, design improvements, and requirement changes. Impact analysis can be performed for each logical change independently or for an entire, new UML model.
4. *Prioritize the impact analysis results* according to the likelihood of occurrence of predicted impacted elements. In object-oriented designs, when considering all direct and indirect dependencies among model elements, impact analysis often results in a large number of (potentially) impacted model elements, thus making their verification impractical. Addressing this issue requires a way to order side effects according to criteria that can be easily evaluated and which are good indicators of the probability of a side effect, for a given change. For example, Briand et al. [20] have explored the use of coupling measures and predictive statistical models for that purpose.

³ Even when there is no change to any of the model element properties, the model element implementation may require change.

Chapter 3 – Approach Taken

As mentioned above, the identification of model inconsistencies is important in ensuring that the impact analysis algorithms used yield correct results. Inconsistencies may be automatically modeled and detected by a set of *consistency rules*. These rules must be implemented in any tool that supports impact analysis on UML models. Note that some CASE (Computer Aided Software Engineering) tools such as Together® [5] provide some amount of consistency checking, however, this is not extensive enough for performing impact analysis.

In this work, consistency verification of a UML model refers to correctness checking on the level of static semantics [6], and consists of checking for inconsistencies across different model diagrams. Each inconsistency check is modeled as a consistency rule (criterion). Such a rule identifies the collection of model elements that cause the inconsistency. If the collection is *not* empty then the inconsistency is present in the model. For the set of consistency rules C , if any $c_i \in C$ returns a non-empty collection then the model is said to be inconsistent.

Sixty-Five (65) consistency rules are defined in this work. A simple example consistency rule is described informally⁴ as follows.

“Each operation that is invoked in a sequence message must be defined in the class diagram, in the specific class of the target object of the message.”

Each element in a UML model is defined by a set of *properties*, for example, a class has a visibility and some attributes. Thus, the identification of a change to a model

⁴ It can also be expressed using OCL on the meta-model.

element requires checking if any of its properties has changed. Each model element change is classified according to a *change taxonomy*, each *leaf change* of which corresponds to one change detection rule. The change detection rules specify formally and precisely how to detect the model changes. The change taxonomy derived in this work reflects changes to class diagrams, sequence diagrams, and statecharts. The complete change taxonomy contains 97 leaf changes. More details are provided in Chapter 4.

Once the consistency of each UML model version has been verified and the changes between the two versions detected, the next step is to automatically perform impact analysis using *impact analysis rules*. That is, rules that determine what model elements may be impacted *directly* or *indirectly* (through transitive closure) by each model element change (Chapter 5). Impact analysis rules depend on the type of change for which impact analysis is performed, thus one such rule is defined for each leaf change in the taxonomy. This resulted in 97 impact analysis rules.

In order for impact analysis to be useful and practical, it is necessary to find ways to indicate what model elements should be checked first as they, and their implementations (code), are more likely to require change. To do so, a measure of distance between a changed element and the resulting potentially impacted elements is defined (Chapter 6). The assumption here is that the larger the distance from the change, the less likely it is that a potentially impacted model element is really impacted.

Figure 1 is a conceptual model (using a class diagram) that provides a useful overview of the concepts presented above.

Figure 1: *Conceptual model outlining the main concepts in this work.*

Chapter 4 – UML Model Changes

UML model changes are the result of *logical changes*, which correspond to error corrections, design improvements, or requirement changes. A logical change usually results in a set of changes to one or more model elements. A set of logical changes is made to a (original) model *version* resulting in a new (changed) version.

The relationship (between two UML models) that specifies that one model is a version of the other is not defined in this work. Instead, the user of any software tool that implements a change detection mechanism based on the principles outlined in this work assumes responsibility for ensuring that the models are related in some way. However, regardless of the strength of the relationship between the two models, such a tool should report on the differences between the models. If there is no relationship between two models then detecting changes between them may produce meaningless results – everything may be changed.

The problem of detecting model changes was divided into the following four sub-problems, each of which is discussed in detail below.

1. Defining *change detection concepts*. That is, defining the fundamental concepts used to determine UML model changes.
2. Defining a *metamodel* for UML models. This metamodel models information about UML models that is necessary for achieving the overall goal of performing impact analysis.

3. Deriving a *taxonomy (classification) of changes* based on the information present in the metamodel. This taxonomy identifies all the changes considered for impact analysis.
4. Defining a precise *change detection criteria* (set of change detection rules), based on the change taxonomy, for change identification. Each change detection criterion (rule) corresponds to exactly one *leaf* change in the taxonomy and details a well-defined specification for identifying that change.

4.1 Change Detection Concepts

In the context of this work, a UML model is an instance (object diagram) of a UML metamodel class diagram. Each UML model object diagram corresponds to one UML model version. A UML metamodel defines the language for specifying a UML model [1].

A UML model is composed of a number of *model elements*, each of which has a number of *properties*. A model element is an instance of a (concrete) subclass of `ModelElement`⁵. A model element property is modeled as the value of a non-static attribute of a model element, or a *change related element* of a model element. Here, the attribute is required to be non-static because the value of a static attribute is the same in all objects that instantiate the attribute's class. Therefore, the value of such an attribute would be the same in all UML model versions. Hence, there would be no point in checking for a change to the value of such an attribute. Note that the value of a non-static attribute inherited from an abstract subclass A of `ModelElement`, by a concrete

⁵ `ModelElement` is an abstract class defined in the UML metamodel (see Chapter 7).

subclass B of A, may be considered a property of an instance of B. This is dependent on any *well-formedness rules*⁶ imposed on B. For example, if `att` is a non-static attribute of an abstract subclass A of `ModelElement`; then, the value of `att` is a property of any instance of a given concrete subclass of A, if there is no well-formedness rule that prohibits this.

Given any model element e_1 in a UML model, a change related element e_2 of e_1 ($e_2 \neq e_1$) is a model element such that when e_2 is changed e_1 is also considered changed. The notion of change related element is formally defined in Definition 1.

Definition 1: Change Related Element

Let E be the set of all model elements in any UML model object diagram. Given any two different model elements e_1 and e_2 ($e_1 \in E$ and $e_2 \in E$ such that $e_1 \neq e_2$), e_2 is regarded as a change related element of e_1 if when e_2 is changed then e_1 is also considered changed.

In the context of the UML metamodel used in this work a change related element is a *part* of an aggregate/composite object. Therefore, for example, if element e_2 is a part of the composite object e_1 , then e_2 is a property of e_1 . Example 1 illustrates this concept.

Aggregation/composition relationships are *transitive* and *antisymmetric* [2]. Since these relationships are transitive, given any two model elements e_1 and e_2 in a UML model, if there exists a path of aggregation/composition links from e_1 to e_2 , then e_2 is a change related element of e_1 . In addition, since aggregation/composition relationships are antisymmetric, there can be no cycles in the directed paths of aggregation/composition

⁶ *Well-formedness rules* define the criteria for a well-formed model.

links [2] in a UML model (object diagram of the UML metamodel). Therefore, given any model element e in a UML model, e cannot be a change related element (property) of itself.

Example 1: Model Element Property

Figure 2 is an example UML metamodel, and Figure 3 is a corresponding object diagram (UML model). In Figure 2, a class (Class) is a classifier (Classifier) and it has a number of attributes (Attribute) and operations (Operation). Each of these elements has a name attribute, which is inherited from the ModelElement class. A class also has an isAbstract attribute and zero or more association ends (AssociationEnd). In Figure 3, the properties of class are operation and attribute plus the values of name and isAbstract. The properties of attribute (and operation) are the values of name and visibility. The properties of end are the values of name and multiplicity. Note that end is *not* a property of class since it is not a part of class.

Figure 2: Example metamodel illustrating the concept of property.

Figure 3: Example UML model corresponding to Figure 2.

4.1.1 Added/Deleted Model Element

Some properties of a model element uniquely identify it among the set of all elements instantiating a metamodel class. These key properties are termed *element identifiers*. If a change to an element identifier occurs then the element is considered deleted and a new element added. For example, an attribute is uniquely identified by its name within its class's namespace. Thus, a changed attribute name is regarded as the deletion of the original attribute and the addition of a new attribute. The use of these element identifiers is the way any impact analysis system can keep track of the identity of model elements across model versions.

An element is considered added/deleted if it exists in the changed/original model version and it does not exist in the original/changed model version. Definition 2 formally defines the concept of added/deleted model elements.

Definition 2: Added/Deleted Model Element

Let E be the set of all model elements in the original model version and let E' be the set of all model elements in the changed version. Let e be any member of E , if

$e \notin E'$ then e is considered deleted. Let e' be any member of E' , if $e' \notin E$ then e' is considered added.

4.1.2 Changed Model Element

A model element is considered changed if any of its properties that do not uniquely identify it is changed. In addition, a model element e is considered changed if there exists a change related element of e that does not exist in both model versions. Note that only elements that exist in both model versions can be changed. Definition 3 and Definition 4 formally define the criteria for a changed model element.

Definition 3: Changed Model Element (Changed Property)

Let E be the set of all model elements in the original UML model version and let E' be the set of all model elements in the changed version. Let e be any member of $E \cap E'$. Let P be the set of properties of e in the original version and let P' be the set of properties of e in the changed version. Let $P_{ei} \subset (P \cap P')$ be the set of element identifiers of e . If any property $p \in ((P \cap P') - P_{ei})$ is changed then e is changed.

Definition 4: Changed Model Element (Added/Deleted Change Related Element)

Let E be the set of all model elements in the original UML model version and let E' be the set of all model elements in the changed version. Let e be any member of $E \cap E'$. Let C be the set of change related elements of e in the original version,

and let C' be the set of change related elements of e in the changed version. If there exists $c \in C \cup C'$ such that $c \notin C \cap C'$ then e is changed.

Example 2: Changed Model Element

The example in Figure 4, which is a modified version of Figure 3, shows four changes to class. The value of `isAbstract` was changed from `false` to `true`. The value of `visibility` in operation was changed from `#public` to `#protected`. Since operation is a property of class, then class is considered changed. An attribute is uniquely identified by its name and since the name of attribute was changed then the original attribute was considered deleted and a new one added. There is only one change to end – the change in the value of its multiplicity attribute. Note that the change to end does not affect class since end is not a property of class.

Figure 4: Example modified version of Figure 3.

4.2 UML Metamodel

OMG (Object Management Group) has defined a standard UML metamodel [1], which is represented in the UML notation. An adaptation of this official metamodel was used in this work. Among some of the changes made was the inclusion of model

constraints, such as preconditions and postconditions, as well as class and state invariants. Other changes made, included the simplification of the metamodel to include only the features of interest in this work, without sacrificing extendibility. For instance, only three model views (static, interaction, and state machine) are included in the modified metamodel. However, the core of the modified metamodel is very similar to that of the official version. This facilitates the inclusion of additional views as well as the addition of other features omitted from the views. Chapter 7 contains more details on the adapted metamodel.

4.3 Change Taxonomy

A change taxonomy is a classification of the possible changes to a UML model. A change taxonomy diagram should describe clearly and precisely how a UML model is changed, as change detection criteria are derived from it.

The change taxonomy derived in this work is shown in Figure 46 to Figure 57 and it contains 97 leaf changes.

4.3.1 Change Taxonomy Model

A change taxonomy is modeled as a *directed rooted tree*, every node (vertex) of which is a *change node*. The root identifies the composite `UMLModel`⁷ class in the UML metamodel and indicates that a UML model may be changed. Any given non-leaf (internal) node n in the tree identifies a subclass C of `ModelElement`. If C is abstract then n indicates that instances of concrete subclasses of C may be changed from one

⁷ Models the UML model in the metamodel (see Chapter 7).

model version to another. Otherwise, if C is concrete then n indicates that instances of C may be changed from one model version to another. A metamodel class is identified (in a taxonomy change node) by its class name or an association end role name. A class may appear more than once in the taxonomy but in different contexts, which are differentiated by role names. That is, a class may appear in several change nodes, each of which uses a different identity (a role name or the class name) of the class.

Each leaf l in the taxonomy tree identifies a subclass C of `ModelElement`, or a non-static attribute of a subclass of `ModelElement`. If C is abstract then l indicates that instances of concrete subclasses of C may be added/deleted to/from a UML model version. Otherwise, if C is concrete then n indicates that instances of C may be added/deleted to/from a model version. The appearance of an attribute in a leaf indicates that the value of the attribute may be changed from one version to another. Note that for simplicity sake a non-static attribute (of a subclass of `ModelElement`) is assumed to have a multiplicity⁸ of “1 . . 1”. A leaf in the taxonomy is referred to as a *leaf change*.

The children of the root of a change taxonomy tree identify the subclasses of `ModelElement` that are part of the composite `UMLModel` object. That is, each child of the root identifies a class C such that C is a part of the composite `UMLModel` class in the metamodel. In addition, each child of a given node n of the taxonomy tree identifies either a class C , such that the class identified in n has an aggregation/composition relationship with C , or an attribute of the class identified in n .

⁸ The *multiplicity* of an attribute refers to the number of possible values of the attribute [1]. For example, if the attribute has a multiplicity of “1 . . 1” then the attribute may have only one value in a given instance. As opposed to say, an attribute `att` (e.g. a list) that has a multiplicity of “0 . . 2”, then `att` may have zero, one or two values in a given instance.

A label present on an edge (arc) in a taxonomy tree states the number (multiplicity) of model elements or attribute values (of the class or attribute identified in the target node) that may be added, changed, or deleted. Given an edge (u, v) in a taxonomy tree, such that u (parent) identifies a class A, and v (child) identifies a class B; then, the upper range of the multiplicity labelled on (u, v) corresponds to the upper range of the multiplicity at the target end of the corresponding aggregation/composition relationship (in the metamodel) between A and B. The lower range of the multiplicity labelled on (u, v) is always “1”, signifying that A can be changed by adding/changing/deleting at least one instance of B. For example, if there exists a composition relationship between two subclasses A and B of `ModelElement` and the multiplicity at the target end B is “0..*”, then, the multiplicity labelled on the corresponding edge in the taxonomy is “1..*”. However, given an edge (u, v) in a taxonomy tree, such that u identifies a class, and v identifies an attribute; then, the multiplicity labelled on (u, v) is always “1”, in keeping with the assumption stated above.

Together, the children of the root describe how a UML model may be changed from one version to another. Collectively, the children of a given node n (except the root) describe how an instance of the class identified in n may be changed from one model version to another. For example, given a change taxonomy node u , with non-leaf children v and w , and a leaf child x . If the multiplicity on each of the edges (u, v) , (u, w) , and (u, x) is “1” and, if u identifies class A, v identifies class B, w identifies class C, and x identifies attribute `att`. Then, the taxonomy states that an A model element is changed by changing at least one of: its B model element, its C model element, or the value of its `att`

attribute. That is, an A model element is changed by changing at least one of its properties.

A change taxonomy is acyclic because, as said in Section 4.1, an element cannot be a property (change related element) of itself. Note that a tree is a connected acyclic graph [3]. Each edge direction shows the one-way change relationship between a model element and its property. That is, a change in the property results in a change to the element, not the opposite.

4.3.2 Pictorial Representation of the Change Taxonomy

The UML notation was used to pictorially represent the change taxonomy. In this representation, a node is modeled as a class and an edge is modeled as a unidirectional composition relationship directed at the element property. The composition relationship signifies that a model element change is composed of changes to its properties. A class (that models a node) name is a combination of a string stating whether the class or attribute identified in the node was added, changed, or deleted, and the identity of the class or the name of the attribute identified in the node. For example, if class B is identified in node n and n indicates that B may be changed; then, the class name of n is “Changed B”.

4.3.3 Change Taxonomy Derivation

The following steps are used to derive a change taxonomy from a given UML metamodel, and Example 3 illustrates how they are applied.

1. Identify the model elements as well as their properties.

2. The UMLModel class is identified in the root.
3. The attributes of the model element classes are identified in leaf nodes.
4. Model elements that can be added/deleted from the model are identified in leaf nodes. This is determined by analyzing the corresponding multiplicities in the UML metamodel.
5. Model elements that may be changed are identified in internal (non-leaf) nodes, as explained in Section 4.3.1 above.
6. Determine the multiplicities of the edges in the tree based on the description in Section 4.3.1 above.

Example 3: Deriving a Change Taxonomy from a given UML Metamodel

The taxonomy in Figure 6 is derived from the diagram in Figure 5 as follows:

1. The concrete subclasses of ModelElement are: CallAction, Message, SequenceDiagramView, ClassifierRole. An action model element is the only property of a Message model element. The value of recurrence is the only property of an action model element.
2. UMLModel is identified in the root.
3. recurrence is the only attribute of a model element class in the metamodel.
4. ClassifierRole and Message model elements are the only ones that may be added/deleted because their corresponding multiplicity on the class diagram is “*”⁹.

⁹ A multiplicity of “*” has an implicit lower range of “0”.

5. The classes in the internal nodes are: SequenceDiagramView, Message, and action. Here, the action role name for CallAction is used.
6. The multiplicities of the edges are shown on the tree in Figure 6.

In Figure 6, the taxonomy root, labelled “Changed UMLModel”, indicates that the UML model may be changed. The UML model is changed by changing its SequenceDiagramView model element. A SequenceDiagramView model element is changed by adding/deleting one or more ClassifierRole model elements, and/or adding/deleting one or more Message model elements, and/or changing one or more Message model elements. A Message model element is changed by changing its action property, which is changed by changing its recurrence attribute value. The IACMTool class models the main class of the software tool developed in this work (see Chapter 7).

Figure 5: Example UML metamodel – change taxonomy example.

Figure 6: Example change taxonomy – derived from Figure 5.

4.4 Change Detection Criteria

Each leaf change of the change taxonomy corresponds to one change detection rule (criterion). For a given metamodel (class diagram), this rule specifies how to determine if such a change has been made to a UML model (object diagram). Each rule consists of a change title, an acronym for the change title, a short textual description of the change, and an OCL (Object Constraint Language) expression that defines how the change is detected. The change title consists of the class name of the taxonomy leaf node, preceded by that of its parent. Each OCL expression is defined on the UML metamodel and returns a set¹⁰ of added, changed, or deleted model elements.

¹⁰ Set here refers to the mathematical set, i.e., a set contains no duplicates [4].

Ninety-Seven (97) rules, corresponding to 97 leaf changes, are defined in this work and are presented in Appendix A. The examples in Figure 7, Figure 8, and Figure 9 illustrate how to detect the set of added, changed, and deleted messages, respectively, of the SequenceDiagramView model element in the UML model. The example diagram in Figure 5 is navigated by the OCL expressions in these figures. The changed version is the version context of the SequenceDiagramView class in the OCL expressions. That is, the change detection rules are defined on the changed version of the UML model. The equals operation used in Figure 8 determines if two recurrence values are equal. The getIDStr() operation in the same figure returns a string (element identifier) that uniquely identifies a Message model element in the model.

Changed SequenceDiagramView - Added Message	
Change Code:	CSDVAM
Description:	In the changed version there exists a set of messages that does not exist in the original version.
OCL Expression:	<pre> context SequenceDiagramView self.message->select(m:Message not self.model.application. originalModel.sequenceDiagramView.message->includes(m)) </pre>

Figure 7: Example Change Detection Rule – Added Model Elements.

Changed Message action - Changed recurrence	
Change Code:	CMaCr
Description:	There exists a set of messages in the model such that for each of these messages, the recurrence property of its action is not the same in the two model versions.
OCL Expression:	<pre> context SequenceDiagramView self.message->select(m1:Message not m1.action.recurrence. equals(self.model.application.originalModel.sequenceDiagramView. message->select(m2:Message m1.getIDStr() = m2.getIDStr()). action.recurrence) </pre>

Figure 8: Example Change Detection Rule – Changed Model Elements.

Changed SequenceDiagramView - Deleted Message**Change Code:** CSDVDM**Description:** In the original version there exists a set of messages that does not exist in the changed version.**OCL Expression:**

```
context SequenceDiagramView
self.model.application.originalModel.sequenceDiagramView.
message->select(m:Message|not self.message->includes(m))
```

Figure 9: *Example Change Detection Rule – Deleted Model Elements.*

Chapter 5 – Impact Analysis

Impact analysis is the identification of the potential consequences (side-effects) of a change, or estimating what needs to be modified to accomplish a change [7]. In the context of UML models, impact analysis is the identification of the model elements *impacted* by a model change. A model element is considered impacted by a model change if a modification to that element or its implementation (in the source code) *may* be needed to accomplish the change – this cannot always be determined with certainty. A model change here refers to a leaf change in the change taxonomy, and as discussed in Chapter 4, a model element is an instance of a concrete subclass of `ModelElement`.

Like many identification methods, impact analysis methods are based on search algorithms and can be classified as follows [7], [8].

1. *Semantically Guided*: Impact analysis is directed by predetermined semantics of objects and relationships derived from semantic network algorithms.
2. *Heuristically Guided*: Impact identification is directed by predetermined rules or heuristics to suggest possible paths to examine or dubious ones to avoid. Such methods may use a rule base of *system-change experience*, for example.
3. *Stochastically Guided*: Impact determination is guided by probabilities derived for the situation.
4. *Hybrid Guidance*: Impact identification uses a combination of the above methods, for example, transitive closure algorithms coupled with probabilities of impact.

5. *Unguided/Exhaustive*: Impact identification is conducted in a brute-force manner using simple traceability relationships in the repository.

The technique used in this work fits best in category 2 (heuristically guided) and employs a predetermined set of impact analysis rules based on system-change experience [7]. This system-change experience refers to the change taxonomy defined in this work. The advantage of this rule-based technique is that it states precisely how to determine the impacted elements for a specific change. Therefore, for example, the user of the impact analysis tool may see the effect of each change and thus be better able to decide which ones should be implemented and when.

One impact analysis rule is defined for each leaf change in the taxonomy, and each rule specifies how to derive the bag(s)¹¹ of impacted elements. There is one bag for each type of impacted elements. Bags are identified because an element may be impacted in several ways by the same model change.

Given any impacted element i , there must exist a navigable path, in the UML model object diagram, from the changed element to i . In addition, only elements that exist in both model versions can be impacted. Definition 5 formalizes the concept of impacted model elements. Further, given any impacted element i , if i is a change related element of a model element e , then e is also impacted. For example, an operation is part of a class, therefore, if an operation is impacted, then the class to which it belongs is also impacted. This is formalized in Definition 6.

¹¹ A bag is a collection in which duplicates are allowed [4].

Definition 5: Impacted Model Elements

Let E be the set of all model elements in the original model version and let E' be the set of all model elements in the changed version. Let $e \in E \cap E'$ be a model element that is changed resulting in a bag $I \subset (E \cap E')$, of impacted elements. Then, $\forall i \in I, e \neq i$ and there exist a navigation path from e to i in the object diagram corresponding to the changed version.

Definition 6: Impacted Change Related Elements

Let E be the set of all model elements in the original model version and let E' be the set of all model elements in the changed version. Let $c \in E \cap E'$ be any change related element of e such that $e \in E \cap E'$. Thus, if c is impacted, then e is also impacted.

Any impact analysis technique must seek to reduce the number of false-positives (over-estimates) and find more impacts that are likely to be *actual impacts*¹². With that consideration, one of the main objectives of this work is to be as precise as possible when identifying impacts. Therefore, based on the information available for a particular leaf change, a rule should identify only the elements that are likely to be impacted while not discounting other possible impacts. For example, if it is possible that an operation o is impacted, then the strategy used here will identify o instead of simply saying that the class to which o belongs is impacted without being specific.

¹² An actual impact affects the model element and/or its implementation (in the source code).

Each leaf change of the taxonomy was analyzed and the possible effects from such a change considered. The fact that the model versions are assumed consistent was also considered. For instance, if an operation is deleted from a class then there is no impact since the model is assumed consistent. That is, for example, there should be no invocation of the operation since it is no longer defined. Each of the impact analysis rules defined in this work is described in a structured and precise manner so that it is easy to review and refine them, for example, as the UML standard is evolving. Table 1 describes the structure of an impact analysis rule. Note that a rule can *invoke* other rules; this is the way transitive closure is modeled in this work. That is, this is how indirect impacts are determined from a given impact. Each invoked rule is a member of the set of defined impact analysis rules. In addition, note that undefined changes may be needed to accomplish a given change, for example, a change to the implementation of a class.

In total, ninety-seven (97) impact analysis rules have been defined, each of which identifies one or more bags of potentially impacted elements. Note that a few of the rules do not identify any impacted elements since it is certain that no resulting changes to any other elements are necessary to accomplish the changes on which these rules are defined. The rules are presented in Appendix B.

To derive an impact analysis rule for a particular leaf change, knowledge of UML is combined with the information represented in the metamodel to identify paths, in an instance of the metamodel, from the changed element to the impacted ones. Example 4, illustrates how an impact analysis rule is derived for a leaf change, from a given metamodel. The example also explains how to interpret the information presented in a rule. Example 5 illustrates the application of an impact analysis rule.

Table 1: Structure of Impact Analysis Rules

Field	Description
Change Code	An acronym for the change title.
Changed Element	Pathname of the changed model element class.
Added/Changed/Deleted Property	The added, changed, or deleted property.
Impacted Element(s)	Pathname of the impacted element(s).
Description	Briefly states the elements impacted and under what conditions.
Rationale	Explains the reason(s) for the impacts.
Resulting Change(s)	States the possible undefined resulting change(s).
Invoked Rule(s)	Lists the rule(s) invoked by the current rule. That is, the transitive closure for a particular change.
OCL Expression(s)	OCL expression(s) describing the formal derivation(s) of impacted elements based on the metamodel. There is one OCL expression for each type of impacted element.

Example 4: Impact Analysis Rule

Figure 10 is an excerpt of the UML model of the tool (IACMTool) developed in this work. In this figure, a model change is modeled by the Change class. The `propertyID` attribute stores a string that identifies the changed property. As mentioned previously, the `getIDStr()` operation in the Message class returns a string that uniquely identifies the message among the set of messages in the model.

The example impact analysis rule in Figure 11 refers to Figure 10. In Figure 11, the change title is first presented, followed by the change code and the class name of the

changed element (SequenceDiagramView). This is followed by the class name of the added property (Message). The class names of the impacted elements are shown next; in this case three types of elements are impacted, namely: ClassClassifier, Operation, and Postcondition. The description field of the rule states under what conditions these elements are impacted, while the rationale explains the reasons for the impacts. The resulting changes are then stated followed by the invoked rule, which models the transitive closure. The resulting changes are followed by OCL expressions describing how to identify the impacted elements.

Figure 10: *Excerpt from IACMTool – impact analysis example.*

In Figure 11, the base class of the classifier role that sends the added message is impacted because it now performs one more action. This action is performed in an operation, which may or may not be represented on the added message, for example if the

added message is a signal. If the added message invokes an operation, then that operation is impacted, and its postcondition is impacted as well. Note that the defined operation¹³ (class operation) is identified instead of the invoked operation (on a message). This is in keeping with one of the objectives of being as precise as possible – a method (implementation) implements a defined operation not the invoked one. The postcondition is impacted because the effect (what is true upon completion) of the operation has changed – even if the postcondition is empty, it may be impacted because its impact says that the effect of the operation may change.

In Figure 11, no further rules are invoked on the impacted class because no leaf change has been identified for it – the implementation is changed and no leaf change is defined for it since such a property does not exist in the metamodel. Similarly, for the same reason, no further rules are invoked for the impacted operation. However, a changed postcondition is a leaf change and thus the impact analysis rule for this change is invoked.

Also in Figure 11, each OCL expression starts with the **context** keyword. The first expression defines three placeholders (variables) that represent navigation expressions, and are repeatedly used in the expressions for the impacted elements. The definition of each of these placeholders is preceded by the **let** keyword. The first placeholder defines an empty set of model elements. The second placeholder identifies the added message, which is a message such that its unique identification (determined by `getIDStr()` string matches `propertyID`. The last placeholder identifies the sending operation, which is a defined operation that is *equal* to the invoked operation of

¹³ As in the official OMG metamodel [1], a defined operation is modeled by the same class as an invoked operation.

the added message. This equivalency check is done by the equals() operation that identifies the defined operation of a class, given its name and actual parameters.

Changed SequenceDiagramView - Added Message	
Change Code:	CSDVAM
Changed Element:	SequenceDiagramView
Added Property:	Message
Impacted Elements:	ClassClassifier, Operation, Postcondition
Description:	The base class of the classifier role that sends the added message is impacted. The operation that sends the added message is impacted, and its postcondition is also impacted.
Rationale:	The sending/source class now sends a new message and one of its operations, actually sending the added message, is impacted. This operation is known or not, depending on whether the message triggering the added message corresponds to an invoked operation. If, for example, it is a signal then the operation may not be known just by looking at the sequence diagram. The impacted postcondition may now not represent the effect (what is true on completion) of its operation.
Resulting Changes:	The implementation of the base class may have to be modified. The method (implementation) of the impacted operation may have to be modified. The impacted postcondition should be checked to ensure that it is still valid.
Invoked Rule:	Changed Class Operation - Changed Postcondition (CCOCpst)
OCL Expressions:	<pre> context Change def: let null:ModelElement = Set() let addedMessage:Message = self.changedElement. oclAsType(SequenceDiagramView).message->select(m:Message m.getIDStr() = self.propertyID) let sendingOperation:Operation = (if addedMessage.activator.action.ocIsTypeOf(CallAction) then addedMessage.sender.base.ocIsType(ClassClassifier). operation->select(o:Operation o.equals(addedMessage. activator.action.ocIsType(CallAction).operation)) else null endif) context Change -- class if not addedMessage.activator.ocIsTypeOf(CallAction) then addedMessage.sender.base.ocIsType(ClassClassifier) else null endif context Change -- operation sendingOperation context Change -- postcondition sendingOperation.postcondition </pre>

Figure 11: Example impact analysis rule - CSDVAM.

Note that the sending ClassifierRole is not identified in this example because its base class is more specific and therefore it is more useful to identify the base class. Similarly, the action model element of the added message is not identified because its operation is more specific.

Example 5: Application of Impact Analysis Rule

The application of the CSDVAM rule presented in Figure 11 is illustrated here. In this example, an excerpt from the Automated Teller Machine (ATM) case study (Chapter 8) is used. A message (and the corresponding classifier role) was added to the sequence diagram shown in Figure 13. The resulting sequence diagram is shown in Figure 14, and the corresponding class diagram is shown in Figure 12. In Figure 13, when Customer presses cancel on the ATM keyboard (Keyboard) the cancelPressed operation is invoked on ATM. In Figure 14, cancelPressed invokes the display operation on Display.

The rule in Figure 11 states that the base class of the classifier role that sends the added message is impacted. Thus, as shown in Figure 12 and Figure 14, ATM is impacted since it is the base class of atm. Furthermore, the rule also states that the operation (and its postcondition) that sends the added message is impacted. Therefore, both cancelPressed and its postcondition are impacted. Note that ATM is impacted twice: (1) because of the reason above, and (2) because cancelPressed is a change related element (see Definition 6) of it. Now, the CSDVAM rule invokes the CCOCpst rule (Figure 15), which states that the operations (and their postconditions) that invoke the changed operation (cancelPressed) are impacted. In this case, as shown in Figure 12

and Figure 14, both `cancel` and its postcondition are impacted. In addition, `Keypad` is also impacted since `cancel` is a change related element of it. Note that the impacts to `Keypad`, `cancel` and its postcondition are indirect impacts while those to `ATM`, `cancelPressed` and its postcondition are direct impacts. The `getInvokingOperations` operation in Figure 15 returns the operations that invoke a given operation in the sequence and statechart diagrams of the model. The CCOCpst impact analysis rule is defined on the UML model in Figure 10.

Figure 12: Class diagram for impact analysis example.

Figure 13: Original sequence diagram for impact analysis example.

Figure 14: *Changed sequence diagram for impact analysis example.*

Changed Class Operation - Changed postcondition	
Change Code:	CCOCpst
Changed Element:	Operation
Changed Property:	Postcondition
Impacted Elements:	Operation, Postcondition
Description:	The operations that invoke the changed operation are impacted. The postconditions of these impacted operations are also impacted.
Rationale:	The effect of each impacted operation may be changed and thus each impacted postcondition may have to be changed.
Resulting Changes:	The method (implementation) of each impacted operation may have to be changed. Each impacted postcondition should be checked to ensure that it now reflects the effect of its operation.
Invoked Rule:	Changed Class Operation - Changed Postcondition (CCOCpst)
OCL Expressions:	<pre> context Change -- operation self.changedElement.oclaType(Operation). getInvokingOperations() context Change -- postcondition self.changedElement.oclaType(Operation). getInvokingOperations().postcondition </pre>

Figure 15: *Example impact analysis rule – CCOCpst.*

Chapter 6 – Distance Measure

When impacts between model elements are indirect, following the general guidelines in [8], the suggestion here is to use a distance measure between a changed model element and the impacted ones. In [8], it is stated that a common assumption¹⁴ is that “if direct impacts have a high potential for being true, then those farther away will be less likely”. Even with a carefully designed set of impact analysis rules and change taxonomy, the number of impacts may be very large. Using a distance measure to filter/order impacts is therefore often necessary in practice. The main related question then becomes: How to define such a distance measure?

Recall that impact analysis rules identify impacted elements and then, in some cases, a number of impact analysis rules are invoked on some of the directly impacted elements. The distance between a changed element and a given impacted element is defined as the number of impact analysis rules that is invoked to identify this impacted element. In Figure 16, for example, each set of impacted elements can be represented as a node in a tree, the arcs (edges) of which are impact analysis invocation rules. Reusing here the example rule from Figure 11 (CSDVAM – adding a message to a sequence diagram), this rule triggers another rule (CCOCp_{st}). As shown in Figure 16, for the impacted postcondition p_1 , CCOCp_{st} (changed operation postcondition) triggers CCOCp_{st}, thus leading to other impacted postconditions and operations. Only nodes of up to a depth of two are shown. The distance of a given impacted element in the tree corresponds to the depth of its node in the tree; for example, the distance of p_2 is 2. Such

¹⁴ This fundamental assumption seems reasonable, but empirical investigations are warranted to validate it.

a distance measure may then be used to either, sort impacts according to their distance from a given changed element, or to even exclude impacted elements farther than a certain distance, which may be a configurable parameter of the tool. If a model element is impacted several times, then the minimum distance can be used (that is, the strongest impact).

Figure 16: Example of distance between a changed element and the impacted ones.

Chapter 7 – Impact Analysis Tool

The Impact Analysis and Change Management tool (IACMTool) developed in this work reads two UML model versions and produces an impact analysis report as well as a consistency verification report. After a model version is read, its consistency is verified. After both versions have been read and checked for internal consistencies, change detection is done to identify all the changes between the two model versions, and classify them according to the taxonomy defined in Appendix A. These changes are then used to perform impact analysis on the model using the impact analysis rule pertaining to each leaf change.

There are seven main packages in the tool, namely: `control`, `umlModel`, `parser`, `modelChanges`, `impactAnalysis`, `reportGeneration`, and `consistencyVerification`. The subsystem decomposition is shown in Figure 17, with the packages and dependencies among them. In particular, the packages contain 99 classes, 68 of which are in the `umlModel` package (the UML metamodel), and the implementation consists of 9,429 lines of Java source code (excluding comments). Each section below provides a synopsis of one of these packages.

7.1 `control` Subsystem

The `control` subsystem, shown in Figure 18, is responsible for the overall control flow of the application. There are nine classes in this subsystem, seven of which are responsible for implementing a state machine that controls the sequencing of activities in the application. The `IACMTool` class is the main class in the tool and

references the main class of each of the other six subsystems in the tool. The `ApplicationInterface` class implements the (graphical) user interface. The state machine control logic is implemented using the state pattern [9]. `ApplicationController` is the context of the state machine, and `ApplicationControllerState` is the abstract state. The concrete states are: `InitialState`, `LoadingOriginalModelState`, `LoadingModelState`, `DetectingChangesState`, and `AnalyzingImpactsState`.

Figure 17: *iACMTool* subsystem decomposition.

The tool facilitates the saving of the reports generated. These reports include change detection, consistency verification and impact analysis reports. The sequence of

steps required to perform impact analysis, given two versions of a UML model, is listed below.

1. Load the original model version
2. Verify the consistency of the original model version
3. Load the changed model version
4. Verify the consistency of the changed model version
5. Detect the changes between the two model versions
6. Analyze the impacts
7. Generate the reports

Figure 18: control *subsystem*.

7.2 `umlModel` Subsystem

The `umlModel` subsystem is a UML metamodel that was adapted from the official OMG standard [1] for the purposes of this work. The adaptation includes the inclusion of only the model views and features that are of interest in this work. Also included in the adaptation is the addition of model elements to represent model constraints, such as preconditions, postconditions, as well as class and state invariants. UML version 1.4 [1] is modeled in the `umlModel` subsystem.

The model views included in this work are: class diagram (static), sequence diagram (interaction), and statechart diagram. The metamodel in this work is designed to facilitate easy upgrading. This is achieved by keeping the core of the metamodel as consistent as possible with the standard. Therefore, additional views such as activity and use case can be easily included. Note that the use case package has already been defined and only one of its classes, namely `Actor`, is specified. This was done because the base of some classifier roles is an actor, so this has to be modeled.

The `umlModel` subsystem has three packages, namely: `foundation`, `behaviouralElements`, and `modelManagement`. There are three classes in the `umlModel` package, namely: `UMLModel`, `ModelView`, and `ModelElement`. Figure 19 shows the packages, classes, and their relationships in `umlModel`. The `IACMTool` class models the application and as shown in the figure, has two model versions, an original as well as a changed version. Each model version is composed of three views plus the model elements. The model elements are indexed by the `xmiID` qualifier, which is used by the parser subsystem to build the models.

Figure 19: *umlModel subsystem.*

7.2.1 foundation Package

The foundation package is the language infrastructure that specifies the static structure of models and is decomposed into the following sub-packages: `core`, `extensionMechanisms`, and `dataTypes` [1]. Figure 20 shows the foundation package. The `core` package specifies the basic concepts required for an elementary metamodel and defines an architectural backbone for attaching additional language constructs, such as meta-classes, meta-associations, and meta-attributes [1]. The `extensionMechanisms` package specifies how model elements are customized and extended with new semantics [1]. The `dataTypes` package defines basic data structures for the language [1].

Figure 20: foundation *package*.

7.2.1.1 core Package

The following description of the core package is taken from [1]: The core package is the most fundamental of the foundation sub-packages. It defines the basic abstract and concrete metamodel constructs needed for the development of object models. Abstract constructs are not instantiable and are commonly used to reify key constructs, share structure, and organize the UML metamodel. Concrete metamodel constructs are instantiable and reflect the modeling constructs used by object modelers. Abstract constructs defined in the core include GeneralizableElement, Relationship, and Classifier. Concrete constructs specified in the core include ClassClassifier, Attribute, Operation, and Association.

The main class diagram for the core package is shown in Figure 21, and it presents some of the basic constructs in this package. In this diagram, the abstract constructs are: NamespaceOwner, GeneralizableElement, Classifier, Feature, StructuralFeature, and BehaviouralFeature. While the concrete

constructs are: Parameter, Constraint, Attribute, and Operation. A NamespaceOwner element owns a namespace and is itself owned by another NamespaceOwner element. It is also a generalizable element (GeneralizableElement), which means it may participate in a generalization relationship. Each element in a namespace is uniquely identified by its name. A Classifier element is a NamespaceOwner element that describes behavioural and structural Feature elements [1]. A Feature element is a property that is encapsulated within a Classifier element [1]. A BehaviouralFeature element is a dynamic feature and a StructuralFeature element is a static feature. A parameter (Parameter element) is an unbound variable that can be changed, passed, or returned [1]. An attribute is modeled by the Attribute class, while an operation is modeled by the Operation class. The Constraint class models semantic conditions or restrictions [1].

Figure 21: foundation::core – main class diagram.

The class diagram in Figure 22 shows the classifiers and invariants of the metamodel. Here, there are three concrete classifier constructs, namely: Interface, ClassClassifier, and DataType. An interface is modeled by Interface, a class is modeled by ClassClassifier, and a data type is modeled by DataType. A class may have zero or more inner (owned) classes. Every class is considered to have a class invariant, and an invariant may access a number of attributes. Preconditions and postconditions are operation contracts, which are invariants. Each operation contract

accesses a number of parameters. Note that an invariant may be empty but not absent. That is, for example, an operation always has a precondition and a postcondition, while a class always has a class invariant – these constraints may be empty.

Figure 22: foundation::core – classifiers class diagram.

Figure 23 presents the relationships class diagram for the core package. This diagram describes the static relationships represented in the metamodel. Four types of relationships are presented in this diagram, they are: association (Association), generalization (Generalization), realization (Realization), and dependency (Dependency). Association, realization, dependency all inherit from the

Relationship class, while association is a generalizable element. A dependency relationship has a client and a supplier classifier. A generalization relationship has a parent and a child classifier. A class implements an interface through a realization relationship. An association relationship has two association ends (AssociationEnd). Each end has a participant; and each end may have zero or more interface specifiers (interfaceSpecifier). An association end may also have zero or more qualifier attributes. In addition to the two ends, an association relationship may also have an association class (AssociationClass).

Figure 23: `foundation::core-relationships` class diagram.

The class diagram (static) view (of the UML model) used in this work is shown in Figure 24. The `ClassDiagramView` class models this view and the diagram shows

that the class diagram view consists of a number of classes, associations, interfaces, realizations, dependencies, and generalizations.

Figure 24: foundation::core – class diagram view.

7.2.1.2 dataTypes Package

The dataTypes sub-package of the foundation package is shown in Figure 25. Here, seven enumeration classes are shown, corresponding to seven types of enumerated metamodel attribute values, these are namely: AggregationKind, ScopeKind, CallConcurrencyKind, OrderingKind, VisibilityKind, ChangeabilityKind, and ParameterDirectionKind. Multiplicity is modeled by a class (Multiplicity) to specifically identify the upper and lower ranges of the multiplicity – this helps in providing precise change detection. A metamodel expression is model by the Expression class, which has a body as well as a language attribute. In the context of this work, OCL is the default language. Three types of expressions are modeled in this metamodel, and are captured by the

IterationExpression, ActionExpression, and BooleanExpression classes.

Figure 25: foundation::dataTypes package.

7.2.1.3 extensionMechanisms Package

The extensionMechanisms sub-package of the foundation package is shown in Figure 26. Here, every model element has zero or more constraints or a model element may have a stereotype. In addition, every model element has zero or more tagged values. This package has been implemented, however, change detection and impact analysis has not been done on it – this is left for future work.

Figure 26: foundation::extensionMechanisms package.

7.2.2 behaviouralElements Package

The behaviouralElements package, shown in Figure 27, consists of four sub-packages, namely: commonBehaviour, collaborations, useCases, and stateMachines. commonBehaviour specifies the core concepts required for behavioural elements [1]. The collaboration package specifies a behavioural context for using model elements to accomplish a particular task [1]. The useCases package specifies behaviour using actors and use cases [1]. The stateMachines package defines behaviour using finite-state transition systems [1].

Figure 27: behaviouralElements *package*.

7.2.2.1 commonBehaviour Package

The commonBehaviour package is the most fundamental of the sub-packages that compose the behaviouralElements package [1]. It specifies the core concepts required for dynamic elements and provides the infrastructure to support collaborations, state machines, and use cases [1]. An action (Action) is a specification of an executable statement that forms an abstraction of a computational procedure that results in a change in the state of the model [1]. An action can be realized by sending a message to an object, or by modifying a link or a value of an attribute [1]. An action may contain a specification of the actual arguments (ActualArgument). That is, a sequence of arguments containing expressions for determining the actual instances to be used when the action is performed (or executed). An ActionSequence is composed of a sequence of (discrete) actions. Two types of actions are defined in this work, namely CallAction and UndefinedAction. A call action (CallAction) invokes an operation on an object, while an undefined action (UndefinedAction) does not

explicitly specify an invoked operation. The UndefinedAction class models all other types of actions, such as signal action, create action, and destroy action (see [1], [2]).

Figure 28: behaviouralElements::commonBehaviour package.

7.2.2.2 collaborations Package

Figure 29 shows a class diagram of the collaborations package. In this diagram, a collaboration is composed of one or more classifier roles (ClassifierRole), which may send and/or receive a number of messages (Message). A classifier role has one base classifier; in the context of this work, a base is either a class or an actor. A classifier role may also have a number of available operations, which is a

subset of the operations available in the base classifier. A message may have an activator, which is the message that invokes it. A message may also have zero or more predecessors and successors.

Figure 29: `behaviouralElements::collaborations – roles`.

Figure 30 shows another class diagram of the collaborations package. In this diagram, a collaboration is also composed of a number of interactions (Interaction), which is composed of a set of partially ordered messages [1] (Message). A collaboration represents an operation or a classifier. A collaboration may also use a number of other collaborations.

Figure 30: `behaviouralElements::collaborations` – *interaction*.

The sequence diagram view of the model is shown in Figure 31. The `SequenceDiagramView` class models the sequence diagram view, which is made up of a number of classifier roles and messages.

Figure 31: *Sequence diagram view class diagram.*

7.2.2.3 stateMachines Package

Figure 32 shows the main class diagram for the stateMachines package. A state machine (StateMachine) defines the dynamic behaviour of an object. Each state machine has one top state and a number of transitions. Each transition (Transition) has a source state vertex (StateVertex) and a target state vertex. A transition may also have a guard condition, an effect, and a trigger event. A state may have an exit action, as well as an entry action and an activity (doActivity), which is repeatedly performed while the object is in that state. A composite state (CompositeState) has a number of sub-vertices, a subset of which may also be composite states.

Figure 32: behaviouralElements::stateMachines – main.

Figure 33 shows the events class diagram of the stateMachines package. In this diagram, an event (Event) has a sequence of parameters, and only two types of events are defined in this work, namely CallEvent and UndefinedEvent. CallEvent models events in which an invoked operation is explicit, while UndefinedEvent models all other types of events, such as signal events (see [1], [2]).

Figure 33: behaviouralElements::stateMachines – events.

The state chart diagram view class diagram is shown in Figure 34. Here, StatechartDiagramView models this view and is made up of a number of states and transitions.

Figure 34: Statechart diagram view class diagram.

7.2.3 modelManagement Package

The model management (modelManagement) package is shown in Figure 35. Here, a package contains a number of model elements via the ElementImport class, which handles model element packaging.

Figure 35: modelManagement package.

7.3 parser Subsystem

The parser subsystem is shown in Figure 36 and is responsible for parsing XMI (XML Metadata Interchange [10], [11]) and text files. The XMI files describe UML models and the text files are used for tool configuration purposes. XMI 1.1 for UML 1.4 (OMG) is used in this work. Note that this does not pose a great restriction on the tool's usage, because tools such as Together® [5] are able to read different XMI files for different UML versions and export XMI files that meet the requirements of the tool. The XMI parser in this tool was tailored to read the XMI format generated by Together® [5].

The `XMIParser` class uses the JAXP API (Java API for XML Processing) [13] and the DOM (Document Object Model) [14] specification to implement the XMI parser.

This is done by first processing the XML data to create a tree-based object-oriented hierarchical representation of the document. This tree structure is then parsed to build the UML model. The `XMIParserConstants` interface stores the constants used by the XMI parser.

Figure 36: *parser subsystem*.

7.4 consistencyVerification Subsystem

The `consistencyVerification` subsystem, shown in Figure 37, is responsible for performing consistency verification of UML models. `ConsistencyVerifier` is the main class and it identifies a number of inconsistencies (`Inconsistency`). Each inconsistency has an `id` that is mapped to a string identifying the type of inconsistency. Each inconsistency identifies a collection of model elements (`inconsistentElement`) that are responsible for that inconsistency.

The consistency rules are implemented in the `umlModel` subsystem. This along with the fact that there is only one method invocation from the

consistencyVerification subsystem to the umlModel subsystem, results in low coupling between the two subsystems. The ConsistencyVerifier class is responsible for invoking the consistency rules on the umlModel subsystem. The specific invocation order of these rules is handled by the UMLModel class. The invocation of the inconsistency rules is not hard-coded, instead the Method class in the Java API [21] is used to dynamically identify and invoke the methods that implement these rules. This was further facilitated by the adoption of a standardized naming scheme for these methods.

Figure 37: consistencyVerification *subsystem*.

7.5 modelChanges Subsystem

Each of the 97 change detection rules corresponds to one method in the implementation of the tool. Based on this large number of methods, a suitable solution

had to be found on where to implement these methods. One solution is to place all the methods in one class – thus this class would have more than 97 methods. This solution is undesirable, because of complexity concerns. In addition, if the tool is modified in the future to include more change detection rules then the complexity of this class would increase. Another issue is that, upon examination of the rules, coupling concerns would arise. Each rule accesses the metamodel elements often, thus there would be strong coupling between the `modelChanges` and `umlModel` subsystems.

Another solution is to implement each change detection rule of an element in the class that models it. Therefore, for example, all the changes to an `Operation` model element would be implemented in the `Operation` class. This solves the class complexity problem. The question then is: In what order should the methods be invoked? The change taxonomy defines a top-down approach to change detection, where the changes to the views are first determined, followed by the changes to their element properties, etc. The same approach was used in the implementation, by using recursion. The `ModelElement` class defines a `determineChanges` method that is responsible for two tasks: (1) invoking the change detection rules defined in a model element class, and (2) determining the changes to the element properties of the model element. The changes to the element properties of a model element are determined by invoking the `determineChanges` method on each of its element properties. All the changes to the model are detected by invoking the `determineChanges` method on each model view, and by recursion, the changes to each element (and their element properties) in that view are detected. This solution results in very low coupling between the `modelChanges` and `umlModel` subsystems – there is only one method invocation from

modelChanges to umlModel. Note that in this solution change detection is done precisely as defined in the taxonomy. The change detection algorithm is summarized in Figure 38.

Let E be the set of model elements in the model, and let $V \subset E$ be the set of model views in the model.

```
1. for each  $v \in V$  do
2.     DETERMINECHANGES( $v$ )

DETERMINECHANGES( $e$ :ModelElement)
1. invoke the change detection rules defined in  $e$ 's class
2. for each  $p \in E$ , such that  $p$  is an element property of  $e$  do
3.     DETERMINECHANGES( $p$ )
```

Figure 38: *Change detection algorithm.*

The modelChanges subsystem is shown in Figure 39. Here, the application has a change detector (ChangeDetector), which determines a set of model changes (Change) resulting from logical changes (LogicalChange). Each change has a code that corresponds to the change title acronym, and a propertyID that identifies the property that was changed. Each change also has a changed model element (changedElement).

Figure 39: modelChanges *subsystem*.

7.6 impactAnalysis Subsystem

The modelChanges subsystem produces a list of Change objects. Each of these objects has information about the model element changed as well as the change code and property changed. The responsibility of the impactAnalysis subsystem is to; using these Change objects, identify the impacted elements. As said before, impacted elements are identified using the impact analysis rules. There are 97 impact analysis rules corresponding to 97 methods in the tool's implementation. The impact analysis rules are implemented in the umlModel subsystem in a similar fashion to that used for the change detection rules. In addition, the impactAnalysis subsystem only has to invoke one

method (`determineImpacts`) in the `umlModel` package to determine the impacts for a particular change. Given a change, the `determineImpacts` method is responsible for invoking the correct impact analysis method. This is done by using a consistent naming scheme for these methods. The name of each impact analysis rule consists of the string “`determineImpacts`” plus the corresponding change code. The invocations of the impact analysis methods are not hard-coded – this would complicate the `determineImpacts` method in the `umlModel` class. Instead, the `Method` class in the Java API [21] is used to dynamically identify and invoke each method given its name and that of the class in which it is defined. This strategy reduces coupling between the `impactAnalysis` and `umlModel` subsystems. The impact analysis algorithm is briefly summarized in Figure 40.

<pre>for each $c_i \in C$, where C is the set of changes identified invoke m_i such that m_i is the method that implements the impact analysis rule corresponding to c_i.</pre>
--

Figure 40: *Summary of the impact analysis algorithm.*

Figure 41 shows the impact analysis subsystem. Here, `ImpactAnalyzer` is the main class in the subsystem and it determines a number of impacts (`Impact`). Each impact has a propagator that is used for transitive closure. Each `Impact` identifies a number of impacted elements, and each `Impact` is defined on a particular `Change`.

One of the main issues that arose in the implementation of this subsystem was the identification of overloaded operations in sequence diagrams. This was dealt with using the same strategy used in [12].

Figure 41: *impactAnalysis subsystem.*

7.7 reportGeneration Subsystem

Three types of reports are generated by the tool, namely: consistency verification, change detection, and impact analysis. Note that an impact analysis report includes a change detection report, however, a change detection report can be generated independently. The desired report is generated by invoking the appropriate operation on the class responsible for that type of report. The reportGeneration subsystem is shown in Figure 42. In this subsystem, consistency verification reports are handled by the ConsistencyVerificationReport class, change detection reports are handled by

the ChangeDetectionReport class, and impact analysis reports are handled by the ImpactAnalysisReport class.

Figure 42: reportGeneration *subsystem*.

Chapter 8 – Case Study

An Automated Teller Machine (ATM) system was selected for the case study. Here, logical changes were made to the ATM's model and impact analysis performed. The results were then used to determine how the cumulative numbers of impacted elements vary with the distances from the change. This provides information about the precision of the impact analysis rules.

Since the strategy used in this work is based on performing impact analysis on the model rather than on the source code, it is prudent to investigate the accuracy of the strategy, with respect to the implementation. This investigation was also done; it is outlined in Section 8.3 below. The case study is described in Section 8.1, and the results are discussed in Section 8.2. The conclusions are then proposed in Section 8.4.

8.1 Description

An ATM card is inserted by the customer who then enters the PIN (Personal Identification Number). After which, transactions such as deposit, inquiry, withdrawal and transfer can be performed. A receipt is then issued upon the completion of all the transactions in the session. The original version of the ATM's UML model (Appendix D) contains a class diagram consisting of 19 classes, and a use case diagram consisting of 15 use cases – each use case being associated with one sequence diagram. Most of the sequence diagrams contain between three and seven messages; for instance, sequence diagrams for the `ATMStartUp` and `ATMShutOff` use cases contain seven and three messages, respectively. The sequence diagram for the `Transaction` use case is the

most intricate one – it consists of 22 messages. There are 15 attributes and 18 operations in the class diagram, as well as four inheritance, eleven association, and three dependency relationships.

Eleven realistic logical changes were made to the original version of the ATM's UML model. These logical changes are of three types: requirements changes (five), design improvements (two), and error corrections (four). These logical changes resulted in 70 model element changes, and their distribution in the change taxonomy is shown in Table 2. The change taxonomy is presented in Appendix A and the complete description of the logical changes is provided in Appendix D.

Table 2: Change Distribution – ATM Case Study

Change Code	Change Title	No. of Changes
CCDVAA	Changed ClassDiagramView – Added Association	1
CCDVAC	Changed ClassDiagramView – Added Class	1
CCDVAI	Changed ClassDiagramView – Added Interface	1
CCDVAR	Changed ClassDiagramView – Added Realization	1
CSDVACR	Changed SequenceDiagramView – Added ClassifierRole	2
CSDVAM	Changed SequenceDiagramView – Added Message	35
CSDVDM	Changed SequenceDiagramView – Deleted Message	12
CAECm	Changed AssociationEnd – Changed multiplicity	2
CCAA	Changed ClassClassifier – Added Attribute	2
CCDA	Changed ClassClassifier – Deleted Attribute	1
CCCiAbs	Changed ClassClassifier – Changed isAbstract	1
CCAO	Changed ClassClassifier – Added Operation	4
CCRCbCC	Changed ClassifierRole – Changed base ClassClassifier	4
CMAcr	Changed Message action – Changed recurrence	1
CACiV	Changed Attribute – Changed initialValue	1
CCOCpst	Changed ClassClassifier Operation – Changed postcondition	1

A few of the logical changes are described as follows: One logical change (requirement change) stems from the need to be able to keep track of how many times per session a user attempts to enter the PIN – after three invalid PIN entries the card is retained. This logical change translates into eleven model element changes. Another logical change (model improvement) changed the ATM's state representation from an integer to an enumeration class, and resulted in 34 model element changes. Two of the logical changes provide confirmation messages to the ATM users. In addition, one of the logical changes consists of making the Account class abstract since only its subclasses, such as Savings, are instantiated. One of the logical changes that corresponded to an error correction was the correction of a spelling mistake in the invariant of the Savings class.

8.2 Results

Of the 70 model element changes resulting from the eleven logical changes, 53 resulted in impacts. This is due to a combination of: (1) the corresponding rules not identifying any impacts since it is certain that no further change is necessary to accomplish the original change, and (2) the conditions necessary to identify impacted elements have not been met. For example, the CCAO change (see Table 2) does not result in any impacts because the rule states that there are no impacts since the model is assumed consistent (see Appendix B). Another example is that one of the CAECm changes does not result in any impacts because the changed association end is not navigable – a necessary condition for the identification of impacts.

Figure 43, Figure 44, and Figure 45 each plots, for every one of the 53 model element changes, a curve/point representing the cumulative number of impacted elements, classes, and operations (y-axis), respectively for each distance value (x-axis). Each of these diagrams shows only four curves because only four changes propagate impacts farther than a distance of one. The reason for this is that impacts are propagated only if other impact analysis rules are invoked – leading to indirect impacts. In this case, most (32) of the changes do not result in impacts being propagated past distance one because of either: (1) the rules that are invoked (at distance one) do not specify any invoked rules, or (2) the necessary conditions for the invocations of the rules invoked by the rules at distance one are not met. For example, impact analysis rules corresponding to the CCCiAbs and CCDVAA changes do not specify the invocation of any other rules, therefore no impacts are propagated. Note that the CCCiAbs and CCDVAA rules do not specify invoked rules since the model is assumed to be consistent and so these rules do not result in indirect impacts. In another example, the CSDVAM impact analysis rule specifies the invocation of another rule (CCOCpst). However, in some cases, the necessary condition – the identification of the operation that sends the added message, was not satisfied because the message does not have an activator (because it was sent by an actor). Note that some of the curves in Figure 43, Figure 44, and Figure 45 overlap because they correspond to the same cumulative number of impacts at the same distance. In addition, note that each element in the graphs is counted only once, regardless of how many times it has been impacted.

More importantly, when there is propagation of impacts, Figure 43, Figure 44, and Figure 45 show that the curves are not exponential, as suggested in [8] (see Section

1.2), but rather linear. This is important as it suggests that the impact analysis rules defined in this work are rather precise. In addition, the maximum distance for impacted elements is limited to six.

In the analysis above, impact analysis is performed for all logical changes together. However, if required, impact analysis can be performed for each logical change separately to evaluate its individual cost, for example in deciding on what logical changes to implement in the next release. This is done by making the changes, to the UML model, corresponding only to the desired logical change – the impact analysis method used in this work is unable to identify logical changes.

Figure 43: *Cumulative number of all impacted elements vs. distance.*

Figure 44: *Cumulative number of impacted classes vs. distance.*

Figure 45: *Cumulative number of impacted operations vs. distance.*

8.3 Accuracy

An investigation was done to examine the accuracy of the strategy with respect to the implementation (source code) of the model. In this investigation, the impacts resulting from a subset (six) of the eleven logical changes made to the ATM's UML model was checked against the ATM's source code. The only logical changes not used in this investigation are those that do not produce impacts to the implementation. For example, one of the logical changes not used involves correcting a syntax error in the postcondition of an operation. This logical change is not included because such a change does not affect the code – it is an error correction in the model and does not change its semantics.

In this investigation, each impact (both direct and indirect) resulting from the selected logical changes was checked against the code to determine if a change in the implementation is required. Here, only changes to the model that affect the implementation are considered – assuming the implementation is consistent with the

model. For example, if a message is added to a sequence diagram then there must be a change to the implementation of the class that sends this added message. Therefore, by checking the impacts against the code, it is assumed that the set of false-positives (over-estimates) will not contain any impacts that affect the model and does not affect the code.

The results of the investigation are shown in Table 3, which shows, for each logical change, code impacts¹⁵ at different distances for classes and operations, as well as the corresponding false-positives. The distances are indicated by commas (“,”), where, for instance, “17, 0” indicates 17 code impacts at distance one and zero code impacts at distance two, and “4” indicates four code impacts at distance one. For instance, for logical change #2, the table shows the identification of 17 changed classes at distance one and two changed classes at distance two, and it appears that the changes at distance one are the only ones that result in a change to the code. Note that in the table, impacts are counted as oppose to impacted elements – impacts includes several impacts to the same element.

Table 3: Code Impacts vs. False-Positives – ATM Case Study

Logical Change	Classes		Operations	
	Code Impacts	False-Positives	Code Impacts	False-Positives
1	4	0	1	0
2	17, 0	0, 2	10, 0	0, 2
3	1, 0, 0, 0, 0, 0	0, 4, 3, 5, 4, 1	1, 0, 0, 0, 0, 0	0, 4, 3, 5, 4, 1
4	1, 0, 0, 0	0, 1, 1, 1	1, 0, 0, 0	0, 1, 1, 1
5	1	0	0	0
6	1	0	0	0

¹⁵ Code impacts are impacts that affect the implementation.

The results in Table 3 indicate that there are no code impacts above distance one. The results also show that all the impacts occurring at distance one are code impacts.

8.4 Conclusions

Though more case studies are necessary to draw definitive conclusions, it can be stated that the results in Figure 43, Figure 44, and Figure 45 are probably due to the use of semantic-based impact analysis rules, instead of connectivity graphs (see [8]). Thus, facilitating a more refined identification of impacted elements as well as the reduction of false-positives.

The results in Table 3 suggest that only the impacts at distance one should be considered. However, this observation may not be correct and should be verified by performing other case studies on larger systems – possible reasons for this are:

1. Encapsulation in object-oriented systems reduces the propagation of changes. In addition, encapsulation strength varies from system-to-system depending on the knowledge and experience of the software designers, amongst other factors. Therefore, in software systems where encapsulation is not enforced, changes may result in code impacts above distance one.
2. The reason for the absence of code impacts above distance one is because of the nature of the changes made. The third and fourth logical changes (in Table 3) involved the display of messages on the ATM display; this did not result in any real effect on the system. For instance, the changes made to operations that send the display messages, do not result in the need for changes to the

operations that invoke these operations. On the other hand, a change¹⁶ to an operation's postcondition resulting in, for example, the negation of a Boolean result, would impact any operation that invokes this operation. This could change the result of this subsequent operation, leading to code impacts above distance one. Therefore, the nature of the changes on which impact analysis is performed may determine if code impacts occur above distance one.

3. The logical changes resulted in changes that cover only some of the changes in the taxonomy. Therefore, it cannot be concluded that no code impacts will occur at distances above one, since some of the changes have not been exercised.

The fact that all the impacts occurring at distance one are code impacts would suggest that the impact analysis rules are accurate in identifying direct impacts. However, as with the previous observation, more case studies are needed to verify this assumption.

¹⁶ In the identification of model changes for the case study, such a change was not feasible – one of the main objectives in selecting changes was to select changes that are as realistic as possible.

Chapter 9 – Conclusions

Presented in this thesis is a methodology, supported by a prototype tool (IACMTool), to perform impact analysis and change management of UML software analysis/design models. Consistency verification rules, change detection criteria, as well as impact analysis rules have been formally defined in OCL on an adaptation of the official UML meta-model [1].

The impact analysis methodology employed in this work, as well as the tool is assessed through a case study, thus providing an initial demonstration of its feasibility and practicality. The results are encouraging as it was shown that, with carefully designed impact analysis rules based on UML diagram semantics, and assumptions on the way the notation is used, the number of elements impacted by changes grows linearly (and not exponentially) when accounting for indirect impacts. This suggests that the impact analysis rules are rather precise, an important result given that a refined identification of impacted elements and the reduction of false-positives is known to be a major challenge when automating impact analysis.

An investigation was also performed, in the ATM case study, on the accuracy of the impact analysis rules with regards to the code. The results indicate that the strategy is accurate in identifying direct impacts. However, no code impacts (impacts that affect the implementation) were found in the set of indirect impacts. These results should be verified by additional case studies because encapsulation as well as the nature of the changes made may affect the results. It is also important to cover as many changes in the taxonomy as possible, since this would provide results that are more definitive.

The notion of distance measure was also defined to be able to sort impacts, according to their likelihood of occurrence, based on the distance between changed model elements and impacted elements. Empirical verification of this measure is required to determine if it is a good heuristic.

Though a conscious effort was made to be as exhaustive as possible when identifying consistency rules, possible changes to UML models, and impact analysis rules, the strategy may be refined as more experience is gained, especially by applying the change impact analysis strategy to additional case studies. All three types of rules have been defined using OCL on the UML metamodel and it is expected that such precise and formal definitions will help refine and evolve the methodology.

9.1 Thesis Limitations

The limitations of the thesis are divided into the following three categories: practical limitation, limitations of the methodology, and limitations due to implementation issues.

9.1.1 Practical Limitation

- The refinement of the impact analysis strategy is limited by the information present in the UML model. For example, if only empty operation contracts are used then the strategy will not be able to identify the attributes used by an operation. This is used in the rules that address changes to attributes.

9.1.2 Limitations of the Methodology

- Since a UML model describes a software system at a higher level of abstraction than the source code, then the strategy used here may provide less precise results than code-based strategies. For example, unexpected results that cannot be determined from the model may appear at implementation time.
- The impact analysis framework presented in this thesis makes certain assumptions about the constructs present in the metamodel. For instance, the framework assumes that classes are used and that they have attributes, which may be static. In addition, it is assumed that aggregation and composition relationships exist. However, if the definitions of these constructs were to be changed in say, a future version of UML then modifications will have to be made to the framework.

9.1.3 Implementation Limitations

- Semantic analysis of OCL model constraints was not done, therefore, a simple syntax change to such a constraint would be considered by the tool to be more severe than it actually is. For example, if the postcondition `account.balance() > 0` is changed to `account.balance() > 0` then this would be treated as a changed postcondition, rather than just a syntax change because the strategy is unable to detect the difference – a spelling mistake (“account”) was corrected.
- The tool can only read XMI 1.1 UML 1.4 files. The reason for this is that the development of an XMI parser was not one of the main objectives of the

work. It was done for completeness and so a conscious effort was made to make the parser as simple as possible without sacrificing the ability to interpret the information needed. However, this does not greatly affect the usability of the tool because CASE tools such as Together® [5] are able to read various XMI formats and generate the required format.

9.2 Future Work

Additional case studies should be performed to verify the results obtained in the case study. The three issues (encapsulation, nature of the changes, and change distribution) outlined above should be considered when performing these case studies. The framework can be expanded to include additional views, such as use case and activity diagrams. Persistency can also be implemented in the tool to improve its usability. Currently, the UML models that are loaded are not saved to secondary storage and so they have to be reloaded every time the application is started – this may take a long time depending on the size of the model. An object-oriented database, such as probably Poet 9.0 [22], can be used for this purpose. Note however, that the tool is able to save the reports generated.

Another improvement is to implement OCL constraint equivalency check – by performing semantic analysis of OCL constraints. This would provide even better refinement of the impact analysis rules. Currently, to check if an OCL constraint, such as a postcondition, is changed, the entire constraint is treated like a string and then string equivalency check is performed. This was done because of the complexity (and time

limitation) in performing semantic analysis of OCL constraints. More information on semantic analysis of OCL constraints can be found in [23].

A more difficult improvement is to store the rules (consistency verification, change detection, and impact analysis) for example, in a database, and modify the tool to read these rules. The rules would be written in OCL, and thus semantic analysis of OCL constraints is required. This change would facilitate easy addition and modification of the rules, as required. An OCL compiler such as the one discussed in [23] can be used to check the constraints.

References

- [1] OMG, “OMG Unified Modeling Language Specification”, Version 1.4, September 2001, <http://www.omg.org/technology/documents/formal/uml.htm>.
- [2] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1999.
- [3] T. H. Cormen and C. E. Leiserson, R. L. Rivest, *Introduction to Algorithms*, MIT Press, 1990.
- [4] J. Warmer and A. Kleppe, *The Object Constraint Language, Precise Modeling with UML*, Addison-Wesley, 1999.
- [5] TogetherSoft™, “Together®”, <http://www.togethersoft.com>.
- [6] B. Hnatkowska, Z. Huzar, and J. Magott, “Consistency Checking in UML Models”, <http://www.fit.vutbr.cz/events/ism/2001/pdf/hnatkowska.pdf>.
- [7] S. A. Bohner and R. S. Arnold, “An Introduction to Software Change Impact Analysis”, in S. A. Bohner, R. S. Arnold, Eds., *Software Change Impact Analysis*, IEEE Computer Society Press, pp. 1-26, 1996.
- [8] S. A. Bohner, “Software Change Impacts – An Evolving Perspective”, *Proceedings, International Conference on Software Maintenance*, Montreal, Quebec, Canada, IEEE Computer Society Press, pp. 263-272, October 3-6, 2002.
- [9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, pp. 305-313, 1995.
- [10] OMG, “OMG XML Metadata Interchange (XMI) Specification”, Version 1.1, November 2000, <http://www.omg.org/cgi-bin/doc?formal/00-11-02>.
- [11] T. J. Grose, G. C. Doney, and S. A. Brodsky, *Mastering XMI, Java Programming with XMI, XML, and UML*, John Wiley & Sons Inc., 2002.
- [12] L. C. Briand, Y. Labiche, G. Soccar, “Automating Impact Analysis and Regression Test Selection Based on UML Designs”, Carleton University, Technical Report SCE-02-04, http://www.sce.carleton.ca/Squall/Articles/TR_SCE-02-04.pdf, March 2002.
- [13] Sun Microsystems, Java API for XML Processing (JAXP), <http://java.sun.com/xml/jaxp/index.html>, 2002.
- [14] W3C, “Document Object Model (DOM) Level 2 Core Specification”, Version 1.0, <http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113>, 2000.

- [15] B. Bruegge and A. H. Dutoit, *Object-Oriented Software Engineering - Conquering Complex and Changing Systems*, Prentice Hall, 2000.
- [16] S. A Bohner, R. S. Arnold, *Software Change Impact Analysis*, IEEE Computer Society Press, 1996.
- [17] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima and C. Chen, "Change Impact Identification in Object Oriented Software Maintenance", *Proceedings, International Conference on Software Maintenance*, Victoria, British Columbia, Canada, IEEE Computer Society Press, pp. 202-211, September 19 –23, 1994.
- [18] A. von Mayrhauser and N. Zhang, "Automated Regression Testing using DBT and Sleuth", *Journal of Software Maintenance: Research and Practice*, vol. 11 (2), John Wiley & Sons, Ltd., pp. 93-116, March/April 1999.
- [19] L. C. Briand, Y. Labiche and G. Soccar, "Automating Impact Analysis and Regression Test Selection Based on UML Designs", *Proceedings, International Conference on Software Maintenance*, Montreal, Quebec, Canada, IEEE Computer Society, pp. 252-261, October 3-6, 2002.
- [20] L. C. Briand, J. Wüst, and H. Lounis, "Using Coupling Measurement for Impact Analysis in Object-Oriented Systems", *Proceedings, International Conference on Software Maintenance*, Oxford, England, IEEE Computer Society, pp. 475-482, August 30 – September 3, 1999.
- [21] Sun Microsystems, "Java™ 2 Platform, Standard Edition, version 1.4.1 API Specification", <http://java.sun.com/j2se/1.4.1/docs/api/overview-summary.html>.
- [22] Poet Software Corporation, "FastObjects™ 9.0", <http://www.poet.com>.
- [23] Technische Universität Dresden, "OCL Compiler", <http://dresden-ocl.sourceforge.net/index.html>.

Appendices

Appendix A – Change Detection

A.1 Change Taxonomy

The complete change taxonomy is presented below, and it corresponds to the metamodel shown in the `umlModel` subsystem presented in Section 7.2. Note the use of the inheritance relationship between some nodes of the taxonomy; this signifies that the change identified in the subclass node is the same as that in the super-class node. For instance, in Figure 48, `Changed base ClassClassifier` inherits from `Changed ClassClassifier`; this indicates that a `base ClassClassifier` is changed in the same manner as a `ClassClassifier`.

Figure 46: *Change Taxonomy – Changed UMLModel.*

Figure 47: *Change Taxonomy – Changed ClassDiagramView.*

Figure 48: *Change Taxonomy – Changed SequenceDiagramView.*

Figure 49: *Change Taxonomy – Changed StatechartDiagramView.*

Figure 50: *Change Taxonomy – Changed Association.*

Figure 51: *Change Taxonomy – Changed ClassClassifier.*

Figure 52: *Change Taxonomy – Changed Attribute.*

Figure 53: *Change Taxonomy – Changed Operation.*

Figure 54: *Change Taxonomy – Changed Interface.*

Figure 55: *Change Taxonomy – Changed Parameter.*

Figure 56: *Change Taxonomy – Changed State.*

Figure 57: *Change Taxonomy – Changed StateMachine action.*

A.2 Change Detection Rules

The change detection rules defined in the thesis are listed below. Each rule contains the change title, a brief description of the change, followed by the OCL expression specifying how to detect the change. The changed version of the model is the version context for the rules. The following operations are used in the rules:

<code>equals:</code>	Returns a Boolean value indicating whether two values are equal.
<code>getIDstr:</code>	Returns a string that uniquely identifies a model element in the model.
<code>getParameterPosition:</code>	Returns an integer value corresponding to the position of a parameter in an operation's signature.

1. Changed ClassDiagramView - Added Association

Change Code: CCDVAA

Description: In the changed model version there exists association relationships that do not exist in the original version.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.association->select(a:Association|not self.model.
application.originalModel.classDiagramView.
association->includes(a))`

2. Changed ClassDiagramView - Deleted Association

Change Code: CCDVDA

Description: In the original model version there exist association relationships that do not exist in the changed version.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.model.application.originalModel.classDiagramView.
association->exists(a:Association|not self.
association->includes(a))`

3. Changed ClassDiagramView - Added Class

Change Code: CCDVAC

Description: In the changed model version there exist classes that do not exist in the original version.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.classClassifier->select(c:ClassClassifier|not self.model.
application.originalModel.classDiagramView.
classClassifier->includes(c))`

4. Changed ClassDiagramView - Deleted Class

Change Code: CCDVDC

Description: In the original model version there exist classes that do not exist in the changed version.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.model.application.originalModel.classDiagramView.
classClassifier->exists(c:ClassClassifier|not self.
classClassifier->includes(c))`

5. Changed ClassDiagramView - Added Dependency

Change Code: CCDVAD

Description: In the changed model version there exist dependency relationships that do not exist in the original version.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.dependency->select(d:Dependency|not self.model.
application.originalModel.classDiagramView.
dependency->includes(d))`

6. Changed ClassDiagramView - Deleted Dependency

Change Code: CCDVDD

Description: In the original model version there exist dependency relationships that do not exist in the changed version.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.model.application.originalModel.classDiagramView.
dependency->exists(d:Dependency|not self.
dependency->includes(d))`

7. Changed ClassDiagramView - Added Generalization

Change Code: CCDVAG

Description: In the changed model version there exist generalization relationships that do not exist in the original version.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.generalization->select(g:Generalization|not self.model.
application.originalModel.classDiagramView.
generalization->includes(g))`

8. Changed ClassDiagramView - Deleted Generalization

Change Code: CCDVDG

Description: In the original model version there exist generalization relationships that do not exist in the changed version.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.model.application.originalModel.classDiagramView.
generalization->exists(g:Generalization|not self.
generalization->includes(g))`

9. Changed ClassDiagramView - Added Interface

Change Code: CCDVAI

Description: In the changed model version there exist interfaces that do not exist in the original version.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.interface->select(i:Interface|not self.model.
application.originalModel.classDiagramView.
interface->includes(i))`

10. **Changed ClassDiagramView - Deleted Interface**
Change Code: CCDVDI
Description: In the original model version there exist interfaces that do not exist in the changed version.
OCL Expression: `context model::foundation::core::ClassDiagramView
self.model.application.originalModel.classDiagramView.
interface->exists(i:Interface|not self.
interface->includes(i))`
11. **Changed ClassDiagramView - Added Realization**
Change Code: CCDVAR
Description: In the changed model version there exist realization relationships that do not exist in the original version.
OCL Expression: `context model::foundation::core::ClassDiagramView
self.realization->select(r:Realization|not self.model.
application.originalModel.classDiagramView.
realization->includes(r))`
12. **Changed ClassDiagramView - Deleted Realization**
Change Code: CCDVDR
Description: In the original model version there exist realization relationships that do not exist in the changed version.
OCL Expression: `context model::foundation::core::ClassDiagramView
self.model.application.originalModel.classDiagramView.
realization->exists(r:Realization|not self.
realization->includes(r))`
13. **Changed SequenceDiagramView - Added ClassifierRole**
Change Code: CSDVACR
Description: In the changed model version there exist classifier roles that do not exist in the original version.
OCL Expression: `context model::behaviouralElements::collaborations::
SequenceDiagramView
self.classifierRole->select(cr:ClassifierRole|not self.model.
application.originalModel.sequenceDiagramView.
classifierRole->includes(cr))`
14. **Changed SequenceDiagramView - Deleted ClassifierRole**
Change Code: CSDVDCR
Description: In the original model version there exist classifier roles that do not exist in the changed version.
OCL Expression: `context model::behaviouralElements::collaborations::
SequenceDiagramView
self.model.application.originalModel.sequenceDiagramView.
classifierRole->exists(cr:ClassifierRole|not self.
classifierRole->includes(cr))`
15. **Changed SequenceDiagramView - Added Message**
Change Code: CSDVAM
Description: In the changed model version there exist messages that do not exist in the original version.
OCL Expression: `context model::behaviouralElements::collaborations::
SequenceDiagramView
self.message->select(m:Message|not self.model.
application.originalModel.sequenceDiagramView.
message->includes(m))`

16. Changed SequenceDiagramView - Deleted Message

Change Code: CSDVDM

Description: In the original model version there exist messages that do not exist in the changed version.

OCL Expression: `context model::behaviouralElements::collaborations::
SequenceDiagramView
self.model.application.originalModel.sequenceDiagramView.
message->exists(m:Message|not self.
message->includes(m))`

17. Changed StatechartDiagramView - Added CompositeState

Change Code: CStDVACS

Description: In the changed model version there exist composite states that do not exist in the original version.

OCL Expression: `context model::behaviouralElements::stateMachines::
StatechartDiagramView
self.compositeState->select(cs:CompositeState|not self.model.
application.originalModel.statechartDiagramView.
compositeState->includes(cs))`

18. Changed StatechartDiagramView - Deleted CompositeState

Change Code: CStDVDCS

Description: In the original model version there exist composite states that do not exist in the changed version.

OCL Expression: `context model::behaviouralElements::stateMachines::
StatechartDiagramView
self.model.application.originalModel.statechartDiagramView.
compositeState->exists(cs:CompositeState|not self.
compositeState->includes(cs))`

19. Changed StatechartDiagramView - Added SimpleState

Change Code: CStDVASS

Description: In the changed model version there exist simple states that do not exist in the original version.

OCL Expression: `context model::behaviouralElements::stateMachines::
StatechartDiagramView
self.simpleState->select(ss:SimpleState|not self.model.
application.originalModel.statechartDiagramView.
simpleState->includes(ss))`

20. Changed StatechartDiagramView - Deleted SimpleState

Change Code: CStDVDSS

Description: In the original model version there exist simple states that do not exist in the changed version.

OCL Expression: `context model::behaviouralElements::stateMachines::
StatechartDiagramView
self.model.application.originalModel.statechartDiagramView.
simpleState->exists(ss:SimpleState|not self.
simpleState->includes(ss))`

21. Changed StatechartDiagramView - Added Transition

Change Code: CStDVAT

Description: In the changed model version there exist transitions that do not exist in the original version.

OCL Expression: `context model::behaviouralElements::stateMachines::
StatechartDiagramView
self.transition->select(t:Transition|not self.model.
application.originalModel.statechartDiagramView.
transition->includes(t))`

22. Changed StatechartDiagramView - Deleted Transition

Change Code: CSiDVT

Description: In the original model version there exist transitions that do not exist in the changed version.

OCL Expression: `context model::behaviouralElements::stateMachines::
StatechartDiagramView
self.model.application.originalModel.statechartDiagramView.
transition->exists(t:Transition|not self.
transition->includes(t))`

23. Changed AssociationEnd - Changed aggregation

Change Code: CAECa

Description: There exists a set of association ends in the model such that the aggregation property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.association.end->select(e1:AssociationEnd|e1.
aggregation <> self.model.application.originalModel.
classDiagramView.association->select(a:Association|
a.getIDStr() = e1.association.getIDStr()).
end->select(e2:AssociationEnd|e1.getIDStr() = e2.getIDStr()).
aggregation)`

24. Changed AssociationEnd - Changed changeability

Change Code: CAECc

Description: There exists a set of association ends in the model such that the changeability property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.association.end->select(e1:AssociationEnd|e1.
changeability <> self.model.application.originalModel.
classDiagramView.association->select(a:Association|
a.getIDStr() = e1.association.getIDStr()).
end->select(e2:AssociationEnd|e1.getIDStr() = e2.getIDStr()).
changeability)`

25. Changed AssociationEnd - Added interfaceSpecifier

Change Code: CAEAiS

Description: There exists a set of association ends in the model such that each one has at least one interface specifier in the changed model version that it didn't have in the original version.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.association.end->select(e1:AssociationEnd|
e1.interfaceSpecifier->exists(i1:Classifier|not self.
model.application.originalModel.classDiagramView.
association->select(a:Association|a.getIDStr() =
e1.association.getIDStr()).end->select(e2:AssociationEnd|
e2.getIDStr() = e1.getIDStr()).
interfaceSpecifier->exists(i2:interfaceSpecifier|
i1.getIDStr() = i2.getIDStr())))`

26. Changed AssociationEnd - Changed interfaceSpecifier

Change Code: CAECiS

Description: There exists a set of association ends in the model such that each one has an interface specifier that is not the same in both model versions. Note that the implementation of this rule assumes that all the changed interface and class classifiers in the model have been previously identified.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.association.end->select(e:AssociationEnd|
e.interfaceSpecifier->exists(i1:Classifier|self.model.
application.changeDetector.change.
changedElement->exists(i2:Classifier|i1.getIDStr() =
i2.getIDStr())))`

27. Changed AssociationEnd - Deleted interfaceSpecifier

Change Code: CAEDiS

Description: There exists a set of association ends in the model such that each one has at least one interface specifier in the original model version that it doesn't have in the changed version.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.model.application.originalModel.classDiagramView.
association.end->select(e1:AssociationEnd|e1.
interfaceSpecifier->exists(i1:Classifier|not self.
association->select(a:Association:a.getIDStr() =
e1.association.getIDStr()).end->select(e2:AssociationEnd|
e2.getIDStr() = e1.getIDStr()).
interfaceSpecifier->exists(i2:Classifier|
i1.getIDStr() = i2.getIDStr())))`

28. Changed AssociationEnd - Changed isNavigable

Change Code: CAECiN

Description: There exists a set of association ends in the model such that the isNavigable property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.association.end->select(e1:AssociationEnd|e1.
isNavigable <> self.model.application.originalModel.
classDiagramView.association->select(a:Association|
a.getIDStr() = e1.association.getIDStr()).
end->select(e2:AssociationEnd|e1.getIDStr() = e2.getIDStr()).
isNavigable)`

29. Changed AssociationEnd - Changed multiplicity

Change Code: CAECm

Description: There exists a set of association ends in the model such that the multiplicity property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.association.end->select(e1:AssociationEnd|not e1.
multiplicity.equals(self.model.application.originalModel.
classDiagramView.association->select(a:Association|
a.getIDStr() = e1.association.getIDStr()).
end->select(e2:AssociationEnd|e1.getIDStr() = e2.getIDStr()).
multiplicity))`

30. Changed AssociationEnd - Changed ordering

Change Code: CAECo

Description: There exists a set of association ends in the model such that the ordering property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.association.end->select(e1:AssociationEnd|e1.
ordering <> self.model.application.originalModel.
classDiagramView.association->select(a:Association|
a.getIDStr() = e1.association.getIDStr()).
end->select(e2:AssociationEnd|e1.getIDStr() = e2.getIDStr()).
ordering)`

31. Changed AssociationEnd - Added qualifier

Change Code: CAEAq

Description: There exists a set of association ends in the model such that each one has at least one qualifier in the changed model version that it didn't have in the original version.

OCL Expression:

```
context model::foundation::core::ClassDiagramView
self.association.end->select(e1:AssociationEnd|
e1.qualifier->exists(q1:Attribute|not self.
model.application.originalModel.classDiagramView.
association->select(a:Association|a.getIDStr() =
e1.association.getIDStr()).end->select(e2:AssociationEnd|
e2.getIDStr() = e1.getIDStr()).
qualifier->exists(q2:Attribute|q1.getIDStr() = q2.getIDStr())))
```

32. Changed AssociationEnd - Changed qualifier type

Change Code: CAECqt

Description: There exists a set of association ends in the model such that for each one the type property of at least one of its qualifiers is not the same in both model versions.

OCL Expression:

```
context model::foundation::core::ClassDiagramView
self.association.end->select(e1:AssociationEnd|
e1.qualifier->exists(q1:Attribute|not q1.type.equals(self.
model.application.originalModel.classDiagramView.
association->select(a:Association|a.getIDStr() =
e1.association.getIDStr()).end->select(e2:AssociationEnd|
e2.getIDStr() = e1.getIDStr()).qualifier->select(q2:Attribute|
q1.getIDStr() = q2.getIDStr()).type))
```

33. Changed AssociationEnd - Deleted qualifier

Change Code: CAEDq

Description: There exists a set of association ends in the model such that each one has at least one qualifier in the original model version that it doesn't have in the changed version.

OCL Expression:

```
context model::foundation::core::ClassDiagramView
self.model.application.originalModel.classDiagramView.
association.end->select(e1:AssociationEnd|e1.
qualifier->exists(q1:Attribute|not self.
association->select(a:Association|a.getIDStr() =
e1.association.getIDStr()).end->select(e2:AssociationEnd|
e2.getIDStr() = e1.getIDStr()).
qualifier->exists(q2:Attribute|
q1.getIDStr() = q2.getIDStr())))
```

34. Changed AssociationEnd - Changed targetScope

Change Code: CAECTS

Description: There exists a set of association ends in the model such that the targetScope property of each one is not the same in both model versions.

OCL Expression:

```
context model::foundation::core::ClassDiagramView
self.association.end->select(e1:AssociationEnd|e1.
targetScope <> self.model.application.originalModel.
classDiagramView.association->select(a:Association|
a.getIDStr() = e1.association.getIDStr()).
end->select(e2:AssociationEnd|e1.getIDStr() = e2.getIDStr()).
targetScope)
```

35. Changed AssociationEnd - Changed visibility

Change Code: CAECv

Description: There exists a set of association ends in the model such that the visibility property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.association.end->select(e1:AssociationEnd|e1.
visibility <> self.model.application.originalModel.
classDiagramView.association->select(a:Association|
a.getIDStr() = e1.association.getIDStr()).
end->select(e2:AssociationEnd|e1.getIDStr() = e2.getIDStr()).
visibility)`

36. Changed ClassClassifier - Added Attribute

Change Code: CCAA

Description: There exists a set of classes in the model such that each one has at least one attribute in the changed model version that it didn't have in the original version.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.classClassifier->select(c1:ClassClassifier|
c1.attribute->exists(a1:Attribute|not self.
model.application.originalModel.classDiagramView.
classClassifier->select(c2:ClassClassifier|
c2.getIDStr() = c1.getIDStr()).
attribute->exists(a2:Attribute|a1.getIDStr() = a2.getIDStr())))`

37. Changed ClassClassifier - Deleted Attribute

Change Code: CCDA

Description: There exists a set of classes in the model such that each one has at least one attribute in the original model version that it doesn't have in the changed version.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.model.application.originalModel.classDiagramView.
classClassifier->select(c1:ClassClassifier|c1.
attribute->exists(a1:Attribute|not self.
classClassifier->select(c2:ClassClassifier|
c2.getIDStr() = c1.getIDStr()).
attribute->exists(a2:Attribute|
a1.getIDStr() = a2.getIDStr())))`

38. Changed ClassClassifier - Changed invariant

Change Code: CCCi

Description: There exists a set of class classifiers in the model such that the invariant property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.classClassifier->select(c1:ClassClassifier|not c1.
invariant.equals(self.model.application.originalModel.
classDiagramView.classClassifier->select(c2:ClassClassifier|
c1.getIDStr() = c2.getIDStr()).invariant))`

39. Changed ClassClassifier - Changed isAbstract

Change Code: CCCiAbs

Description: There exists a set of class classifiers in the model such that the isAbstract property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.classClassifier->select(c1:ClassClassifier|c1.
isAbstract <> self.model.application.originalModel.
classDiagramView.classClassifier->select(c2:ClassClassifier|
c1.getIDStr() = c2.getIDStr()).isAbstract)`

40. Changed ClassClassifier - Changed isActive

Change Code: CCCiA

Description: There exists a set of class classifiers in the model such that the isActive property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.classClassifier->select(c1:ClassClassifier|c1.
isActive <> self.model.application.originalModel.
classDiagramView.classClassifier->select(c2:ClassClassifier|
c1.getIDStr() = c2.getIDStr()).isActive)`

41. Changed ClassClassifier - Changed isLeaf

Description: There exists a set of class classifiers in the model such that the isLeaf property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.classClassifier->select(c1:ClassClassifier|c1.
isLeaf <> self.model.application.originalModel.
classDiagramView.classClassifier->select(c2:ClassClassifier|
c1.getIDStr() = c2.getIDStr()).isLeaf)`

42. Changed ClassClassifier - Changed isRoot

Change Code: CCCiR

Description: There exists a set of class classifiers in the model such that the isRoot property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.classClassifier->select(c1:ClassClassifier|c1.
isRoot <> self.model.application.originalModel.
classDiagramView.classClassifier->select(c2:ClassClassifier|
c1.getIDStr() = c2.getIDStr()).isRoot)`

43. Changed ClassClassifier - Changed multiplicity

Change Code: CCCm

Description: There exists a set of class classifiers in the model such that the multiplicity property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.classClassifier->select(c1:ClassClassifier|not c1.
multiplicity.equals(self.model.application.originalModel.
classDiagramView.classClassifier->select(c2:ClassClassifier|
c1.getIDStr() = c2.getIDStr()).multiplicity))`

44. Changed ClassClassifier - Added Operation

Change Code: CCAO

Description: There exists a set of classes in the model such that each one has at least one operation in the changed model version that it didn't have in the original version.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.classClassifier->select(c1:ClassClassifier|
c1.operation->exists(o1:Operation|not self.
model.application.originalModel.classDiagramView.
classClassifier->select(c2:ClassClassifier|
c2.getIDStr() = c1.getIDStr()).
operation->exists(o2:Operation|o1.getIDStr() = o2.getIDStr()))`

45. Changed ClassClassifier - Deleted Operation

Change Code: CCDO

Description: There exists a set of classes in the model such that each one has at least one operation in the original model version that it doesn't have in the changed version.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.model.application.originalModel.classDiagramView.
classClassifier->select(c1:ClassClassifier|c1.
operation->exists(o1:Operation|not self.
classClassifier->select(c2:ClassClassifier|
c2.getIDStr() = c1.getIDStr()).
operation->exists(o2:Operation|
o1.getIDStr() = o2.getIDStr()))`

46. Changed ClassClassifier - Changed visibility

Change Code: CCCv

Description: There exists a set of class classifiers in the model such that the visibility property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.classClassifier->select(c1:ClassClassifier|c1.
visibility <> self.model.application.originalModel.
classDiagramView.classClassifier->select(c2:ClassClassifier|
c1.getIDStr() = c2.getIDStr()).visibility)`

47. Changed Interface - Added Operation

Change Code: CIAO

Description: There exists a set of interfaces in the model such that each one has at least one operation in the changed model version that it didn't have in the original version.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.interface->select(i1:Interface|
i1.operation->exists(o1:Operation|not self.
model.application.originalModel.classDiagramView.
interface->select(i2:Interface|
i2.getIDStr() = i1.getIDStr()).
operation->exists(o2:Operation|o1.getIDStr() = o2.getIDStr()))`

48. Changed Interface - Deleted Operation

Change Code: CIDO

Description: There exists a set of interfaces in the model such that each one has at least one operation in the original model version that it doesn't have in the changed version.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.model.application.originalModel.classDiagramView.
interface->select(i1:Interface|i1.
operation->exists(o1:Operation|not self.
interface->select(i2:Interface|
c2.getIDStr() = c1.getIDStr()).
operation->exists(o2:Operation|
o1.getIDStr() = o2.getIDStr()))`

49. Changed ClassifierRole - Added availableOperation

Change Code: CCRaao

Description: There exists a set of classifier roles in the model such that each one has at least one available operation in the changed model version that it didn't have in the original version.

OCL Expression: `context model::behaviouralElements::collaborations::
SequenceDiagramView
self.classifierRole->select(cr1:ClassifierRole|
cr1.availableOperation->exists(ao1:Operation|not self.
model.application.originalModel.sequenceDiagramView.
classifierRole->select(cr2:ClassifierRole|
cr2.getIDStr() = cr1.getIDStr()).
availableOperation->exists(ao2:Operation|
ao1.getIDStr() = ao2.getIDStr()))`

50. Changed ClassifierRole - Deleted availableOperation

Change Code: CCRDaO

Description: There exists a set of classifier roles in the model such that each one has at least one available operation in the original model version that it doesn't have in the changed version.

OCL Expression: `context model::behaviouralElements::collaborations::
SequenceDiagramView
self.model.application.originalModel.sequenceDiagramView.
classifierRole->select(cr1:ClassifierRole|cr1.
availableOperation->exists(ao1:Operation|not self.
classifierRole->select(cr2:ClassifierRole|
cr2.getIDStr() = cr1.getIDStr()).
availableOperation->exists(ao2:Operation|
ao1.getIDStr() = ao2.getIDStr()))`

51. Changed ClassifierRole - Changed base ClassClassifier

Change Code: CCRcbCC

Description: There exists a set of classifier roles in the model such that the base class classifier property of each one is not the same in both model versions.

OCL Expression: `context model::behaviouralElements::collaborations::
SequenceDiagramView
self.classifierRole->select(cr1:ClassifierRole|not cr1.
base.oclAsType(ClassClassifier).equals(self.model.
application.originalModel.sequenceDiagramView.
classifierRole->select(cr2:ClassifierRole|
cr1.getIDStr() = cr2.getIDStr()).base.
oclAsType(ClassClassifier)))`

52. Changed ClassifierRole - Changed multiplicity

Change Code: CCRcm

Description: There exists a set of classifier roles in the model such that the multiplicity property of each one is not the same in both model versions.

OCL Expression: `context model::behaviouralElements::collaborations::
SequenceDiagramView
self.classifierRole->select(cr1:ClassifierRole|not cr1.
multiplicity.equals(self.model.application.originalModel.
sequenceDiagramView.classifierRole->select(cr2:ClassifierRole|
cr1.getIDStr() = cr2.getIDStr()).multiplicity))`

53. Changed Message action - Changed recurrence

Change Code: CMaCr

Description: There exists a set of messages in the model such that for each of these messages, the recurrence property of its action is not the same in the two model versions.

OCL Expression: `context model::behaviouralElements::collaborations::
SequenceDiagramView
self.message->select(m1:Message|not m1.action.recurrence.
equals(self.model.application.originalModel.
sequenceDiagramView.message->select(m2:Message|m1.getIDStr() =
m2.getIDStr()).action.recurrence))`

54. Changed CompositeState - Added subvertex

Change Code: CCSAs

Description: There exists a set of composite states in the model such that each one has at least one subvertex in the changed model version that it didn't have in the original version.

OCL Expression: `context model::behaviouralElements::stateMachines::
StatechartDiagramView
self.compositeState->select(cs1:CompositeState|
cs1.subvertex->exists(sv1:StateVertex|not self.
model.application.originalModel.statechartDiagramView.
compositeState->select(cs2:CompositeState|
cs2.getIDStr() = cs1.getIDStr()).
subvertex->exists(sv2:StateVertex|
sv1.getIDStr() = sv2.getIDStr()))`

55. Changed CompositeState - Deleted subvertex

Change Code: CCSDs

Description: There exists a set of composite states in the model such that each one has at least one subvertex in the original model version that it doesn't have in the changed version.

OCL Expression: `context model::behaviouralElements::stateMachines::
StatechartDiagramView
self.model.application.originalModel.statechartDiagramView.
compositeState->select(cs1:CompositeState|cs1.
subvertex->exists(sv1:StateVertex|not self.
compositeState->select(cs2:CompositeState|
cs2.getIDStr() = cs1.getIDStr()).
subvertex->exists(sv2:StateVertex|
sv1.getIDStr() = sv2.getIDStr()))`

56. Changed Transition - Changed guard

Change Code: CTCg

Description: There exists a set of transitions in the model such that the guard property of each one is not the same in both model versions.

OCL Expression: `context model::behaviouralElements::stateMachines::
StatechartDiagramView
self.transition->select(t1:Transition|not t1.
guard.equals(self.model.application.originalModel.
statechartDiagramView.transition->select(t2:Transition|
t1.getIDStr() = t2.getIDStr()).guard))`

57. Changed Attribute - Changed changeability

Change Code: CACc

Description: There exists a set of attributes in the model such that the changeability property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.classClassifier.attribute->select(a1:Attribute|a1.
changeability <> self.model.application.originalModel.
classDiagramView.classClassifier->select(c:ClassClassifier|
c.getIDStr() = a1.classClassifier.getIDStr()).
attribute->select(a2:Attribute|
a1.getIDStr() = a2.getIDStr()).changeability)`

58. Changed Attribute - Changed initialValue

Change Code: CACiV

Description: There exists a set of attributes in the model such that the initialValue property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.classClassifier.attribute->select(a1:Attribute|a1.
initialValue <> self.model.application.originalModel.
classDiagramView.classClassifier->select(c:ClassClassifier|
c.getIDStr() = a1.classClassifier.getIDStr()).
attribute->select(a2:Attribute|
a1.getIDStr() = a2.getIDStr()).initialValue)`

59. Changed Attribute - Changed multiplicity

Change Code: CACm

Description: There exists a set of attributes in the model such that the multiplicity property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.classClassifier.attribute->select(a1:Attribute|not a1.
multiplicity.equals(self.model.application.originalModel.
classDiagramView.classClassifier->select(c:ClassClassifier|
c.getIDStr() = a1.classClassifier.getIDStr()).
attribute->select(a2:Attribute|
a1.getIDStr() = a2.getIDStr()).multiplicity))`

60. Changed Attribute - Changed ordering

Change Code: CACo

Description: There exists a set of attributes in the model such that the ordering property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.classClassifier.attribute->select(a1:Attribute|a1.
ordering <> self.model.application.originalModel.
classDiagramView.classClassifier->select(c:ClassClassifier|
c.getIDStr() = a1.classClassifier.getIDStr()).
attribute->select(a2:Attribute|
a1.getIDStr() = a2.getIDStr()).ordering)`

61. Changed Attribute - Changed ownerScope

Change Code: CACoS

Description: There exists a set of attributes in the model such that the ownerScope property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.classClassifier.attribute->select(a1:Attribute|a1.
ownerScope <> self.model.application.originalModel.
classDiagramView.classClassifier->select(c:ClassClassifier|
c.getIDStr() = a1.classClassifier.getIDStr()).
attribute->select(a2:Attribute|
a1.getIDStr() = a2.getIDStr()).ownerScope)`

62. Changed Attribute - Changed targetScope

Change Code: CACtS

Description: There exists a set of attributes in the model such that the targetScope property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.classClassifier.attribute->select(a1:Attribute|a1.
targetScope <> self.model.application.originalModel.
classDiagramView.classClassifier->select(c:ClassClassifier|
c.getIDStr() = a1.classClassifier.getIDStr()).
attribute->select(a2:Attribute|
a1.getIDStr() = a2.getIDStr()).targetScope)`

63. Changed Attribute - Changed type

Change Code: CACt

Description: There exists a set of attributes in the model such that the type property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.classClassifier.attribute->select(a1:Attribute|not a1.
type.equals(self.model.application.originalModel.
classDiagramView.classClassifier->select(c:ClassClassifier|
c.getIDStr() = a1.classClassifier.getIDStr()).
attribute->select(a2:Attribute|
a1.getIDStr() = a2.getIDStr()).type))`

64. Changed Attribute - Changed visibility

Change Code: CACv

Description: There exists a set of attributes in the model such that the initialValue property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.classClassifier.attribute->select(a1:Attribute|a1.
visibility <> self.model.application.originalModel.
classDiagramView.classClassifier->select(c:ClassClassifier|
c.getIDStr() = a1.classClassifier.getIDStr()).
attribute->select(a2:Attribute|
a1.getIDStr() = a2.getIDStr()).visibility)`

65. Changed ClassClassifier Operation - Changed concurrency

Change Code: CCOCc

Description: There exists a set of (class) operations in the model such that the concurrency property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.classClassifier.operation->select(o1:Operation|o1.
concurrency <> self.model.application.originalModel.
classDiagramView.classClassifier->select(c:ClassClassifier|
c.getIDStr() = o1.classClassifier.getIDStr()).
operation->select(o2:Operation|
o1.getIDStr() = o2.getIDStr()).concurrency)`

66. Changed ClassClassifier Operation - Changed isAbstract

Change Code: CCOCiAbs

Description: There exists a set of (class) operations in the model such that the isAbstract property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.classClassifier.operation->select(o1:Operation|o1.
isAbstract <> self.model.application.originalModel.
classDiagramView.classClassifier->select(c:ClassClassifier|
c.getIDStr() = o1.classClassifier.getIDStr()).
operation->select(o2:Operation|
o1.getIDStr() = o2.getIDStr()).isAbstract)`

67. Changed ClassClassifier Operation - Changed isPolymorphic

Change Code: CCOCiP

Description: There exists a set of (class) operations in the model such that the isPolymorphic property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.classClassifier.operation->select(o1:Operation|o1.
isPolymorphic <> self.model.application.originalModel.
classDiagramView.classClassifier->select(c:ClassClassifier|
c.getIDStr() = o1.classClassifier.getIDStr()).
operation->select(o2:Operation|
o1.getIDStr() = o2.getIDStr()).isPolymorphic)`

68. Changed ClassClassifier Operation - Changed isQuery

Change Code: CCOCiQ

Description: There exists a set of (class) operations in the model such that the isQuery property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.classClassifier.operation->select(o1:Operation|o1.
isQuery <> self.model.application.originalModel.
classDiagramView.classClassifier->select(c:ClassClassifier|
c.getIDStr() = o1.classClassifier.getIDStr()).
operation->select(o2:Operation|
o1.getIDStr() = o2.getIDStr()).isQuery)`

69. Changed ClassClassifier Operation - Changed ownerScope

Change Code: CCOCoS

Description: There exists a set of (class) operations in the model such that the ownerScope property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.classClassifier.operation->select(o1:Operation|o1.
ownerScope <> self.model.application.originalModel.
classDiagramView.classClassifier->select(c:ClassClassifier|
c.getIDStr() = o1.classClassifier.getIDStr()).
operation->select(o2:Operation|
o1.getIDStr() = o2.getIDStr()).ownerScope)`

70. Changed Class Operation - Changed precondition

Change Code: CCOCpre

Description: There exists a set of (class) operations in the model such that the precondition property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.classClassifier.operation->select(o1:Operation|not o1.
precondition.equals(self.model.application.originalModel.
classDiagramView.classClassifier->select(c:ClassClassifier|
c.getIDStr() = o1.classClassifier.getIDStr()).
operation->select(o2:Operation|
o1.getIDStr() = o2.getIDStr()).precondition))`

71. Changed ClassClassifier Operation - Changed postcondition

Change Code: CCOCpst

Description: There exists a set of (class) operations in the model such that the postcondition property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.classClassifier.operation->select(o1:Operation|not o1.
postcondition.equals(self.model.application.originalModel.
classDiagramView.classClassifier->select(c:ClassClassifier|
c.getIDStr() = o1.classClassifier.getIDStr()).
operation->select(o2:Operation|
o1.getIDStr() = o2.getIDStr()).postcondition))`

72. Changed ClassClassifier Operation - Changed visibility

Change Code: CCOCv

Description: There exists a set of (class) operations in the model such that the visibility property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.classClassifier.operation->select(o1:Operation|o1.
visibility <> self.model.application.originalModel.
classDiagramView.classClassifier->select(c:ClassClassifier|
c.getIDStr() = o1.classClassifier.getIDStr()).
operation->select(o2:Operation|
o1.getIDStr() = o2.getIDStr()).visibility)`

73. Changed Interface Operation - Changed concurrency

Change Code: CIOCc

Description: There exists a set of (interface) operations in the model such that the concurrency property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.interface.operation->select(o1:Operation|o1.
concurrency <> self.model.application.originalModel.
classDiagramView.interface->select(i:Interface|
i.getIDStr() = o1.interface.getIDStr()).
operation->select(o2:Operation|
o1.getIDStr() = o2.getIDStr()).concurrency)`

74. Changed Interface Operation - Changed isPolymorphic

Change Code: CIOCiP

Description: There exists a set of (interface) operations in the model such that the isPolymorphic property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.interface.operation->select(o1:Operation|o1.
isPolymorphic <> self.model.application.originalModel.
classDiagramView.interface->select(i:Interface|
i.getIDStr() = o1.interface.getIDStr()).
operation->select(o2:Operation|
o1.getIDStr() = o2.getIDStr()).isPolymorphic)`

75. Changed Interface Operation - Changed isQuery

Change Code: CIOCiQ

Description: There exists a set of (interface) operations in the model such that the isQuery property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.interface.operation->select(o1:Operation|o1.
isQuery <> self.model.application.originalModel.
classDiagramView.interface->select(i:Interface|
i.getIDStr() = o1.interface.getIDStr()).
operation->select(o2:Operation|
o1.getIDStr() = o2.getIDStr()).isQuery)`

76. Changed Interface Operation - Changed ownerScope

Change Code: CIOCoS

Description: There exists a set of (interface) operations in the model such that the ownerScope property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.interface.operation->select(o1:Operation|o1.
ownerScope <> self.model.application.originalModel.
classDiagramView.interface->select(i:Interface|
i.getIDStr() = o1.interface.getIDStr()).
operation->select(o2:Operation|
o1.getIDStr() = o2.getIDStr()).ownerScope)`

77. Changed Interface Operation - Changed precondition

Change Code: CIOCpre

Description: There exists a set of (interface) operations in the model such that the precondition property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.interface.operation->select(o1:Operation|o1.
precondition <> self.model.application.originalModel.
classDiagramView.interface->select(i:Interface|
i.getIDStr() = o1.interface.getIDStr()).
operation->select(o2:Operation|
o1.getIDStr() = o2.getIDStr()).precondition)`

78. Changed Interface Operation - Changed postcondition

Change Code: CIOCpst

Description: There exists a set of (interface) operations in the model such that the postcondition property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.interface.operation->select(o1:Operation|o1.
postcondition <> self.model.application.originalModel.
classDiagramView.interface->select(i:Interface|
i.getIDStr() = o1.interface.getIDStr()).
operation->select(o2:Operation|
o1.getIDStr() = o2.getIDStr()).postcondition)`

79. Changed (Class) Operation Parameter - Changed defaultValue

Change Code: CCOPCdV

Description: There exists a set of (class operation) parameters in the model such that the defaultValue property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.classClassifier.operation.parameter->select(p:Parameter|
not p.defaultValue.equals(self.model.application.originalModel.
classDiagramView.classClassifier->select(c:ClassClassifier|
c.getIDStr() = p.operation.classClassifier.getIDStr()).
operation->select(o:Operation|o.getIDStr() =
p.operation.getIDStr()).parameter->asSequence->at(p.
operation.getParameterAtPosition(p)).defaultValue))`

80. Changed (Class) Operation Parameter - Changed direction

Change Code: CCOPCd

Description: There exists a set of (class operation) parameters in the model such that the direction property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.classClassifier.operation.parameter->select(p:Parameter|
p.direction <> self.model.application.originalModel.
classDiagramView.classClassifier->select(c:ClassClassifier|
c.getIDStr() = p.operation.classClassifier.getIDStr()).
operation->select(o:Operation|o.getIDStr() =
p.operation.getIDStr()).parameter->asSequence->at(p.
operation.getParameterAtPosition(p)).direction)`

81. Changed (Class) Operation Parameter - Changed name

Change Code: CCOPCn

Description: There exists a set of (class operation) parameters in the model such that the name property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.classClassifier.operation.parameter->select(p:Parameter|
p.name <> self.model.application.originalModel.
classDiagramView.classClassifier->select(c:ClassClassifier|
c.getIDStr() = p.operation.classClassifier.getIDStr()).
operation->select(o:Operation|o.getIDStr() =
p.operation.getIDStr()).parameter->asSequence->at(p.
operation.getParameterAtPosition(p)).name)`

82. Changed (Interface) Operation Parameter - Changed defaultValue

Change Code: CIOPCdV

Description: There exists a set of (interface operation) parameters in the model such that the defaultValue property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.interface.operation.parameter->select(p:Parameter|
not p.defaultValue.equals(self.model.application.originalModel.
classDiagramView.interface->select(i:Interface|
i.getIDStr() = p.operation.interface.getIDStr()).
operation->select(o:Operation|o.getIDStr() =
p.operation.getIDStr()).parameter->asSequence->at(p.
operation.getParameterAtPosition(p)).defaultValue)`

83. Changed (Interface) Operation Parameter - Changed direction

Change Code: CIOPCd

Description: There exists a set of (interface operation) parameters in the model such that the direction property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.interface.operation.parameter->select(p:Parameter|
p.direction <> self.model.application.originalModel.
classDiagramView.interface->select(i:Interface|
i.getIDStr() = p.operation.interface.getIDStr()).
operation->select(o:Operation|o.getIDStr() =
p.operation.getIDStr()).parameter->asSequence->at(p.
operation.getParameterAtPosition(p)).direction)`

84. Changed (Interface) Operation Parameter - Changed name

Change Code: CIOPCn

Description: There exists a set of (interface operation) parameters in the model such that the name property of each one is not the same in both model versions.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.interface.operation.parameter->select(p:Parameter|
p.name <> self.model.application.originalModel.
classDiagramView.interface->select(i:Interface|
i.getIDStr() = p.operation.interface.getIDStr()).
operation->select(o:Operation|o.getIDStr() =
p.operation.getIDStr()).parameter->asSequence->at(p.
operation.getParameterAtPosition(p)).name)`

85. Changed State - Added doActivity

Change Code: CSAdA

Description: There exists a set of states in the model such that each one has an activity in the changed model version and none in the original version.

OCL Expression: `context model::behaviouralElements::stateMachines::
StatechartDiagramView
self.state->select(s1:State|s1.doActivity->size = 1 and self.
model.application.originalModel.statechartDiagramView.
state->select(s2:State|s2.getIDStr() = s1.getIDStr()).
doActivity->size = 0)`

86. Changed State - Deleted doActivity

Change Code: CSDdA

Description: There exists a set of states in the model such that each one has an activity in the original model version and none in the changed version.

OCL Expression: `context model::behaviouralElements::stateMachines::
StatechartDiagramView
self.state->select(s1:State|s1.doActivity->size = 0 and self.
model.application.originalModel.statechartDiagramView.
state->select(s2:State|s2.getIDStr() = s1.getIDStr()).
doActivity->size = 1)`

87. Changed State - Added entry

Change Code: CSAe

Description: There exists a set of states in the model such that each one has an entry action in the changed model version and none in the original version.

OCL Expression: `context model::behaviouralElements::stateMachines::
StatechartDiagramView
self.state->select(s1:State|s1.entry->size = 1 and self.
model.application.originalModel.statechartDiagramView.
state->select(s2:State|s2.getIDStr() = s1.getIDStr()).
entry->size = 0)`

88. Changed State - Deleted entry

Change Code: CSDe

Description: There exists a set of states in the model such that each one has an entry action in the original model version and none in the changed version.

OCL Expression: `context model::behaviouralElements::stateMachines::
StatechartDiagramView
self.state->select(s1:State|s1.entry->size = 0 and self.
model.application.originalModel.statechartDiagramView.
state->select(s2:State|s2.getIDStr() = s1.getIDStr()).
entry->size = 1)`

89. Changed State - Added exit

Change Code: CSAex

Description: There exists a set of states in the model such that each one has an exit action in the changed model version and none in the original version.

OCL Expression: `context model::behaviouralElements::stateMachines::
StatechartDiagramView
self.state->select(s1:State|s1.exit->size = 1 and self.
model.application.originalModel.statechartDiagramView.
state->select(s2:State|s2.getIDStr() = s1.getIDStr()).
exit->size = 0)`

90. Changed State - Deleted exit

Change Code: CSDex

Description: There exists a set of states in the model such that each one has an exit action in the original model version and none in the changed version.

OCL Expression: `context model::behaviouralElements::stateMachines::
StatechartDiagramView
self.state->select(s1:State|s1.exit->size = 0 and self.
model.application.originalModel.statechartDiagramView.
state->select(s2:State|s2.getIDStr() = s1.getIDStr()).
exit->size = 1)`

91. Changed State - Added internalTransition

Change Code: CSAiT

Description: There exists a set of states in the model such that each one has at least one internal transition in the changed model version that it didn't have in the original version.

OCL Expression: `context model::behaviouralElements::stateMachines::
StatechartDiagramView
self.state->select(s1:State|
s1.internalTransition->exists(it1:Transition|not self.
model.application.originalModel.statechartDiagramView.
state->select(s2:State|s2.getIDStr() = s1.getIDStr()).
internalTransition->exists(it2:Transition|
it1.getIDStr() = it2.getIDStr())))`

92. Changed State - Deleted internalTransition

Change Code: CSDiT

Description: There exists a set of states in the model such that each one has at least one internal transition in the original model version that it doesn't have in the changed version.

OCL Expression: `context model::behaviouralElements::stateMachines::
StatechartDiagramView
self.model.application.originalModel.statechartDiagramView.
state->select(s1:State|s1.
internalTransition->exists(it1:Transition|not self.
state->select(s2:State|s2.getIDStr() = s1.getIDStr()).
internalTransition->exists(it2:Transition|
it1.getIDStr() = it2.getIDStr())))`

93. Changed State - Changed invariant

Change Code: CSCi

Description: There exists a set of states in the model such that the invariant property of each one is not the same in both model versions.

OCL Expression: `context model::behaviouralElements::stateMachines::
StatechartDiagramView
self.state->select(s1:State|not s1.
invariant.equals(self.model.application.originalModel.
statechartDiagramView.state->select(s2:State|
s1.getIDStr() = s2.getIDStr()).invariant))`

The following two definitions are used in rules 94 to 97 inclusive:

```
context model::behaviouralElements::stateMachines::StatechartDiagramView
let smActions:Action = self.state.action->union(self.transition.action)
let originalSMActions:Action = self.model.application.originalModel.
    statechartDiagramView.state.action->union(self.model.application.
    originalModel.statechartDiagramView.transition.action)
```

94. Changed StateMachine action - Added (Discrete) action

Change Code: CSMAADa

Description: There exists a set of state machine action sequences such that in the changed model version each one has a discrete action that it doesn't have in the original version.

OCL Expression: **context** model::behaviouralElements::stateMachines::
StatechartDiagramView
smActions->select(as1:ActionSequence|as1.
action->exists(a1:Action|not
originalSMActions->select(as2:ActionSequence|
as1.getIDStr() = as2.getIDStr()).
action->exists(a2:Action|a1.getIDStr() = a2.getIDStr()))

95. Changed StateMachine action - Deleted (Discrete) action

Change Code: CSMADDa

Description: There exists a set of state machine action sequences such that in the original model version each one has a discrete action that it doesn't have in the changed version.

OCL Expression: **context** model::behaviouralElements::stateMachines::
StatechartDiagramView
originalSMActions->select(as1:ActionSequence|as1.
action->exists(a1:Action|not
smActions->select(as2:ActionSequence|
as1.getIDStr() = as2.getIDStr()).
action->exists(a2:Action|a1.getIDStr() = a2.getIDStr()))

96. Changed StateMachine action - Changed recurrence

Change Code: CSMAcR

Description: There exists a set of state machine action sequences such that the recurrence property of each one is not the same in both model versions.

OCL Expression: **context** model::behaviouralElements::stateMachines::
StatechartDiagramView
smActions->select(as1:ActionSequence|not as1.recurrence.
equals(originalSMActions->select(as2:ActionSequence|
as1.getIDStr() = as2.getIDStr()).recurrence))

97. Changed StateMachine action - Changed script

Change Code: CSMAcS

Description: There exists a set of state machine action sequences such that the script property of each one is not the same in both model versions.

OCL Expression: **context** model::behaviouralElements::stateMachines::
StatechartDiagramView
smActions->select(as1:ActionSequence|not as1.script.
equals(originalSMActions->select(as2:ActionSequence|
as1.getIDStr() = as2.getIDStr()).script))

Appendix B – Impact Analysis Rules

The impact analysis rules defined in this work are listed below. The format of the rules is described in Table 1. Note that each element that is identified by the rules is checked by the tool to ensure that it is present in both model versions before it is considered impacted. The following operations are used in the impact analysis rules defined below.

<code>getOtherEnd:</code>	Defined on an association. Given the id of an association end, it returns the other end of the current association.
<code>containsNavigation:</code>	Defined on an invariant. Given the target class and an association, it determines if the current invariant contains a navigation to the target class via the association.
<code>getProperty:</code>	Defined on each model element that requires it. Returns the property of the current model element that matches the given <code>propertyID</code> .
<code>getInvokingOperations:</code>	Defined on an operation. Returns the operations in the sequence and statechart diagrams that invoke the current operation.

The following definition is used in some of the rules below.

```
def:  
  let null:ModelElement = Set()
```

1. Changed ClassDiagramView - Added Association

Change Code: CCDVAA

Changed Element: model::foundation::core::ClassDiagramView

Added Property: model::foundation::core::Association

Impacted Element: model::foundation::core::ClassClassifier

Description: The participant of each end of the added association is impacted if the opposite end is navigable.

Rationale: Each impacted class has gained a navigable association.

Resulting Change: The class invariant and operation contracts (preconditions and postconditions) may have to be updated to reflect this change. In addition, additional operations may have to be defined, and/or methods (implementations of the operations) updated to reflect the change. In addition, a variable storing reference(s) to objects of the class at the opposite end may have to be added to the class's implementation.

OCL Expression:

```
context modelChanges::Change
let addedAssociation:Association = self.changedElement.
oclAsType(ClassDiagramView).
association->select(a:Association|
a.getIDStr() = self.propertyID)
addedAssociation.end->select(e:AssociationEnd|
addedAssociation.getOtherEnd(e.getIDStr()).
isNavigable = true).participant
```

2. Changed ClassDiagramView - Deleted Association

Change Code: CCDVDA

Changed Element: model::foundation::core::ClassDiagramView

Deleted Property: model::foundation::core::Association

Impacted Element: model::foundation::core::ClassClassifier

Description: Each class that is a participant, in the original version, of one of the deleted association's ends is impacted if the opposite end is navigable.

Rationale: Each impacted class has lost a navigable association.

Resulting Change: Methods (implementations of the operations) may have to be updated and/or deleted to reflect the change. In addition, the variable storing reference(s) to objects of the class at the opposite end may have to be deleted from the class's implementation. Note that since the model is assumed to be consistent the class invariant and operation contracts in each impacted class should not contain navigation expressions to the class that uses the deleted association.

OCL Expression:

```
context modelChanges::Change
let deletedAssociation:Association = self.changedElement.
oclAsType(ClassDiagramView).
association->select(a:Association|
a.getIDStr() = self.propertyID)
deletedAssociation.end->select(e:AssociationEnd|
addedAssociation.getOtherEnd(e.getIDStr()).
isNavigable = true).participant->select(p:ClassClassifier|
p.view.model.application.changedModel.classDiagramView.
classClassifier->exists(c:ClassClassifier|
c.getPathname() = p.getPathname()))
```

3. Changed ClassDiagramView - Added ClassClassifier

Change Code: CCDVAC

Changed Element: model::foundation::core::ClassDiagramView

Added Property: model::foundation::core::ClassClassifier

Description: There is no impact since the model is assumed to be consistent.

4. Changed ClassDiagramView - Deleted ClassClassifier

Change Code: CCDVDC

Changed Element: model::foundation::core::ClassDiagramView

Deleted Property: model::foundation::core::ClassClassifier

Impacted Elements: model::foundation::core::ClassClassifier
model::foundation::core::Interface

Description: Each class that had a navigable association, or dependency relationship (as a client) with the deleted class is impacted. The descendants of the deleted class are also impacted. The interfaces that had a dependency relationship (as a client) with the deleted class are also impacted.

Rationale: Each of the impacted classes and interfaces can no longer access the services of the deleted class (directly or indirectly).

Resulting Changes: The implementation of the impacted classes may have to be modified. This modification may include the deletion of the variable that stores the reference to objects of the deleted class. In addition, methods may have to be modified to reflect the change. The implementation of the impacted interfaces may have to be modified to reflect the change. Note that since the model is assumed to be consistent the class invariant and operation contracts in each impacted class should not contain navigation expressions to the deleted class. In addition, since the model is assumed to be consistent, there should be no parameter or attribute that has the deleted class as its type.

OCL Expressions:

```
context modelChanges::Change def:
    let deletedClass:ClassClassifier = self.changedElement.
        oclAsType(ClassDiagramView).
        classClassifier->select(c:ClassClassifier|
            c.getIDStr() = self.propertyID)

context modelChanges::Change -- associated classes
    let deletedClassEnd:AssociationEnd = deletedClass.
        associationEnd

    if deletedClassEnd.isNavigable = true then
        deletedClassEnd.association.getOtherEnd(deletedClassEnd.
            getIDStr()).participant
    else
        null
    endif

context modelChanges::Change -- dependent classifiers
    deletedClass.clientDependency.client

context modelChanges::Change -- subclasses
    deletedClass.specialization.child
```

5. Changed ClassDiagramView - Added Dependency

Change Code: CCDVAD

Changed Element: model::foundation::core::ClassDiagramView

Added Property: model::foundation::core::Dependency

Impacted Elements: model::foundation::core::ClassClassifier
model::foundation::core::Interface

Description: The client classifier (class or interface) is impacted.

Rationale: The impacted classifier now has a dependency relationship with another classifier.

Resulting Change: The implementation of the impacted classifier may have to be modified to reflect the change. This modification may include, for instance, updating methods (in the case of classes) to reflect the change.

OCL Expression:

```
context modelChanges::Change
    self.changedElement.oclAsType(ClassDiagramView).
    dependency->select(d:Dependency|
        d.getIDStr() = self.propertyID).client
```

6. Changed ClassDiagramView - Deleted Dependency

Change Code: CCDVDD

Changed Element: model::foundation::core::ClassDiagramView

Deleted Property: model::foundation::core::Dependency

Impacted Elements: model::foundation::core::ClassClassifier
model::foundation::core::Interface

Description: The classifier (class or interface) that was the client in the deleted dependency is impacted.

Rationale: The impacted classifier has now lost its dependency relationship with another classifier.

Resulting Change: Same as that of CCDVAD.

OCL Expression: -- Same as that of CCDVAD.

7. Changed ClassDiagramView - Added Generalization

Change Code: CCDVAG

Changed Element: model::foundation::core::ClassDiagramView

Added Property: model::foundation::core::Generalization

Impacted Elements: model::foundation::core::ClassClassifier
model::foundation::core::Interface

Description: The child classifier (class or interface) is impacted.

Rationale: The impacted classifier is now a sub-classifier of another classifier.

Resulting Change: The implementation of the impacted classifier may have to be modified, for example, in the case of a class classifier, the class declaration code may have to specify that the class is now a subclass of another class.

OCL Expression:

```
context modelChanges::Change
self.changedElement.oclaType(ClassDiagramView).
generalization->select(g:Generalization|
g.getIDStr() = self.propertyID).child
```

8. Changed ClassDiagramView - Deleted Generalization

Change Code: CCDVDG

Changed Element: model::foundation::core::ClassDiagramView

Deleted Property: model::foundation::core::Generalization

Impacted Elements: model::foundation::core::ClassClassifier
model::foundation::core::Interface

Description: Same as that of CCDVAG.

Rationale: The impacted classifier is no longer a sub-classifier of its former super-classifier.

Resulting Changes: The implementation of the impacted classifier may have to be modified, for example, in the case of a class classifier, the class declaration code may have to be modified to reflect the change.

OCL Expression: -- Same as that of CCDVAG.

9. Changed ClassDiagramView - Added Interface

Change Code: CCDVAI

Changed Element: model::foundation::core::ClassDiagramView

Added Property: model::foundation::core::Interface

Description: There is no impact since the model is assumed to be consistent.

10. Changed ClassDiagramView - Deleted Interface

Change Code:	CCDVIDI
Changed Element:	model::foundation::core::ClassDiagramView
Deleted Property:	model::foundation::core::Interface
Impacted Elements:	model::foundation::core::ClassClassifier model::foundation::core::Interface
Description:	Each class that had a navigable association, or dependency (as a client) relationship with the deleted interface is impacted. The classes that realized the deleted interface are impacted as well. In addition, the descendants of the deleted interface as well as the interfaces that had a dependency relationship (as a client) with the deleted interface are also impacted.
Rationale:	Each of the impacted classes can no longer access the services of the deleted interface (directly or indirectly). The impacted interfaces are no longer sub-interfaces of the deleted interface – in the case of the former child interfaces; and for the other impacted interfaces, they can no longer access the services of the deleted interface.
Resulting Changes:	The implementation of the impacted classes may have to be modified. This modification may include the deletion of the variable that stores the reference to objects of the classes that realized the deleted interface. In addition, methods may have to be modified to reflect the change. The implementation of the interfaces that inherited from the deleted interface may have to be modified. This modification may include the interface declaration code, for example.
OCL Expressions:	<pre>context modelChanges::Change def: let deletedInterface:Interface = self.changedElement. oclAsType(ClassDiagramView).oclAsType(ClassDiagramView). interface->select(i:Interface i.getIDStr() = self.propertyID) context modelChanges::Change -- associated classes let deletedInterfaceEnd:AssociationEnd = deletedInterface. specifiedEnd if deletedInterfaceEnd.isNavigable = true then deletedInterfaceEnd.association. getOtherEnd(deletedInterfaceEnd.getIDStr()).participant else null endif context modelChanges::Change -- dependent classes and interfaces deletedInterface.clientDependency.client context modelChanges::Change -- sub-interfaces deletedInterface.specialization.child</pre>

11. Changed ClassDiagramView - Added Realization

Change Code:	CCDVAR
Changed Element:	model::foundation::core::ClassDiagramView
Added Property:	model::foundation::core::Realization
Impacted Element:	model::foundation::core::ClassClassifier
Description:	The implementation class is impacted.
Rationale:	The impacted class now has to implement all the operations in the interface (specification).
Resulting Change:	The implementation of the impacted class may have to be modified to define the methods that implement the interface's operations.
OCL Expression:	<pre>context modelChanges::Change self.changedElement.oclAsType(ClassDiagramView). realization->select(r:Realization r.getIDStr() = self.propertyID).implementation</pre>

12. Changed ClassDiagramView - Deleted Realization

Change Code: CCDVDR

Changed Element: model::foundation::core::ClassDiagramView

Deleted Property: model::foundation::core::Realization

Impacted Element: model::foundation::core::ClassClassifier

Description: The class that was the implementation class in the realization relationship is impacted.

Rationale: The impacted class no longer realizes the deleted interface.

Resulting Change: The implementation of the impacted class may have to be modified, for example, the class declaration code may have to be modified to reflect the change.

OCL Expression: -- Same as that of CCDVAR.

13. Changed SequenceDiagramView - Added ClassifierRole

Change Code: CSDVACR

Changed Element: model::behaviouralElements::collaborations::SequenceDiagramView

Added Property: model::behaviouralElements::collaborations::ClassifierRole

Description: There is no impact since the model is assumed to be consistent.

14. Changed SequenceDiagramView - Deleted ClassifierRole

Change Code: CSDVDCR

Changed Element: model::behaviouralElements::collaborations::SequenceDiagramView

Deleted Property: model::behaviouralElements::collaborations::ClassifierRole

Description: There is no impact since the model is assumed to be consistent.

15. Changed SequenceDiagramView - Added Message

Change Code: CSDVAM

Changed Element: model::behaviouralElements::collaborations::SequenceDiagramView

Added Property: model::behaviouralElements::collaborations::Message

Impacted Elements: model::foundation::core::ClassClassifier
model::foundation::core::Operation
model::foundation::core::Postcondition

Description: The base class of the classifier role that sends the added message is impacted. The operation that sends the added message is impacted, and its postcondition is also impacted.

Rationale: The sending/source class now sends a new message and one of its operations, actually sending the added message, is impacted. This operation is known or not, depending on whether the message triggering the added message corresponds to an invoked operation. If, for example, it is a signal then the operation may not be known just by looking at the sequence diagram. The impacted postcondition may now not represent the effect (what is true on completion) of its operation.

Resulting Changes: The implementation of the base class may have to be modified. The method (implementation) of the impacted operation may have to be modified. The impacted postcondition should be checked to ensure that it is still valid.

Invoked Rule: Changed Class Operation – Changed Postcondition (CCOCpst)

OCL Expressions:

```
context modelChanges::Change def:
  let addedMessage:Message = self.changedElement.
    oclAsType(SequenceDiagramView).
    message->select(m:Message|m.getIDStr() = self.propertyID)

  let sendingOperation:Operation =
    (if addedMessage.activator.action.ocIsTypeOf(CallAction)
     then
       addedMessage.sender.base.ocIsTypeOf(ClassClassifier).
       operation->select(o:Operation|o.equals(addedMessage.
         activator.action.ocIsTypeOf(CallAction).operation))
     else
       null
     endif)

context modelChanges::Change -- class
  if not addedMessage.activator.ocIsTypeOf(CallAction) then
    addedMessage.sender.base.ocIsTypeOf(ClassClassifier)
  else
    null
  endif

context modelChanges::Change -- operation
  sendingOperation

context modelChanges::Change -- postcondition
  sendingOperation.postcondition
```

16. Changed SequenceDiagramView - Deleted Message

Change Code: CSDVDM
Changed Element: model::behaviouralElements::collaborations::SequenceDiagramView
Deleted Property: model::behaviouralElements::collaborations::Message
Impacted Elements: model::foundation::core::ClassClassifier
model::foundation::core::Operation
model::foundation::core::Postcondition
Description: The base class of the classifier role that sent the deleted message is impacted. The operation, that sent the deleted message is impacted, and its postcondition is also impacted.
Rationale: The impacted class now sends one less message. The impacted operation also now sends one less message. The impacted postcondition may now not represent the effect of its operation.
Resulting Changes: The implementation of the base class may have to be modified. The method of the impacted operation may have to be modified. The impacted postcondition should be checked to ensure that it is still valid.
Invoked Rule: Changed Class Operation – Changed Postcondition (CCOCpst)
OCL Expressions: -- Same as that of CSDVAM.

17. Changed StateDiagramView - Added CompositeState

Change Code: CStDVACS
Changed Element: model::behaviouralElements::stateMachines::StatechartDiagramView
Added Property: model::behaviouralElements::stateMachines::CompositeState
Impacted Element: model::foundation::core::ClassClassifier
Description: The context class, of the state machine to which the added composite state belongs, is impacted.
Rationale: The impacted class's state machine now has one more state.
Resulting Change: The implementation of the impacted class (or class cluster to which it belongs) may have to be modified to account for the new state.
OCL Expression:

```
context modelChanges::Change
self.changedElement.oclAsType(StatechartDiagramView).
compositeState->select(cs:CompositeState|
cs.getIDStr() = self.propertyID).stateMachine.context
```

18. Changed StateDiagramView - Deleted CompositeState

Change Code: CStDVDCS
Changed Element: model::behaviouralElements::stateMachines::StatechartDiagramView
Deleted Property: model::behaviouralElements::stateMachines::CompositeState
Impacted Element: model::foundation::core::ClassClassifier
Description: The context class, of the state machine to which the deleted composite state belonged, is impacted.
Rationale: The impacted class's state machine now has one less state.
Resulting Change: The implementation of the impacted class (or class cluster to which it belongs) may have to be modified to account for the deleted state.
OCL Expression: -- Same as that of CStDVACS.

19. Changed StateDiagramView - Added SimpleState

Change Code: CStDVASS
Changed Element: model::behaviouralElements::stateMachines::StatechartDiagramView
Added Property: model::behaviouralElements::stateMachines::SimpleState
Impacted Element: model::foundation::core::ClassClassifier
Description: The context class, of the state machine to which the added simple state belongs, is impacted.
Rationale: The impacted class's state machine now has one more state.
Resulting Change: The implementation of the impacted class (or class cluster to which it belongs) may have to be modified to account for the new state.
OCL Expression:

```
context modelChanges::Change
self.changedElement.oclaAsType(StatechartDiagramView).
simpletState->select(ss:SimpleState|
ss.getIDStr() = self.propertyID).stateMachine.context
```

20. Changed StateDiagramView - Deleted SimpleState

Change Code: CStDVDSS
Changed Element: model::behaviouralElements::stateMachines::StatechartDiagramView
Deleted Property: model::behaviouralElements::stateMachines::SimpleState
Impacted Element: model::foundation::core::ClassClassifier
Description: The context class, of the state machine to which the deleted simple state belonged, is impacted.
Rationale: The impacted class's state machine now has one less state.
Resulting Change: The implementation of the impacted class (or class cluster to which it belongs) may have to be modified to account for the deleted state.
OCL Expression: -- Same as that of CStDVASS.

21. Changed StateDiagramView - Added Transition

Change Code: CStDVAT
Changed Element: model::behaviouralElements::stateMachines::StatechartDiagramView
Added Property: model::behaviouralElements::stateMachines::Transition
Impacted Element: model::foundation::core::ClassClassifier
Description: The context class, of the state machine to which the added transition belongs, is impacted.
Rationale: The impacted class's state machine now has one more transition.
Resulting Change: The implementation of the impacted class (or class cluster to which it belongs) may have to be modified to account for the new transition.
OCL Expression:

```
context modelChanges::Change
self.changedElement.oclaAsType(StatechartDiagramView).
transition->select(t:Transition|
t.getIDStr() = self.propertyID).stateMachine.context
```

22. Changed StateDiagramView - Deleted Transition

Change Code: CStDVDT

Changed Element: model::behaviouralElements::stateMachines::
StatechartDiagramView

Deleted Property: model::behaviouralElements::stateMachines::Transition

Impacted Element: model::foundation::core::ClassClassifier

Description: The context class, of the state machine to which the deleted transition belonged, is impacted.

Rationale: The impacted class's state machine now has one less transition.

Resulting Change: The implementation of the impacted class (or class cluster to which it belongs) may have to be modified to account for the deleted transition.

OCL Expression: -- Same as that of CStDVAT.

23. Changed AssociationEnd - Changed aggregation

Change Code: CAECa

Changed Element: model::foundation::core::AssociationEnd

Changed Property: aggregation

Impacted Element: model::foundation::core::ClassClassifier

Description: There is no impact when aggregation is changed from "no aggregation" to "aggregation" since the model is assumed to be consistent. The association end's participant (in the meta-model) is impacted when aggregation is changed from "no aggregation" to "composition". There is no impact when aggregation is changed from "aggregation" to "no aggregation". The association end's participant (in the meta-model) is impacted when aggregation is changed from "aggregation" to "composition". The association end's participant (in the meta-model) is impacted when aggregation is changed from "composition" to "no aggregation". The association end's participant (in the meta-model) is impacted when aggregation is changed from "composition" to "aggregation".

Rationale: When aggregation is changed to "composition" then the participant is impacted because it now has to handle the life time issues of the parts, i.e. the parts live and die with the composite, and so the impacted class has to account for this. However, when aggregation is changed from "composition", then the participant is impacted because now it is no longer responsible for the lifetime issues of the parts and thus the class (participant) may have to be changed to reflect this.

Resulting Change: Certain methods of the impacted class may have to be changed to reflect the change in the lifetime dependencies of the parts in the composition relationship, for example, constructors and destructors may have to be changed.

OCL Expression:

```
context modelChanges::Change
let changedEnd:AssociationEnd = self.changedElement.
oclAsType(AssociationEnd)

if (changedEnd.aggregation = #composite) or (changedEnd.
    association.view.model.application.originalModel.
    classDiagramView.association->select(a:Association|
    a.getIDStr() = changedEnd.association.getIDStr()).
    end->select(e:AssociationEnd|e.getIDStr() =
    changedEnd.getIDStr()).aggregation = #composite) then
    changedEnd.participant
else
    null
endif)
```

24. Changed AssociationEnd - Changed changeability

Change Code: CAECc

Changed Element: model::foundation::core::AssociationEnd

Changed Property: changeability

Impacted Elements: model::foundation::core::ClassClassifier
model::foundation::core::Operation

Description: The class at the unchanged association end is impacted if the changed association end is navigable. The impacted operations are those in the class at the unchanged association end, such that the postcondition of each of these operations contains at least one navigation (via the changed association) to the class at the changed association end.

Rationale: The criterion for adding, modifying and deleting links to objects of the class at the changed association end has changed and thus the method of each impacted operation may have to be changed to reflect this change. The implementation of the impacted class may have to be changed.

Resulting Changes: The implementation (method) of each impacted operation should be checked to ensure that it does not violate the new changeability criterion. If changeability was changed to "changeable" then the methods of the impacted operations may have to be modified to allow for the addition, modification or deletion of links to objects of the class at the changed association end. If changeability was changed from "frozen" to "addOnly" then the methods of the impacted operations may have to be modified to allow for the addition of links to objects of the class at the changed association end. In addition, additional operations may have to be defined (and implemented) in the impacted class. The postconditions of some operations in the impacted class may not show a navigation to the class at the changed association end even though their methods have such navigations. The implementation of the impacted class thus has to be checked for the occurrences of these methods.

OCL Expressions:

```
context modelChanges::Change Def:
  let affectedClass:ClassClassifier = self.changedElement.
    oclAsType(AssociationEnd).association.
    getOtherEnd(self.changedElement.oclAsType(AssociationEnd).
    getIDStr()).participant

  context modelChanges::Change -- operations
    affectedClass.operation->select(o:Operation|o.postcondition.
    containsNavigation(self.changedElement.
    oclAsType(AssociationEnd).participant,
    self.changedElement.oclAsType(AssociationEnd).association))

  context modelChanges::Change -- class
    if self.changedElement.oclAsType(AssociationEnd).isNavigable =
      true then
      affectedClass
    else
      null
    endif
```

25. Changed AssociationEnd - Added interfaceSpecifier

Change Code: CAEAiS

Changed Element: model::foundation::core::AssociationEnd

Added Property: interfaceSpecifier

Description: There is no impact since the model is assumed to be consistent.

26. Changed AssociationEnd - Changed interfaceSpecifier

Change Code: CAECiS

Changed Element: model::foundation::core::AssociationEnd

Changed Property: interfaceSpecifier

Description: "Changed interfaceSpecifier" refers to a change in the classifier's specification. For example, if the interface specifier is an interface, then an added operation is treated as a changed specification. There is no impact since the model is assumed to be consistent.

27. Changed AssociationEnd - Deleted interfaceSpecifier

Change Code: CAEDiS

Changed Element: model::foundation::core::AssociationEnd

Deleted Property: interfaceSpecifier

Description: There is no impact – the original services are still available and have not changed. Note that an association end interface specifier specifies the subset of the functionalities of a classifier that are needed in the association. Therefore, deleting the interface specifier does not result in any impacts.

28. Changed AssociationEnd - Changed isNavigable

Change Code: CAECiN

Changed Element: model::foundation::core::AssociationEnd

Changed Property: isNavigable

Impacted Element: model::foundation::core::ClassClassifier

Description: If isNavigable equals true then the class at the unchanged association end is impacted.

Rationale: The impacted class is now able to navigate to the classifier at the unchanged association end via the changed association.

Resulting Change: The existing operations of the impacted class may have to be modified and/or new operations defined to access link(s) to the class at the changed association end.

OCL Expression:

```
context modelChanges::Change
if self.changedElement.oclAsType(AssociationEnd).isNavigable =
    true then
    self.changedElement.oclAsType(AssociationEnd).association.
        getOtherEnd(self.changedElement.oclAsType(AssociationEnd).
            getIDStr()).participant
else
    null
endif
```


29. Changed AssociationEnd - Changed multiplicity

Change Code: CAECm

Changed Element: model::foundation::core::AssociationEnd

Changed Property: multiplicity

Impacted Elements: model::foundation::core::ClassClassifier
model::foundation::core::Invariant
model::foundation::core::Operation
model::foundation::core::Operation

Description: The class at the unchanged association end is impacted if the changed association end is navigable. The invariant of the class at the unchanged association end is impacted if it contains any navigation (via the changed association) to the class at the changed association end. The bag of impacted operations is such that each impacted operation in the bag belongs to the class at the unchanged association end, and each impacted operation's pre/postcondition contains at least one navigation (via the changed association) to the class at the changed association end. The impacted pre/postconditions are those of the impacted operations that contain navigations (via the changed association) to the class at the changed association end.

Rationale: The impacted class's implementation may have to be changed. The impacted invariant and pre/postconditions may be accessing links that no longer exist or may have to be modified to account for the new number of links. The method of each impacted operation may have to be modified.

Resulting Changes: The method of each impacted element may have to be modified to reflect the changed multiplicity, i.e. the changed multiplicity affects the number of accessible links. It may also affect how the links are accessed. The implementation of the impacted class may have to be changed to reflect the change. For example, if the multiplicity was changed from 1 to *, then the implementation needs to define an attribute that has a multiplicity greater than 1 to store the links to objects of the class at the changed association end. In addition, new operations may have to be defined to facilitate access to the new number of links.

Invoked Rules: Changed Class - Changed Invariant (CCCI).
Changed Class Operation - Changed Precondition (CCOCpre)
Changed Class Operation - Changed Postcondition (CCOCpst)

OCL Expressions:

```

context modelChanges::Change Def:
    let affectedClass:ClassClassifier = self.changedElement.
        association.getOtherEnd(self.changedElement.getIDStr()).
        participant

context modelChanges::Change -- class
    if self.changedElement.isNavigable = true then
        affectedClass
    else
        null
    endif

context modelChanges::Change -- invariant
    if affectedClass.invariant.containsNavigation(self.
        changedElement.participant, self.changedElement.
        association) then
        affectedClass.invariant
    else
        null
    endif

context modelChanges::Change -- operations
    affectedClass.operation->select(o:Operation|o.postcondition.
        containsNavigation(self.changedElement.participant, self.
        changedElement.association) or o.precondition.
        containsNavigation(self.changedElement.participant,
        self.changedElement.association))

context modelChanges::Change -- preconditions
    affectedClass.operation.precondition->select(pr:Precondition|
        pr.containsNavigation(self.changedElement.participant,
        self.changedElement.association))

context modelChanges::Change -- postconditions
    affectedClass.operation.postcondition->select(ps:Postcondition|
        ps.containsNavigation(self.changedElement.participant,
        self.changedElement.association))

```

30. Changed AssociationEnd - Changed ordering

Change Code: CAECo

Changed Element: model::foundation::core::AssociationEnd

Changed Property: ordering

Impacted Elements: model::foundation::core::ClassClassifier
model::foundation::core::Operation

Description: The class at the unchanged association end is impacted if the changed association end is navigable. The bag of impacted operations is such that each impacted operation in the bag belongs to the class (impacted class) at the unchanged association end, and each impacted operation's pre/postcondition contains at least one navigation (via the changed association) to the class at the changed association end.

Rationale: The method of each impacted operation may have to be modified to reflect the new ordering criterion. For example, if the method adds links then it (the method) may have to be changed to reflect the new ordering criterion. Additional operations may have to be defined to implement the new ordering criterion. In addition, a data type change may be required for the variable (in the implementation of the impacted class) that stores the links to the objects of the class at the changed association end.

Resulting Changes: The impacted class's implementation may have to be modified. The method of each impacted operation may have to be modified.

OCL Expressions:

```
context modelChanges::Change Def:
    let affectedClass:ClassClassifier = self.changedElement.
        association.getOtherEnd(self.changedElement.getIDStr()).
        participant

context modelChanges::Change -- class
    if self.changedElement.isNavigable = true then
        affectedClass
    else
        null
    endif

context modelChanges::Change -- operations
    affectedClass.operations->select(o:Operation|o.postcondition.
        containsNavigation(self.changedElement.participant,
        self.changedElement.association) or o.precondition.
        containsNavigation(self.changedElement.participant,
        self.changedElement.association))
```

31. Changed AssociationEnd - Added qualifier

Change Code: CAEAq

Changed Element: model::foundation::core::AssociationEnd

Added Property: qualifier

Impacted Elements: model::foundation::core::ClassClassifier
model::foundation::core::Operation

Description: Same as that of CAECo.

Rationale: The method of each impacted operation may have to be changed so that it now uses the added qualifier in link selection. Data structure(s) may have to be defined to facilitate the new lookup feature that the added qualifier introduces. In addition, new operations may have to be defined (and implemented) to facilitate the change.

Resulting Changes: Same as that of CAECo.

OCL Expressions: -- Same as that of CAECo.

32. Changed AssociationEnd - Changed qualifier type

Change Code: CAECqt

Changed Element: model::foundation::core::AssociationEnd

Changed Property: qualifier

Impacted Elements: model::foundation::core::ClassClassifier
model::foundation::core::Operation

Description: Same as that of CAECq.

Rationale: The method of each impacted operation may have to be changed so that it is now consistent with the changed qualifier type. The implementation of the qualifier may have to be changed.

Resulting Changes: Same as that of CAECq.

OCL Expressions: -- Same as that of CAECq.

33. Changed AssociationEnd - Deleted qualifier

Change Code: CAEDq

Changed Element: model::foundation::core::AssociationEnd

Deleted Property: qualifier

Impacted Elements: model::foundation::core::ClassClassifier
model::foundation::core::Operation

Description: Same as that of CAECq.

Rationale: The method of each impacted operation may have to be changed so that it doesn't use the deleted qualifier in link selection. Certain operations defined specifically for the deleted qualifier may have to be deleted. The declaration of the data structure used for the deleted qualifier may have to be deleted.

Resulting Changes: Same as that of CAECq.

OCL Expressions: -- Same as that of CAECq.

34. Changed AssociationEnd - Changed targetScope

Change Code: CAECtS

Changed Element: model::foundation::core::AssociationEnd

Changed Property: targetScope

Impacted Elements: model::foundation::core::ClassClassifier
model::foundation::core::Operation

Description: Same as that of CAECq.

Rationale: The method of each impacted operation may have to be changed so that it is now consistent with the changed target scope. The declaration of variables containing links to the target class in the implementation of the impacted class may have to be changed.

Resulting Changes: Same as that of CAECq.

OCL Expressions: -- Same as that of CAECq.

35. Changed AssociationEnd - Changed visibility

Change Code: CAECv

Changed Element: model::foundation::core::AssociationEnd

Changed Property: visibility

Description: There is no impact since the model is assumed to be consistent.

36. Changed ClassClassifier - Added Attribute

Change Code: CCAA
Changed Element: model::foundation::core::ClassClassifier
Added Property: model::foundation::core::Attribute
Description: There is no impact since the model is assumed consistent. However, the class may have to be modified to include operations that access the new attribute and/or existing operations may have to be modified to access the new attribute. In addition, the class invariant may have to be modified to reflect the constraints (if any) on the attribute.

37. Changed ClassClassifier - Deleted Attribute

Change Code: CCDA
Changed Element: model::foundation::core::ClassClassifier
Deleted Property: model::foundation::core::Attribute
Description: There is no impact since the model is assumed to be consistent. However, the implementation of the changed class may have to be updated.

38. Changed ClassClassifier - Changed invariant

Change Code: CCCi
Changed Element: model::foundation::core::ClassClassifier
Changed Property: invariant
Description: There is no impact since the model is assumed to be consistent.

39. Changed ClassClassifier - Changed isAbstract

Change Code: CCCiAbs
Changed Element: model::foundation::core::ClassClassifier
Changed Property: isAbstract
Impacted Element: model::foundation::core::ClassClassifier
Description: The classes that have a navigable association to the changed class are impacted. The classes that have a dependency relationship (as a client) with the changed class are also impacted.
Rationale: If isAbstract is now true, then now it is not possible to have links to objects of the changed class, only links to concrete subclasses. If isAbstract is now false, then it is now possible to have links to objects of the changed class.
Resulting Change: The implementation of the impacted classes may have to be changed. For example, some of the methods that access the changed class may have to be changed.
OCL Expression:

```
context modelChanges::Change
self.changedElement.associationEnd->select(e:AssociationEnd|
e.isNavigable = true)->forAll(e:AssociationEnd|e.association.
getOtherEnd(e.getIDStr()))->union(self.changedElement.
clientDependency.client->select(c:Classifier|
c.ocIsTypeOf(ClassClassifier)))
```

40. Changed ClassClassifier - Changed isActive

Change Code: CCCiA
Changed Element: model::foundation::core::ClassClassifier
Changed Property: isActive
Description: There is no impact. However, the implementation of the changed class may have to be updated.

41. Changed ClassClassifier - Changed isLeaf

Change Code: CCCiL

Changed Element: model::foundation::core::ClassClassifier

Changed Property: isLeaf

Description: There is no impact since the model is assumed to be consistent. However, the implementation of the changed class may have to be updated. For example, the variable declaration section of the code may have to be changed to state that the class cannot be extended.

42. Changed ClassClassifier - Changed isRoot

Change Code: CCCiR

Changed Element: model::foundation::core::ClassClassifier

Changed Property: isRoot

Description: There is no impact since the model is assumed to be consistent. However, the implementation of the changed class may have to be updated.

43. Changed ClassClassifier - Changed multiplicity

Change Code: CCCm

Changed Element: model::foundation::core::ClassClassifier

Changed Property: multiplicity

Description: There is no impact since the model is assumed to be consistent. However, the implementation of the changed class may have to be updated.

44. Changed ClassClassifier - Added Operation

Change Code: CCAO

Changed Element: model::foundation::core::ClassClassifier

Added Property: model::foundation::core::Operation

Description: There is no impact since the model is assumed to be consistent. However, the implementation of the changed class may have to be updated.

45. Changed ClassClassifier - Deleted Operation

Change Code: CCDO

Changed Element: model::foundation::core::ClassClassifier

Deleted Property: model::foundation::core::Operation

Description: There is no impact since the model is assumed to be consistent. However, the implementation of the changed class may have to be updated.

46. Changed ClassClassifier - Changed visibility

Change Code: CCCv

Changed Element: model::foundation::core::ClassClassifier

Changed Property: visibility

Description: There is no impact since the model is assumed to be consistent. However, the implementation of the changed class may have to be updated. For example, the variable declaration section of the code may have to be changed to state the new visibility.

47. Changed Interface - Added Operation

Change Code: CIAO

Changed Element: model::foundation::core::Interface

Added Property: model::foundation::core::Operation

Description: There is no impact since the model is assumed to be consistent. However, the implementation of the changed interface may have to be updated.

48. Changed Interface - Deleted Operation

Change Code: CIDO
Changed Element: model::foundation::core::Interface
Deleted Property: model::foundation::core::Operation
Impacted Element: model::foundation::core::ClassClassifier
Description: The classes that realize the changed interface are impacted if they declare the deleted operation.
Rationale: The impacted classes realize an interface operation that has been deleted from the realized interface.
Resulting Change: The deleted interface operation may have to be deleted from the impacted classes. The method of the deleted operation may also have to be deleted from the impacted classes.
OCL Expression:

```
context modelChanges::Change
self.changedElement.implementationRealization.
implementation->select(c:ClassClassifier|
c.operation->includes(self.changedElement.
oclAsType(Interface).getOperation(self.propertyID).
oclAsType(Operation)))
```

49. Changed ClassifierRole - Added availableOperation

Change Code: CCRAaO
Changed Element: model::behaviouralElements::collaborations::ClassifierRole
Added Property: availableOperation
Description: There is no impact since the model is assumed to be consistent.

50. Changed ClassifierRole - Deleted availableOperation

Change Code: CCRDaO
Changed Element: model::behaviouralElements::collaborations::ClassifierRole
Deleted Property: availableOperation
Description: There is no impact since the model is assumed to be consistent.

51. Changed ClassifierRole - Changed base ClassClassifier

Change Code: CCRCbCC
Changed Element: model::behaviouralElements::collaborations::ClassifierRole
Changed Property: base
Description: Handled by the rules that deal with a changed class.

52. Changed ClassifierRole - Changed multiplicity

Change Code: CCRCm
Changed Element: model::behaviouralElements::collaborations::ClassifierRole
Changed Property: multiplicity
Description: Handled by the rule that deals with a changed class multiplicity (CCCM).

53. Changed Message action - Changed recurrence

Change Code: CMaCr

Changed Element: model::behaviouralElements::collaborations::Message

Changed Property: action.recurrence

Impacted Elements: model::foundation::core::ClassClassifier
model::foundation::core::Operation
model::foundation::core::Postcondition

Description: The base class of the classifier role that sends the changed message is impacted. The operation, that sends the changed message is impacted, and its postcondition is also impacted.

Rationale: One of the impacted class's actions has been changed. The impacted operation action has also been changed. The impacted postcondition may now not represent the effect (what must be true on completion) of its operation.

Resulting Changes: The implementation of the impacted class may have to be modified. The method of the impacted operation may have to be modified. The impacted postcondition should be checked to ensure that it is correct.

Invoked Rule: Changed Class Operation - Changed Postcondition (CCOCpst)

OCL Expressions:

```
context modelChanges::Change def:
    let changedMessage:Message = self.changedElement.
        oclAsType(Message)

    let sendingOperation:Operation =
        (if changedMessage.activator.action.oclIsTypeOf(CallAction)
         then
             changedMessage.sender.base.oclAsType(ClassClassifier)
             operation->select(o:Operation|
                 o.equals(changedMessage.activator.action.operation))
         else
             null
         endif)

context modelChanges::Change -- class
    changedMessage.sender.base.oclAsType(ClassClassifier)

context modelChanges::Change -- operation
    sendingOperation

context modelChanges::Change -- postcondition
    sendingOperation.postcondition
```

54. Changed CompositeState - Added subvertex

Change Code: CCSAs

Changed Element: model::behaviouralElements::stateMachines::CompositeState

Added Property: subvertex

Impacted Element: model::foundation::core::ClassClassifier

Description: The context class of the state machine to which the composite class belongs is impacted.

Rationale: The context class now has one more state.

Resulting Change: The implementation of the impacted class (or class cluster) may have to be modified to account for the extra state and the corresponding logic.

OCL Expression:

```
context modelChanges::Change
    self.changedElement.stateMachine.context
```


55. Changed CompositeState - Deleted subvertex

Change Code: CCSDs
Changed Element: model::behaviouralElements::stateMachines::CompositeState
Deleted Property: subvertex
Impacted Element: model::foundation::core::ClassClassifier
Description: The context class of the state machine to which the composite class belongs is impacted.
Rationale: The context class now has one less state.
Resulting Change: The implementation of the impacted class (or class cluster) may have to be modified to account for the deleted state and the corresponding logic.
OCL Expression: -- Same as that of CCSAs.

56. Changed Transition - Changed guard

Change Code: CTCg
Changed Element: model::behaviouralElements::stateMachines::Transition
Changed Property: guard
Impacted Element: model::foundation::core::ClassClassifier
Description: The context class of the state machine to which the transition belongs is impacted.
Rationale: The condition required to trigger the event (in the context class state machine), of the changed transition, has changed.
Resulting Change: The implementation of the impacted class (or class cluster) may have to be modified to account for the changed guard condition.
OCL Expression: -- Same as that of CCSAs.

57. Changed Attribute - Changed changeability

Change Code: CACc

Changed Element: model::foundation::core::Attribute

Changed Property: changeability

Impacted Elements: model::foundation::core::ClassClassifier
model::foundation::core::Operation

Description: The bag of impacted operations is such that the changed attribute's changeability is not "changeable" and the postcondition of each impacted operation possibly updates the changed attribute. The owner (class) of the attribute is also impacted.

Rationale: If the attribute's changeability was changed from "changeable" to "frozen" then each impacted operation's method may have to be changed to ensure that the attribute's value is not updated nor no additional values are added/deleted to/from the attribute. If the changeability was changed from "changeable" to "addOnly" then each impacted operation's method may have to be changed to ensure that the attribute's value is not updated nor no values deleted from the attribute. The implementation of the impacted class may have to be changed to ensure that the attribute's new changeability property is observed. For example, if the changeability was changed from "frozen" to "changeable" then some operations may have to be modified to update the attribute and/or new operations defined to update the attribute. The variable declaration for the attribute may have to be changed as well.

Resulting Changes: The implementation of the impacted operation may have to be changed. The implementation of the impacted class may have to be changed and/or operations defined/deleted/modified.

OCL Expressions:

```
context modelChanges::Change Def:
  let affectedClass:ClassClassifier = self.changedElement.
    getClassClassifier()

  context modelChanges::Change -- class
    affectedClass

  context modelChanges::Change -- operations
    if self.changedElement.oclaType(Attribute).getProperty(self.
      propertyID).oclaType(ChangeableKind) <> #changeable
      then
        affectedClass.getOperations()->select(o:Operation|
          o.postcondition.possiblyUpdatesVariable(self.
            changedElement.name))
      else
        null
    endif
```

58. Changed Attribute - Changed initialValue

Change Code: CACiV

Changed Element: model::foundation::core::Attribute

Changed Property: initialValue

Impacted Elements: model::foundation::core::Precondition
model::foundation::core::Postcondition

Description: The bag of impacted preconditions is such that each impacted precondition uses the changed attribute. The bag of impacted postconditions is such that each impacted postcondition uses the changed attribute.

Rationale: The changed initial value may now violate the impacted preconditions. The changed initial value may now change the impacted postconditions.

Invoked Rules: Changed Class Operation - Changed Precondition (CCOCpre)
Changed Class Operation - Changed Postcondition (CCOCpst)

OCL Expressions:

```
context modelChanges::Change -- precondition
self.changedElement.getClassClassifier().getOperations().
precondition->select(pr:Precondition|pr.usesVariable(self.
changedElement.name))

context modelChanges::Change -- postcondition
self.changedElement.getClassClassifier().getOperations().
postcondition->select(ps:Postcondition|ps.usesVariable(self.
changedElement.name))
```

59. Changed Attribute - Changed multiplicity

Change Code: CACm

Changed Element: model::foundation::core::Attribute

Changed Property: multiplicity

Impacted Elements: model::foundation::core::ClassClassifier
model::foundation::core::Operation

Description: The bag of impacted operations is such that the precondition of each impacted operation uses the changed attribute or the postcondition of each impacted operation accesses the changed attribute. The class that owns the changed attribute is also impacted.

Rationale: The methods of the impacted operations may not be accessing the correct attribute values. The variable declaration for the changed attribute may have to be changed. In addition, new operations may have to be defined, or operations deleted. Methods may also have to be changed to accomplish the change to the attribute.

Resulting Changes: The implementation of the impacted class may have to be changed. The implementation of the impacted methods may have to be changed.

OCL Expressions:

```
context modelChanges::Change Def:
let affectedClass:ClassClassifier = self.changedElement.
getClassClassifier()

context modelChanges::Change -- class
affectedClass

context modelChanges::Change -- operations
affectedClass.getOperations()->select(o:Operation|
o.precondition.usesVariable(self.changedElement.name) or
o.postcondition.accessesVariable(self.changedElement.name))
```

60. Changed Attribute - Changed ordering

Change Code: CACo

Changed Element: model::foundation::core::Attribute

Changed Property: ordering

Impacted Elements: model::foundation::core::ClassClassifier
model::foundation::core::Operation

Description: Same as that of CACm.

Rationale: The variable declaration (in the implementation of the impacted class) for the changed attribute may have to be changed. In addition, new operations may have to be defined, or operations deleted. Methods may also have to be changed to accomplish the changed ordering of the changed attribute.

Resulting Changes: Same as that of CACm.

OCL Expressions: -- Same as that of CACm.

61. Changed Attribute - Changed ownerScope

Change Code: CACoS

Changed Element: model::foundation::core::Attribute

Changed Property: ownerScope

Impacted Elements: model::foundation::core::ClassClassifier
model::foundation::core::Operation

Description: Same as that of CACm.

Rationale: The variable declaration (in the implementation of the impacted class) for the changed attribute may have to be changed. In addition, if the attribute is now static then an operation may have to be defined to update its value. In addition, the methods of the impacted operations should be checked to ensure that their access to the changed attribute is of the correct scope.

Resulting Changes: Same as that of CACm.

OCL Expressions: -- Same as that of CACm.

62. Changed Attribute - Changed targetScope

Change Code: CACtS

Changed Element: model::foundation::core::Attribute

Changed Property: targetScope

Impacted Elements: model::foundation::core::ClassClassifier
model::foundation::core::Operation

Description: Same as that of CACm.

Rationale: The variable declaration (in the implementation of the impacted class) for the changed attribute may have to be changed. In addition, if the attribute now stores static values then the methods of operations that access the attribute may have to be checked to ensure consistency.

Resulting Changes: Same as that of CACm.

OCL Expressions: -- Same as that of CACm.

63. Changed Attribute - Changed type

Change Code: CACt

Changed Element: model::foundation::core::Attribute

Changed Property: type

Impacted Elements: model::foundation::core::ClassClassifier
model::foundation::core::Operation

Description: Same as that of CACm.

Rationale: The variable declaration (in the implementation of the impacted class) for the changed attribute may have to be changed. The methods of the impacted operations may have to be changed if they contain type incompatibilities in regards to the changed attribute.

Resulting Changes: Same as that of CACm.

OCL Expressions: -- Same as that of CACm.

64. Changed Attribute - Changed visibility

Change Code: CACv

Changed Element: model::foundation::core::Attribute

Changed Property: visibility

Description: There is no impact since the model is assumed to be consistent.

65. Changed ClassClassifier Operation - Changed concurrency

Change Code: CCOCc

Changed Element: model::foundation::core::Operation

Changed Property: concurrency

Description: There is no impact. The method of the changed operation may have to be modified since concurrent access to it has changed.

66. Changed ClassClassifier Operation - Changed isAbstract

Change Code: CCOCiAbs

Changed Element: model::foundation::core::Operation

Changed Property: isAbstract

Description: There is no impact since the model is assumed to be consistent. However, the method may have to be modified to indicate whether the operation is abstract or not.

67. Changed ClassClassifier Operation - Changed isPolymorphic

Change Code: CCOCiP

Changed Element: model::foundation::core::Operation

Changed Property: isPolymorphic

Description: There is no impact since the model is assumed to be consistent. However, the method may have to be modified to indicate whether the operation can be overridden.

68. Changed ClassClassifier Operation - Changed isQuery

Change Code: CCOCiQ

Changed Element: model::foundation::core::Operation

Changed Property: isQuery

Description: There is no impact since the model is assumed to be consistent. However, the method (implementation) of the changed operation should be checked to ensure that it observes the isQuery property.

69. Changed ClassClassifier Operation - Changed ownerScope

Change Code: CCOCoS
Changed Element: model::foundation::core::Operation
Changed Property: ownerScope
Description: There is no impact since the model is assumed to be consistent. However, the method (implementation) of the changed operation should be checked to ensure that it observes the ownerScope property.

70. Changed ClassClassifier Operation - Changed precondition

Change Code: CCOCpre
Changed Element: model::foundation::core::Operation
Changed Property: precondition
Impacted Element: model::foundation::core::Operation
Description: The operations that invoke the changed operation are impacted..
Rationale: The methods of the impacted operations may now be violating the changed precondition.
Resulting Change: The methods of the impacted operations have to be checked to ensure that the changed precondition is not violated.
OCL Expression: `context modelChanges::Change
self.changedElement.getInvokingOperations()`

71. Changed ClassClassifier Operation - Changed postcondition

Change Code: CCOCpst
Changed Element: model::foundation::core::Operation
Changed Property: postcondition
Impacted Elements: model::foundation::core::Operation
model::foundation::core::Postcondition
Description: The operations that invoke the changed operation are impacted. The postconditions of the impacted operations are also impacted.
Rationale: The effect of each impacted operation may have to be changed and thus each impacted postcondition may have to be changed.
Resulting Change: The method (implementation) of each impacted operation may have to be changed. Each impacted postcondition should be checked to ensure that it now reflects the effect of its operation.
Invoked Rule: Changed Class Operation - Changed Postcondition (CCOCpst)
OCL Expressions: `context modelChanges::Change -- operations
changedOperation.getInvokingOperations()

context modelChanges::Change -- postconditions
changedOperation.getInvokingOperations().postcondition`

72. Changed ClassClassifier Operation - Changed visibility

Change Code: CCOCv
Changed Element: model::foundation::core::Operation
Changed Property: visibility
Description: There is no impact since the model is assumed to be consistent. However, the method may have to be modified to indicate the new visibility of the operation.

73. Changed Interface Operation - Changed concurrency

Change Code: CIOCc
Changed Element: model::foundation::core::Operation
Changed Property: concurrency
Description: There is no impact since the model is assumed to be consistent.

74. Changed Interface Operation - Changed isPolymorphic

Change Code: CIOCiP

Changed Element: model::foundation::core::Operation

Changed Property: isPolymorphic

Description: There is no impact since the model is assumed to be consistent. However, the code may have to be modified to state that the operation cannot be overridden.

75. Changed Interface Operation - Changed isQuery

Change Code: CIOCiQ

Changed Element: model::foundation::core::Operation

Changed Property: isQuery

Description: There is no impact since the model is assumed to be consistent.

76. Changed Interface Operation - Changed ownerScope

Change Code: CIOCoS

Changed Element: model::foundation::core::Operation

Changed Property: ownerScope

Description: There is no impact since the model is assumed to be consistent. However, the code may have to be modified to indicate the new scope.

77. Changed Interface Operation - Changed precondition

Change Code: CIOCpre

Changed Element: model::foundation::core::Operation

Changed Property: precondition

Description: There is no impact since the model is assumed to be consistent. The methods of the operations that realize the changed interface operation should be checked.

78. Changed Interface Operation - Changed postcondition

Change Code: CIOCpst

Changed Element: model::foundation::core::Operation

Changed Property: postcondition

Description: There is no impact since the model is assumed to be consistent. The methods of the operations that realize the changed interface operation should be checked.

79. Changed Class Operation Parameter - Changed defaultValue

Change Code: CCOPCdV

Changed Element: model::foundation::core::Parameter

Changed Property: defaultValue

Impacted Elements: model::foundation::core::Precondition
model::foundation::core::Postcondition

Description: The precondition of the operation to which the changed parameter belongs is impacted if it uses the changed parameter. The postcondition of the operation is also impacted if it uses the changed parameter.

Rationale: The impacted operation contracts (pre/postconditions) are using a parameter whose default value has changed, so conditional statements, for example, may yield different results.

Resulting Change: The method of the operation to which the changed parameter belongs should be checked and possibly changes made to account for the changed parameter default value.

Invoked Rules: Changed Class Operation – Changed Precondition (CCOCpre)
Changed Class Operation – Changed Postcondition (CCOCpst)

OCL Expressions: **context** modelChanges::Change -- precondition
self.changedElement.getOperation().
precondition->select(pr:Precondition|pr.
usesVariable(self.changedElement.name))

context modelChanges::Change -- postcondition
self.changedElement.getOperation().
postcondition->select(ps:Postcondition|ps.
accessesVariable(self.changedElement.name))

80. Changed Class Operation Parameter - Changed direction

Change Code: CCOPCd

Changed Element: model::foundation::core::Parameter

Changed Property: direction

Description: There is no impact since the model is assumed to be consistent. However, the method of the operation to which the changed parameter belongs should be checked to ensure consistency.

81. Changed Class Operation Parameter - Changed name

Change Code: CCOPCn
Changed Element: model::foundation::core::Parameter
Changed Property: name
Impacted Elements: model::foundation::core::Precondition
model::foundation::core::Postcondition
Description: Same as that of CCOPCdV.
Rationale: The impacted operation contracts (pre/postconditions) are using a parameter whose name has changed so they have to account for this change.
Resulting Change: The method of the operation to which the changed parameter belongs should be checked to ensure that it is using the correct name to reference the changed parameter.
Invoked Rules: Changed Class Operation – Changed Precondition (CCOCpre)
Changed Class Operation – Changed Postcondition (CCOCpst)
OCL Expressions:

```
context modelChanges::Change def:
    let parameterOriginalName:String = self.changedElement.
        getOperation().getClassClassifier().view.model.
        application.originalModel.classDiagramView.
        getClassClassifier(self.changedElement.getOperation().
        getClassClassifier().getPathname()).getOperation(self.
        changedElement.getOperation().getSignature()).
        parameter->asSequence->at(self.changedElement.
        getOperation().getParameterPosition(self.
        changedElement)).name

    context modelChanges::Change -- precondition
        self.changedElement.getOperation().
        precondition->select(pr:Precondition|pr.
        usesVariable(parameterOriginalName))

    context modelChanges::Change -- postcondition
        self.changedElement.getOperation().
        postcondition->select(ps:Postcondition|ps.
        usesVariable(parameterOriginalName))
```

82. Changed Interface Operation Parameter - Changed defaultValue

Change Code: CIOPCdV
Changed Element: model::foundation::core::Parameter
Changed Property: defaultValue
Impacted Elements: model::foundation::core::Precondition
model::foundation::core::Postcondition
Description: Same as that of CCOPCdV.
Rationale: Same as that of CCOPCdV.
Invoked Rules: Changed Class Operation – Changed Precondition (CCOCpre)
Changed Class Operation – Changed Postcondition (CCOCpst)
OCL Expressions: -- Same as that of CCOPCdV.

83. Changed Interface Operation Parameter - Changed direction

Change Code: CIOPCd
Changed Element: model::foundation::core::Parameter
Changed Property: direction
Description: There is no impact since the model is assumed to be consistent.

84. Changed Interface Operation Parameter - Changed name

Change Code: CIOPCn

Changed Element: model::foundation::core::Parameter

Changed Property: name

Impacted Elements: model::foundation::core::Precondition
model::foundation::core::Postcondition

Description: Same as that of CCOPCn.

Rationale: Same as that of CCOPCn.

Resulting Changes: The impacted pre/postcondition should be modified so that they reflect the new parameter name.

OCL Expressions:

```
context modelChanges::Change def:
  let parameterOriginalName:String = self.changedElement.
    getOperation().getInterface().view.model.
    application.originalModel.classDiagramView.
    getInterface(self.changedElement.getOperation().
    getInterface().getPathname()).getOperation(self.
    changedElement.getOperation().getSignature()).
    parameter->asSequence->at(self.changedElement.
    getOperation().getParameterPosition(self.
    changedElement)).name

context modelChanges::Change -- precondition
  self.changedElement.getOperation().
  precondition->select(pr:Precondition|pr.
  usesVariable(parameterOriginalName))

context modelChanges::Change -- postcondition
  self.changedElement.getOperation().
  postcondition->select(ps:Postcondition|ps.
  usesVariable(parameterOriginalName))
```

85. Changed State - Added doActivity

Change Code: CSAdA

Changed Element: model::behaviouralElements::stateMachines::State

Added Property: doActivity

Impacted Element: model::foundation::core::ClassClassifier

Description: The class (context class) that owns the state machine of the changed state is impacted.

Rationale: The behaviour of the context class has changed.

Resulting Change: The implementation of the impacted class may have to be modified.

OCL Expression:

```
context modelChanges::Change
  self.changedElement.stateMachine.context
```

86. Changed State - Deleted doActivity

Change Code: CSDdA

Changed Element: model::behaviouralElements::stateMachines::State

Deleted Property: doActivity

Impacted Element: model::foundation::core::ClassClassifier

Description: Same as that of CSAdA.

Rationale: Same as that of CSAdA.

Resulting Change: Same as that of CSAdA.

OCL Expression: -- Same as that of CSAdA.

87. Changed State - Added entry

Change Code: CSAe
Changed Element: model::behaviouralElements::stateMachines::State
Added Property: entry
Impacted Element: model::foundation::core::ClassClassifier
Description: Same as that of CSAdA.
Rationale: Same as that of CSAdA.
Resulting Change: Same as that of CSAdA.
OCL Expression: -- Same as that of CSAdA.

88. Changed State - Deleted entry

Change Code: CSDe
Changed Element: model::behaviouralElements::stateMachines::State
Deleted Property: entry
Impacted Element: model::foundation::core::ClassClassifier
Description: Same as that of CSAdA.
Rationale: Same as that of CSAdA.
Resulting Change: Same as that of CSAdA.
OCL Expression: -- Same as that of CSAdA.

89. Changed State - Added exit

Change Code: CSAex
Changed Element: model::behaviouralElements::stateMachines::State
Added Property: exit
Impacted Element: model::foundation::core::ClassClassifier
Description: Same as that of CSAdA.
Rationale: Same as that of CSAdA.
Resulting Change: Same as that of CSAdA.
OCL Expression: -- Same as that of CSAdA.

90. Changed State - Deleted exit

Change Code: CSDex
Changed Element: model::behaviouralElements::stateMachines::State
Deleted Property: exit
Impacted Element: model::foundation::core::ClassClassifier
Description: Same as that of CSAdA.
Rationale: Same as that of CSAdA.
Resulting Change: Same as that of CSAdA.
OCL Expression: -- Same as that of CSAdA.

91. Changed State - Added internalTransition

Change Code: CSAiT
Changed Element: model::behaviouralElements::stateMachines::State
Added Property: internalTransition
Impacted Element: model::foundation::core::ClassClassifier
Description: Same as that of CSAdA.
Rationale: Same as that of CSAdA.
Resulting Change: Same as that of CSAdA.
OCL Expression: -- Same as that of CSAdA.

92. Changed State - Deleted internalTransition

Change Code: CSDiT
Changed Element: model::behaviouralElements::stateMachines::State
Deleted Property: internalTransition
Impacted Element: model::foundation::core::ClassClassifier
Description: Same as that of CSAdA.
Rationale: Same as that of CSAdA.
Resulting Change: Same as that of CSAdA.
OCL Expression: -- Same as that of CSAdA.

93. Changed State - Changed invariant

Change Code: CSCi
Changed Element: model::behaviouralElements::stateMachines::State
Changed Property: invariant
Impacted Element: model::foundation::core::ClassClassifier
Description: Same as that of CSAdA.
Rationale: A state invariant has been changed so the implementation of the class has to be checked.
Resulting Change: The implementation of the context class may have to be modified.
OCL Expression: -- Same as that of CSAdA.

94. Changed StateMachine action - Added (Discrete) action

Change Code: CSMaADa
Changed Element: model::commonBehaviour::Action
Added Property: action
Impacted Element: model::foundation::core::ClassClassifier
Description: The class (context class) that owns the state machine to which the changed action belongs is impacted.
Rationale: A discrete action has been added to the action performed by a transition in the state machine so the implementation of the class has to be checked.
Resulting Change: The implementation of the context class may have to be modified to account for the added discrete action.
OCL Expression:

```
context modelChanges::Change
  if self.changedElement.state->notEmpty then
    self.changedElement.state.stateMachine.context
  else -- the else clause assumes that the state machine action
    -- belongs to a transition instead of a state
    self.changedElement.transition.stateMachine.context
```

95. Changed StateMachine action - Deleted (Discrete) action

Change Code: CSMaDDa
Changed Element: model::commonBehaviour::Action
Deleted Property: action
Impacted Element: model::foundation::core::ClassClassifier
Description: Same as that of CSMaADA.
Rationale: A discrete action has been deleted from the action performed by a transition in the state machine so the implementation of the class has to be checked.
Resulting Change: The implementation of the context class may have to be modified to account for the deleted discrete action.
OCL Expression: -- Same as that of CSMaADA.

96. Changed StateMachine action - Changed recurrence

Change Code: CSMaCr
Changed Element: model::commonBehaviour::Action
Changed Property: recurrence
Impacted Element: model::foundation::core::ClassClassifier
Description: Same as that of CSMAADa.
Rationale: The recurrence property of an action in the state machine has been changed thus, this has to be reflected in the implementation of the context class.
Resulting Change: The implementation of the context class may have to be modified to account for the changed recurrence property.
OCL Expression: -- Same as that of CSMAADa.

97. Changed StateMachine action - Changed script

Change Code: CSMAcs
Changed Element: model::commonBehaviour::Action
Changed Property: script
Impacted Element: model::foundation::core::ClassClassifier
Description: Same as that of CSMAADa.
Rationale: The script property of an action in the state machine has been changed thus, this has to be reflected in the implementation of the context class.
Resulting Change: The implementation of the context class may have to be modified to account for the changed script property.
OCL Expression: -- Same as that of CSMAADa.

Appendix C – Consistency Rules

Each consistency rule is briefly described in Natural Language. Each of these rules identifies the set of model elements that cause the inconsistency. There are also some consistency warnings listed below, these indicate consistency conditions that should be checked. These warnings arise because further work (semantic analysis of OCL expressions) is needed to extract the required information from the model so that they can be consistency rules.

1. No protected operation can be invoked (in a sequence diagram) by an operation belonging to a class that is not a descendent of the class that owns the invoked operation.
2. No protected operation can be invoked (in a statechart) by an operation belonging to a class that is not a descendent of the class that owns the invoked operation.
3. No private operation can be invoked (in a sequence diagram) by an operation belonging to another class.
4. No private operation can be invoked (in a statechart) by an operation belonging to another class.
5. Each object (in a sequence diagram) must be an instantiation of a class in a class diagram.
6. If an operation appears in a pre or postcondition then it must have the property “query”.
7. An attribute with the “frozen” property cannot be assigned a value in a sequence diagram.
8. An attribute with the “frozen” property cannot be assigned a value in a state transition.
9. A class that has the “leaf” property cannot be extended.
10. A class that has the “root” property cannot extend another class.
11. The comparison types of the attributes in the class invariant must be compatible.
12. The comparison types of attributes in a precondition must be compatible.
13. The comparison types of attributes in a postcondition must be compatible.

14. If an attribute's type is a class then that class has to be visible to the class containing the attribute.
15. If the return type of an operation is a class then that class has to be visible to the class containing the operation.
16. In a sequence diagram, if an attribute is assigned the return value of an operation, then the types have to be compatible.
17. For each message between two objects (in a sequence diagram) there has to be a valid path (navigable) between them.
18. Each attribute in a precondition must appear in the class diagram.
19. Each attribute in a postcondition must appear in the class diagram.
20. Each precondition should not violate the class invariant.
21. Each postcondition should not violate the class invariant.
22. An abstract operation cannot be invoked in a sequence diagram.
23. An abstract operation cannot be invoked in a statechart.
24. A class that contains an abstract operation must be abstract.
25. A concrete descendant of an abstract class must implement each abstract operation of its parent.
26. An operation that is not polymorphic may not be overridden by a descendant class.
27. An operation that has the property "query" cannot be an event in a statechart.
28. A static operation cannot access an instance attribute.
29. A static operation cannot invoke an instance operation.
30. No private attribute can be accessed by an operation of another class.
31. No protected attribute can be accessed by an operation of a class that is not a descendant of the class owning the attribute.
32. Each attribute that is called in a statechart transition must be defined in the corresponding class diagram.
33. Each operation that is invoked in a sequence message must be defined in a class diagram.
34. Each operation that is invoked in a state transition must be defined in a class diagram.
35. There must be no cycles in the directed paths of aggregation links. A class cannot be a part in an aggregation in which it is the whole. A class cannot be a part of an aggregation in which its super-class is the whole.
36. There must not be two (or more) associations present in the static view of the model such that they cannot be distinguished. That is, no two associations in the static view must connect the same two classes having the same name, and the same role names.

37. A class cannot be a part in more than one composition – no composite part may be shared by two composite objects.
38. Each base class that appears in a sequence diagram must be defined in the static view of the model.
39. No attribute of a class and a qualifier or rolename of an associated class can have the same name.
40. If an association end has a private visibility, then its participant can only be accessed, via the association, by the class at the other association end.
41. If an association end has a protected visibility, then its participant can only be accessed, via the association, by the class at the other association end and classes that are descendants of the participant.
42. If a navigation expression occurs in an operation contract, then there must exist a navigable association from the class that owns the contract's operation to the target class in the navigation expression.
43. For each operation that is invoked in a state transition, there must exist a navigable association from the context class to the class that owns the invoked operation.
44. A sequence message cannot update an attribute if the attribute's changeability is not "changeable".
45. A transition's action list cannot update an attribute if the attribute's changeability is not "changeable".
46. The postcondition of an operation must not *possibly update* an attribute whose changeability is not "changeable".
47. The multiplicity range for an attribute must be adhered to by all elements that access it.
48. For every class in which operations use another class there must be a dependency relationship between the two classes in the class diagram.
49. An abstract class cannot be instantiated.
50. A class's multiplicity must not be violated by the multiplicity of any association end in which it is the participant.
51. A class's multiplicity must not be violated by the multiplicity of any classifier role in which it is the base.
52. A class's package visibility should be observed, especially for associations between classes of different packages.
53. A class that realizes an interface must declare all the operations in the interface.
54. A classifier role cannot have an available operation that is not declared in its base class.
55. An operation cannot be invoked on a classifier role that is not present in its set of available operations.
56. An abstract operation cannot be invoked.
57. The directions of all the parameters of any class operation, that realizes an interface operation, must match the directions of the parameters of the interface operation.

58. The default values of all the parameters of any class operation, that realizes an interface operation, must match the default values of the parameters of the interface operation.
59. For all the class operations that realize an interface operation, their concurrency values must be the same as that of the interface operation.
60. For all the class operations that realize an interface operation, their polymorphic properties must be the same as that of the interface operation.
61. For all the class operations that realize an interface operation, their query properties must be the same as that of the interface operation.
62. For all the class operations that realize an interface operation, their owner scope values must be the same as that of the interface operation.
63. For all the class operations that realize an interface operation, their precondition must be the same as that of the interface operation.
64. For all the class operations that realize an interface operation, their postcondition must be the same as that of the interface operation.
65. There must not exists two classifier roles in the sequence diagram view such that both roles have the same pathnames.

Consistency Warnings

1. For all preconditions of operations in a class, they must not (possibly) violate the class invariant.
2. For all postconditions of operations in a class, they must not (possibly) violate the class invariant.
3. For all the state invariants in a state machine of a class, they must not (possibly) violate the class invariant.
4. For each state invariant in a state machine of a class, they must not (possibly) violate the preconditions of the operations, that are invoked in that state.
5. For each state invariant in a state machine of a class, they must not (possibly) violate the postconditions of the operations that have been invoked in a transition whose target state is the state of the state invariant.

Appendix D – Case Study

D.1 Logical Changes

Eleven (11) logical changes were made in the case study. These translated into 70 model changes. The logical changes are described below.

Change # 1

It is required to keep track of how many times per session a user attempts to enter the PIN – after 3 invalid PIN entries the card is retained. This translates into the following changes:

1. (CCAA)1 new attribute in class ATM - numberOfTries : Integer = 0
2. 4 new methods:
 - a. (CCAO)resetNumTries() : Void (Class ATM)
 - b. (CCAO)incrementNumTries() : Void (Class ATM)
 - c. (CCAO)getNumTries() : Void (Class ATM)
 - d. (CCAO)displayRetainCard() : Void (Class Display)
3. 4 new messages in sequence diagrams
 - a. (CSDVAM)1.2: resetNumTries() (CardInsert)
 - b. (CSDVAM)4: try = getNumTries() (PINInvalid)
 - c. (CSDVAM)5: [try <=3] displayRetainCard() (PINInvalid)
 - d. (CSDVAM)1.3: incrementNumTries() (GetPIN)
4. 1 changed message in PINInvalid sequence diagram
 - a. (CMaCr)old – 3: [err=1]pin = getPIN()
 - b. new – 4: [err = 1 and try < 3] pin = getPIN
5. 1 added object to PINInvalid sequence diagram
 - a. (CSDVACR)display:Display

Change #2

The ATM's attribute 'state' would be better represented by an enumeration class than a simple integer. This translates into the following changes:

1. (CCDVAI)New Interface 'java.util Enumeration' – containing two methods
 - a. (CCAO)hasMoreElements():boolean
 - b. (CCAO)nextElement():Object
2. (CCDVAC)New Class 'StateEnum' containing 10 attributes and 10 methods
 - a. (CCAA)Private static final int Off
 - b. (CCAA)Private static final int WitingForCard
 - c. (CCAA)Private static final int GettingPIN
 - d. (CCAA)Private static final int GettingTransType
 - e. (CCAA)Private static final int AskingDoAnother
 - f. (CCAA)Private static final int GettingAccountType

- g. (CCAA)Private static final int GettingTransAmount
 - h. (CCAA)Private static final int PerformingTrans
 - i. (CCAA)Private static final int PrintingReceipt
 - j. (CCAA)Private int CurrentState
 - k. (CCAO)Public int getCurrentState()
 - l. (CCAO)Public void setToOffState()
 - m. (CCAO)Public void setToWaitingForCard()
 - n. (CCAO)Public void setToGettingPINState()
 - o. (CCAO)Public void setToAskingDoAnotherState()
 - p. (CCAO)Public void setToGettingAccountTypeState()
 - q. (CCAO)Public void setToGettingTransAmountState()
 - r. (CCAO)Public void setToPerformingTransState()
 - s. (CCAO)Public void setToPrintingReceiptState()
 - t. (CCAO)Public void setToGettingTransTypeState()
3. (CCDVAR)New Realization - StateEnum realizes Enumeration
 4. (CCDA)Deleted Attribute in Class ATM : state:int
 5. (CCDVAA) Added association between ClassATM and Class StateEnum
 6. (CSDVAM) ATMShutOff – added message ‘1.1.2: myState.setToOffState()’
 7. (CSDVACR) ATMShutOff – added myState:StateEnum
 8. (CSDVACR) ATMStartup – added myState:StateEnum
 9. (CSDVAM) ATMStartup – added message ‘ 1.1.4 : setToWaitingForCardState()’
 10. (CSDVACR)CardInsert – added myState:StateEnum
 11. (CSDVAM)CardInsert – added message ‘3: setToGettingPINState()’
 12. (CSDVACR)GetPIN – added myState:StateEnum
 13. (CSDVAM)GetPIN – added message ‘1.4: setToGettingTransTypeState()’
 14. (CSDVACR)Transaction – added myState: StateEnum
 15. (CSDVAM)Transaction – added message ‘1.3: setToGettingAccountTypeState()’
 16. (CSDVAM)Transaction – added message ‘ 2.3: setToGettingAmountState()’
 17. (CSDVAM)Transaction – added message ‘ 2.4.3: setToPerformingTransState()’
 18. (CSDVAM)Transaction – added message ‘ 2.5.7: set To AskingDoAnotherState()’
 19. (CSDVACR)AskingDoAnother – added myState:StateEnum
 20. (CSDVAM)AskingDoAnother – added message ‘1.3: [response = 1]setToGettingTransTypeState()’
 21. (CSDVAM)AskingDoAnother – added message ‘ 1.4: [response = 2] setToPrintingReceiptState()’
 22. (CSDVACR)PrintReceipt – added myState: StateEnum
 23. (CSDVAM)PrintReceipt – added message ‘2.2: setToWaitingForCardState()’
 24. (CSDVACR)Cancel – added myState: StateEnum
 25. (CSDVAM)Cancel – added message ‘1.1.2: state = getCurrentState()’
 26. (CSDVAM)Cancel – added message ‘1.1.3: [state = 3]setToWaitingForCardState()’
 27. (CSDVAM)Cancel – added message ‘1.1.4: [state = 4 or 5 or 6]setToAskingDoAnotherState()’
 28. (CSDVACR)CardNotReadable – added myState :StateEnum
 29. (CSDVAM)CardNotReadable – added message ‘3: setStateWaitingForCard()’

- 30. (CSDVACR)FailedTransaction – added myState:StateEnum
- 31. (CSDVAM)FailedTransaction – added message ‘ 4: [err=2 or err=3]setToGettingTransAmountState()’
- 32. (CSDVACR)PINInvalid – added myState:StateEnum
- 33. (CSDVAM)PINInvalid – added message ‘ 4: [err = 1 and try < 3] setToGettingPINState()’
- 34. (CSDVAM)PINInvalid – added message ‘ 5:[try >=3] setToWaitingForCardState()’

Change # 3

An account can be owned by at most 2 customers and at least 1 customer

- 1. (CAECm) Account – Customer from 1..* to 1,2

Change # 4

A customer must belong to a bank.

- 1. (CAECm)Bank- Customer from 0..* to 1

Change # 5

Class Account is changed to an Abstract class – Rationale: there will never be an instance of class Account since accounts are always either Savings or Chequing accounts

- 1. (CCCiAbs) Account

Change #6

A confirmation message is displayed to the customer acknowledging the receipt of his/her deposit:

- 1.1.3.1:display(“Your message has been accepted.”) // in the Deposit interaction

Change #7

Provides feedback to the ATM operator upon the loading (cash) of the ATM:

- 1.1.13.1.1.1:displayAmounts() // in the ATMStartUp interaction

Error Correctrions:

Change #8

Changed invariant of the Savings class

Change #9

Changed initial value of the transAmount attribute in the ATM class

Change #10

Corrected a syntax error in the postcondition of the setPIN operation in the Transaction class

Change #11

Changed the message condition for message 1.1.2 in the Inquiry interaction

D.2 ATM System UML Model (Original Version)

Figure 58: ATM System – Class Diagram.

Figure 59: *ATM System – AskingDoAnother Sequence Diagram.*

Figure 60: *ATM System – ATMShutOff Sequence Diagram.*

Figure 61: *ATM System – ATMStartUp Sequence Diagram.*

Figure 62: *ATM System – Cancel Sequence Diagram.*

Figure 63: *ATM System – CardInsert Sequence Diagram.*

Figure 64: *ATM System – CardNotReadable Sequence Diagram.*

Figure 65: *ATM System – FailedTransaction Sequence Diagram.*

Figure 66: *ATM System – Deposit Sequence Diagram.*

Figure 67: *ATM System – GetPIN Sequence Diagram.*

Figure 68: *ATM System – Inquiry Sequence Diagram.*

Figure 69: *ATM System – PINInvalid Sequence Diagram.*

Figure 70: *ATM System – PrintReceipt Sequence Diagram.*

Figure 71: ATM System – Transaction Sequence Diagram.

Figure 72: *ATM System – Transfer Sequence Diagram.*

Figure 73: *ATM System – Withdraw Sequence Diagram.*