

Impact Analysis and Change Management of UML Models

L. C. Briand, Y. Labiche, L. O’Sullivan
Software Quality Engineering Laboratory
Systems and Computer Engineering Department
Carleton University, Ottawa, Ontario, Canada

ABSTRACT

The use of Unified Model Language (UML) analysis/design models on large projects leads to a large number of interdependent UML diagrams. As software systems evolve, those diagrams undergo changes to, for instance, correct errors or address changes in the requirements. Those 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. In this article, we propose a UML model-based approach to impact analysis that can be applied before any implementation of the changes, thus allowing an early decision-making and change planning process. We first verify that the UML diagrams are consistent (consistency check). Then changes between two different versions of a UML model are identified according to a change taxonomy, and model elements that are directly or indirectly impacted by those changes (i.e., may undergo changes) are determined using formally defined impact analysis rules (written with Object Constraint Language). A measure of distance between a changed element and potentially impacted elements is also proposed to prioritize the results of impact analysis according to their likelihood of occurrence. We also present a prototype tool that provides automated support for our impact analysis strategy, that we then apply on a case study to validate both the implementation and methodology.

TABLE OF CONTENTS

Abstract	1
TABLE OF CONTENTS.....	2
1. Introduction.....	3
2. Related Works.....	5
3. Problem definition and objectives	6
4. Overview of the Approach.....	7
5. Tool architecture and Overview.....	9
6. Model changes	11
7. Impact Analysis rules.....	15
8. Distance Measure.....	18
9. Case study	19
10. Conclusions.....	21
Acknowledgements.....	22
References	23
Appendix A System Model	24
Appendix B Change Detection.....	41
B.1 Change Taxonomy	41
B.2 Change Detection Rules.....	51
Appendix C Consistency Verification.....	69
Appendix D Impact Analysis (Side Effect) Rules.....	75
Appendix E Case Study	110
E.1 Logical Changes.....	110
E.2 Change Distribution	113
E.3 Impacts Vs Distance Graphs	113
E.4 UML Model (Original)	115

1. INTRODUCTION

The use of UML (Unified Model Language) analysis/design models [7] on large projects leads to a large number of inter-dependent UML diagrams¹. Those diagrams undergo changes as the 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, several issues require attention. 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 predict the cost and complexity of changes and help decide whether to implement them in a new release [2].

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 those 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 [2].

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 that the complexity of changing Analysis and Design models is also very high. Therefore, we seek 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 the impact analysis can be performed. However, a UML model-based approach to impact analysis looks at impacts to the system before the

¹ That may also contain OCL [13] constraints, e.g., contracts, guard conditions.

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 show up at implementation time. This is an issue that requires further investigation but that will not be addressed in this report.

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 many UML CASE tools. For example, Together®, by TogetherSoft™ [11], 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.

Our work contributes in several complementary ways to providing support for the impact analysis of UML models:

- It defines a methodological framework.
- It provides a set of change detection and impact analysis (side effect) rules, that were derived by systematically analyzing components of UML models (including constraints in the Object Constraint Language [13]) and analyzing changes in actual case studies.
- A prototype tool implements the above principles using a carefully thought-out architecture and an extensible design.
- Case studies have been performed to assess the feasibility and practical challenges of our approach.

This report describes the methodological framework and the fundamental principles underlying the change detection and impact analysis rules, presents our tool's architecture at a high level, and reports on a case study. Section 2 discusses related works. Section 3 provides a precise description of the problems we addressed and the objectives of our research. An overview of the approach, along with some justifications, is given in Section 4. The next Sections, up to Section 9, which presents a case study, then details each of the most important aspects of the approach and provides examples. Section 10 outlines our main conclusions and future work.

2. RELATED WORKS

Bohner [1] examines the general issues involved in change impact analysis, and provides structured guidelines to help find solutions to such issues. For instance, if one considers both direct and indirect (transitive closure) impacts, the results of the impact analysis shows an enormous 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 (e.g., distance between a change and an impact) constraints to structure analysis results.

A large portion of the change impact analysis strategies require source code analysis (see for instance the strategies reported in [3]), where as a few of them are model-based. Kung et al. [9] describes how change impact analysis can be performed from a class diagram, introducing the notion of class firewall (i.e., classes that may be impacted by a change in a given class), and discuss the impact of object-oriented characteristics (e.g., encapsulation, inheritance, polymorphism, dynamic binding) on such an analysis. In [12], 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 [5]. In this method, a rough impact analysis is performed with the sole purpose of classifying the regression test cases as obsolete, retestable, 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 estimates and support to identify ripple effects.

3. PROBLEM DEFINITION AND OBJECTIVES

The support of impact analysis of UML design models can be decomposed into several sub-problems:

1. *Automatically detect and classify changes* across different versions of UML models. Ideally, one modifies a UML model and then uses the impact analysis tool to automatically identify all the changes performed since the last version. We do not want software engineers to have to specify each and every change as we want to avoid the overhead that would prevent the practice of impact analysis. As seen below, changes have to be classified to be able to perform a precise impact analysis.
2. *Verify the consistency of changed diagrams*. The modified model 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, verifying consistency must be supported by tools. Note that this is different from impact analysis as it does not focus on finding (potentially) impacted elements (i.e., whose implementation may require change) but structural inconsistencies between UML diagrams, e.g., a class instance (classifier role²) in a sequence diagram whose class is not in the class diagram.
3. *Perform an impact analysis* to determine the *potential* side effects of changes in the design. 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 only. As a result, an *impacted element* is a UML model element whose properties or implementation *may* require modification as a result of changing another model element (i.e., one of its

² In the UML standard terminology, 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).

properties may change)³. To clarify the terminology we employ, changes to UML diagrams are the result of *logical changes* corresponding to error corrections, design improvements, or requirement changes. We refer to *changes to model elements* when a property of an element has changed from one version of a diagram to another, e.g., the visibility of an operation. A logical change usually results in a set of changes to model elements. Impact analysis can be performed for each logical change independently or for an entire, new UML model.

4. *Prioritize the results of impact analysis* 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. [6] have explored the use of coupling measures and predictive statistical model for that purpose.

4. OVERVIEW OF THE APPROACH

In this section, we do not present all the details of our change impact analysis strategy. Further details are presented in the next sections. Rather we concentrate on the important notions, providing excerpts for all the four steps that are involved in the strategy: consistency checking, change impact analysis, prioritization of impacts.

As mentioned above, the identification of model inconsistencies is important to ensure that the impact analysis algorithms we use yield correct results. Inconsistencies may be automatically modeled and detected by a set of *consistency rules*. Each rule corresponds to one type of inconsistency and must be implemented in any tool supporting impact

³ Even when no model property changes, the model element implementation may require change.

analysis on UML diagrams. We have identified 120 consistency rules⁴. For example, one simple rule we use can be described informally⁵ as:

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 model element in a UML design is defined by a set of *properties*, e.g., a class has attributes. Thus, the identification of a change to a model element requires checking if any of its properties has changed. Each model element change is classified according to a *change taxonomy* in order to associate impact analysis rules with each type of change. The change taxonomy reflects changes to class diagrams, sequence diagrams, and statecharts. More details are provided, for some examples, in Section 6, and the complete change taxonomy contains 97 change categories⁴ (leaf nodes).

Once we have verified that the diagrams of a UML design model are consistent, and model element changes have been detected, the next step is to automatically perform impact analysis using *impact analysis rules*, that is, rules that determine what model elements could be *directly* or *indirectly* (through transitive closure) impacted by each model element change (Section 7). As rules tend to depend on the type of change for which we perform impact analysis, we define one such rule for each change category in the change taxonomy, thus resulting in 97 rules⁴.

In order for impact analysis to be useful and practical, we need to find ways to indicate what model elements should be checked first as they, and their code counterpart, are more likely to require change. To do so, we define measures of distance between the changed elements and potentially impacted elements (Section 8) where the assumption is that the larger the distance, the less likely is the model element to be impacted.

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

⁴ Though we made a conscious effort to be as exhaustive as possible, this number may change as we gain more experience, especially by applying our change impact analysis strategy to different case studies.

⁵ It can also be expressed using OCL on the meta-model

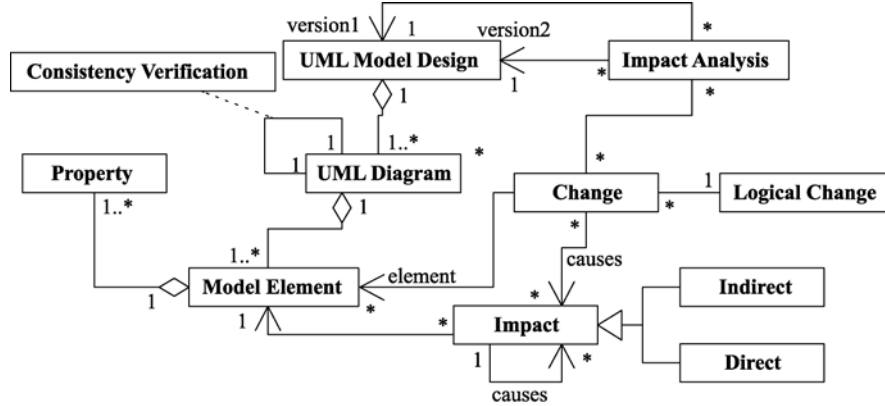


Figure 1 – Conceptual Model

5. TOOL ARCHITECTURE AND OVERVIEW

Our impact analysis tool (iACMTool) reads two versions of a UML model (composed of a number of diagrams and associated OCL constraints) and produces an impact analysis report as well as a consistency verification report. After each version of the model is read, its consistency is first verified. When both versions have been read and checked for internal consistency, change detection is done to identify all the changes between the two versions of the model, and classify them according to the taxonomy we defined (Section 6). These changes are then used to perform impact analysis on the model using the impact analysis rules relevant to each change type.

There are seven main packages in the system, namely: `parser`, `model`, `modelChanges`, `consistencyVerification`, `impactAnalysis`, `reportGeneration`, and `control`. The subsystem decomposition is shown in Figure 2 with packages and dependencies among them. More architectural details can be found in Appendix A. In particular, the packages contain 99 classes, 69 of which are in the `model` package (the UML meta-model), and the current implementation consists of 9064 lines of Java source code, excluding comments.

The `parser` subsystem has two main functions: (1) parsing XMI (XML Metadata Interchange [8]) files that describe the UML models, (2) parsing OCL expressions associated with the models. Parsed model information is then stored in the `model` subsystem, which also handles persistency. The `model` subsystem is a UML meta-model adapted to our requirements (e.g., it has been modified to improve information retrieval

efficiency). This meta-model is based on the official UML meta-model [10] and supports features related to three views of the meta-model: static (class diagram) view, interaction (sequence diagram) view, and the statechart diagram view. This includes classes, interfaces, sequence messages, state machines, but also class invariants, state invariants as well as pre- and post-conditions. It is designed so that it can later be upgraded to include other features of UML such as use case and activity diagrams. The modified UML meta-model is presented in Appendix A and an excerpt is presented in Section 7. The `modelChanges` subsystem is responsible for change detection by analyzing the two versions of a UML design model. The main class in this package, `ChangeDetector`, implements the change detection rules corresponding to the change taxonomy introduced previously and further detailed in Section 6. The `consistencyVerification` subsystem is responsible for checking consistency in each version of the model, using the set of rules discussed above. The `control` subsystem is responsible for the overall control flow of the application. The `impactAnalysis` subsystem is responsible for performing the impact analysis related to a set of model element changes. This subsystem implements the impact analysis rules discussed above and further detailed in Section 7. The `reportGeneration` subsystem is responsible for generating the different types of reports required by the system, including a consistency verification report, and an impact analysis report. Different flavors of the reports may be generated to meet the requirements of the user.

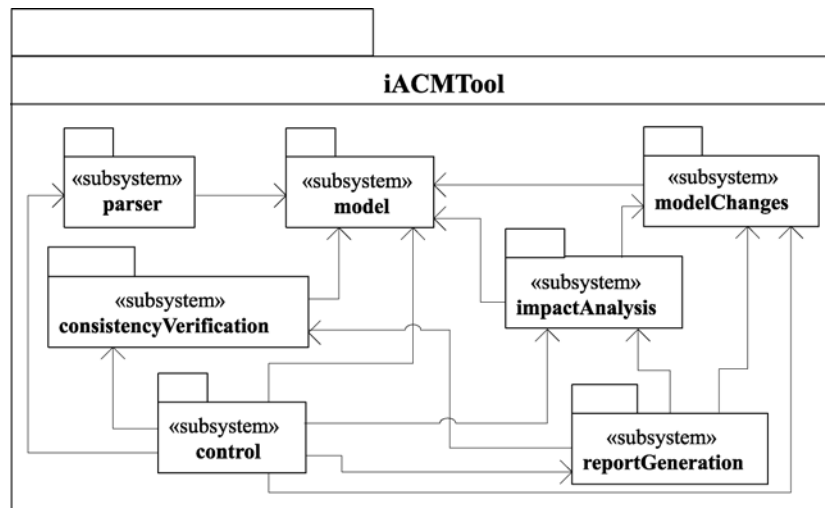


Figure 2 – Impact Analysis Tool Subsystems

6. MODEL CHANGES

To derive the change taxonomy, we analyzed each property of each model element (in the UML meta-model) to determine the possible changes that can occur. An element property is modeled as an attribute or an aggregation link to another element. In the latter case, linked elements are termed *impact related elements* since a change to one of these component elements affects the composite element to which it belongs. For example, if an attribute is changed then the class to which it belongs is considered impacted. A *changed* element property is defined as a changed attribute of the element, or an added or deleted link to an impact related element in the meta-model. For example, using an excerpt of the meta-model in Figure 3, we see that an association end has several properties, some modeled as a link to model elements (qualifier modeled as a link to zero or several attributes) and others as attributes (e.g., `isNavigable` to model whether an association end is navigable).

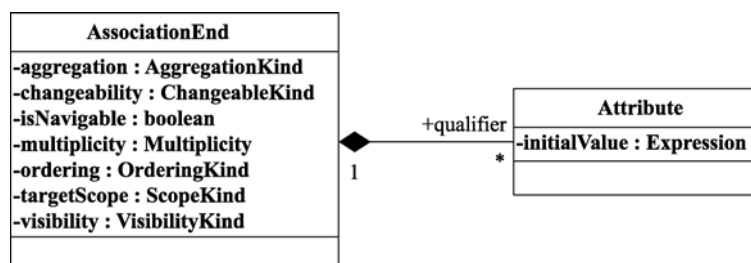


Figure 3 – Example of impact related element from the meta-model

Some element properties uniquely identify the element among the set of all elements instantiating a meta-model class. These properties are not included in the change taxonomy but the element is considered deleted and a new element added if a change to such a property occurs. For example, a class is uniquely identified by its name within its package's namespace, and thus a changed class name is regarded as the deletion of the original class and the addition of a new class. Using such key attributes is the way any impact analysis system can keep track of the identity of model elements across design versions.

We provide below a set of definitions regarding the basic terminology and concepts used throughout the report.

Definition 1: Model element changes

Let $e \in E$, where E is the set of all model elements (i.e., meta-model instances) in the UML design model. Let P be the set of all the properties of e . Let $P_U \subset P$ be the set of properties that uniquely identify e . If any one $p \in (P - P_U)$ is changed, then e is changed.

Definition 2: Impact related elements

Given 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 said to be an impact related element of e_1 if when e_2 is changed then e_1 is considered changed.

Using definitions above, a change taxonomy is provided in Appendix B. The UML class diagram notation is used to describe the taxonomy, as illustrated in Figure 5. Each non-terminal node in the taxonomy represents an abstract change category of a model element. The leaf nodes correspond to one changed element property.

For example, let us look at a simple change example: Adding a message in a sequence diagram. We provide in Figure 4 a description of the change. Each change category has an acronym, a short textual description, and an OCL expression that shows how, based on the `model` subsystem class diagram (which is instantiated by the `parser` subsystem), such changes can be automatically detected. In our example, the OCL expression returns a collection of added messages in a given Sequence Diagram View (we always assume the context of the OCL expression is the *modified* view). Such OCL expressions are logical specifications that ensure our meta-model (in `model`), and the `modelChange` class diagram, are appropriate to implement a workable change retrieval algorithm.

Figure 6 shows an excerpt of the `model` subsystem class diagram (with a link to the `Change` class in `modelChange`) that is navigated by the OCL expression of our example in Figure 4. Since the OCL expression does produce the added messages we wish to obtain and is consistent with the class diagram, we know that the meta-model is sufficient for this particular change detection rule.

Figure 5 shows an excerpt of the change taxonomy where our example change type (added message) is located. We see it is in the changed Sequence Diagram View, which may itself be composed (note the composition) of added messages but also added classifier roles, changed message actions, among others. Changed Classifier Role and Changed Message Action are further decomposed into subcategories that are not shown here and are available in Appendix B. The taxonomy has been designed so that we could define precise impact analysis rules for every leaf change category.

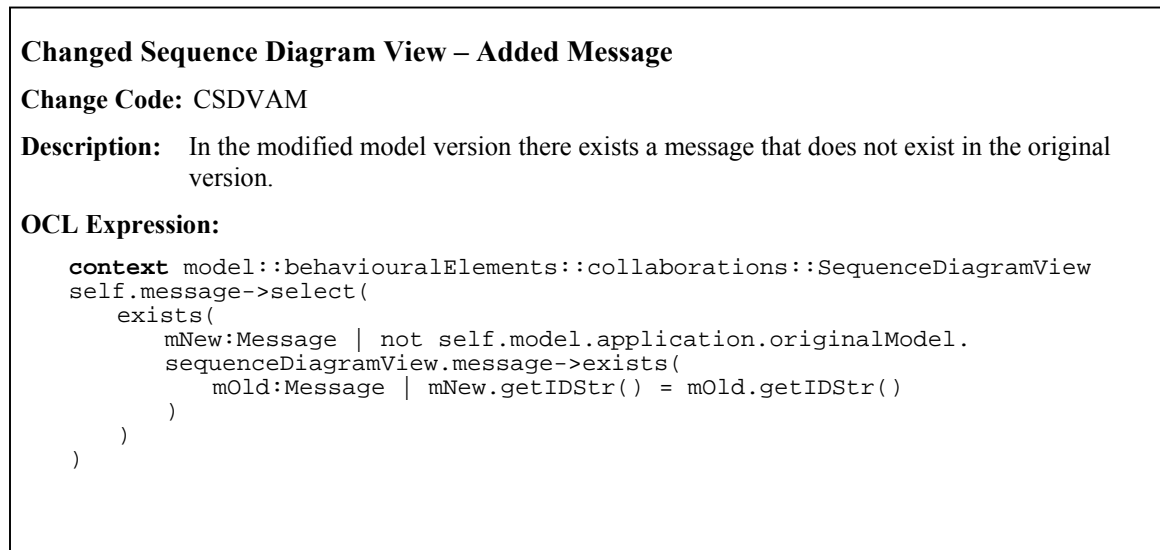


Figure 4 – Example Change Type

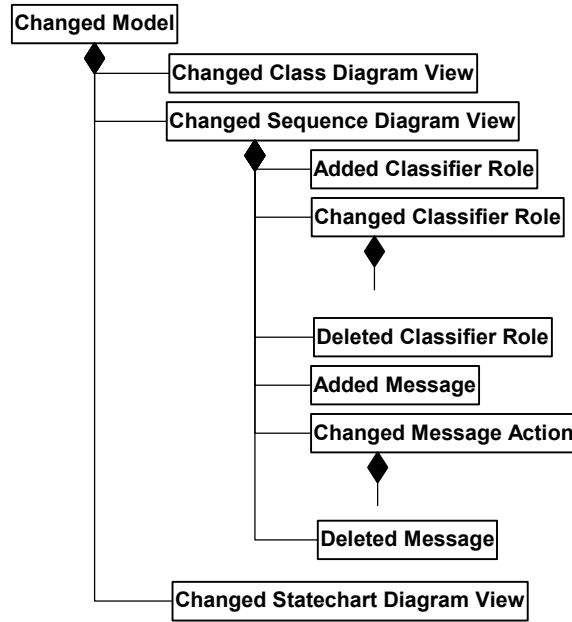


Figure 5 – Excerpt of Change Taxonomy

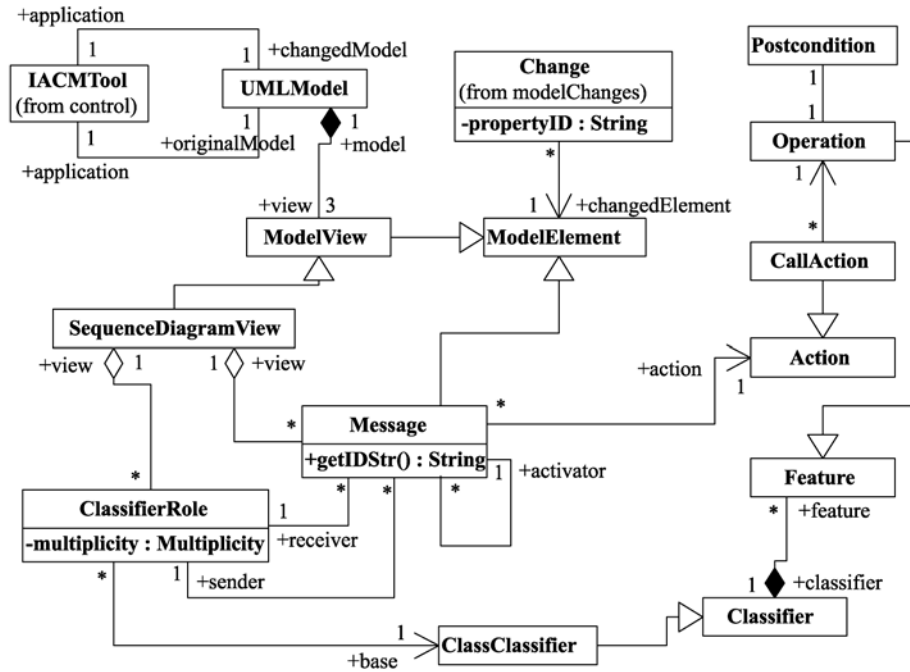


Figure 6 – Excerpt from the iACMTool Class diagram

7. IMPACT ANALYSIS RULES

Each impact analysis rule is a specification (using OCL) of how to derive several collections (i.e., OCL bags⁶) of elements, corresponding to elements of different types (e.g., classes, operations), that are potentially impacted by a particular change (e.g., added message). A model element is considered impacted by a change if a modification to that element or its implementation *may* be needed to accomplish a change (this cannot always be decided with certainty). There is one impact analysis rule for each type of change in the taxonomy.

Definition 3: Bag of impacted elements

Let E and E' be the set of all model elements in the original and modified model version, respectively. Then I is the bag of *impacted elements* (in the modified model version) resulting from that change such that $\forall i \in I, \exists e \in E \cap E'$ such that $e \neq i$ and there is a navigation path from e to i in the object diagram corresponding to the modified model version.

Though this is rare, note that the bag of impacted elements I may be empty, i.e., it is certain that no resulting changes are necessary to accomplish the change that caused the impact. The resulting changes to be made to an impacted model element must be of a type defined in the change taxonomy for the impacted element type.

Impact analysis rules are described in a structured and precise manner so that it is easy to review, refine, and change them, for example as the UML standard is evolving. A sample impact analysis rule is presented in Figure 7 by elaborating on our change detection rule example above (adding a message to a sequence diagram). The change title is presented first, followed by the corresponding change code (CSDVAM⁷), after which the pathname of the changed/added/deleted model element class is presented, followed by the property that has changed. In this case an instance of `SequenceDiagramView` (located inside the `model` subsystem) has been changed and one of its property has been changed: an

⁶ A collection with possibly several occurrences of an element [13]. Bags are derived because it is possible that an element is impacted in several ways by a particular change.

⁷ See example of change detection rule in Figure 4.

instance of `Message` has been added and linked to it. After the property is listed, the pathname of the impacted element class(es) is stated (`ClassClassifier`, `Operation`, and `Postcondition` in this case). A brief discussion follows that states the elements impacted, and under what conditions. The rationale for the change then states the reasons for the impacts. The changes potentially resulting from the impacts are then described and they translate into additional impact analysis rules being invoked. This is the way the transitive closure of impacts is explicitly modeled here: some rules invoke others as direct impacts lead to indirect ones [1]. These descriptions are followed by the OCL expression(s) describing the formal derivation of the impacted elements based on our meta-model (for the rule example in Figure 7, see meta-model excerpt in Figure 6). The first expression in our example (each expression being expressed in a **context**) uses the `let` operator to define two placeholders (variables) for navigation expressions capturing the added message and the sending operation in the class diagram, respectively. The added message is identified as the message having the `IDStr` (string uniquely identifying each model element and returned by the `getIDStr()` operation) corresponding to the changed property of the view associated with the change (`propertyID` in `Change`).

Change Title:	Changed Sequence Diagram – 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 we may not know the operation, 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 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 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.oclIsTypeOf(CallAction) then addedMessage.sender.base.operation->select(o:Operation o.equals(addedMessage.activator.callAction.operation)) else null endif) context modelChanges::Change - class addedMessage.sender.base context modelChanges::Change - operation sendingOperation context modelChanges::Change - postcondition sendingOperation.postcondition </pre>

Figure 7 – Impact Analysis Rule Example

In our example, the changed property is an added message in the `SequenceDiagramView`. Then, the operation that possibly sends the added message is identified. Note that the navigation expression first identifies the base class of the classifier role² that sends the added messages, as we want to identify the operation as described in the class diagram, and not the operation as it is used in the sequence diagram⁸. This identification involves selecting (the `select` operator) the method declaration in the class that corresponds to the

⁸ In this case, the navigation is simply: `addedMessage.activator.callAction.operation`. As in the official UML meta-model, operation invocations and declarations are modeled by the same `Operation` class.

invoked method in the sequence diagram, and is realized by the `equals()` operation in the OCL expression. This operation’s complexity stems from overloaded methods, as described in [4, 5], since UML sequence diagrams do not show parameter (and return) types. Once the added message and the sending operation have been identified, the propagation of the impact, to the sending class, the sending operation, and the postcondition of the sending operation, is described in three OCL expressions, each of them starting with the **context** keyword. Though they only return one element each in this example, those expressions return bags in the general case.

8. DISTANCE MEASURE

When impacts between model elements are indirect, following the general guidelines in [1], we suggest using a distance measure between the changed model elements and the impacted elements. In [1], it is stated that a common assumption⁹ 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 taxonomies, 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 determine impacted elements and then, in some cases, a number of impact analysis rules are invoked again on some of the directly impacted elements. We define the distance between a changed element and a given impacted element to be the number of impact analysis rules that had to be invoked to identify this impacted element. If we use Figure 8 as an example, we can see that the sets of impacted elements can be represented as the nodes of a tree whose arcs are impact analysis invocation rules. We reuse here the rule example in Figure 7 when a message is added to a sequence diagram. This rule triggers, for the impacted postcondition (`p1`), the changed postcondition rule (`CCOCPst`), thus leading to the identification of other impacted postconditions and operations. Only the first two depth levels of the tree are shown. The level in the tree of a given impacted element is the distance associated with this element,

⁹ This fundamental assumption seems reasonable, but empirical investigations are warranted to validate it.

e.g., $\text{distance}(p_2) = 2$. Such a distance measure could then be used to either sort impacts according to their distance from a given changed element or even to exclude impacted elements further than a certain distance set by the tool's user. If a model element is impacted several times, then the minimum distance can be used (i.e., the strongest impact).

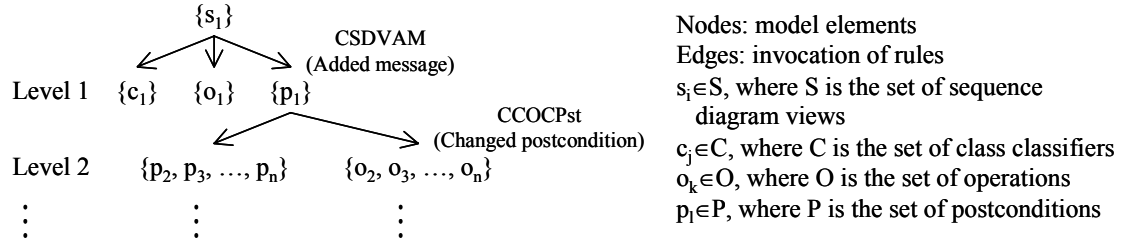


Figure 8 – Example distance between a changed element and an impacted element

9. CASE STUDY

We have selected an Automated Teller Machine (ATM) as a case study: The customer inserts his/her card, enters a PIN and then can performs transactions such as withdrawal and deposit before a receipt is issued by the ATM at the end of all the transactions. The first version of UML documents contains a class diagram (19 classes such as ATM, Bank, Withdrawal) and a use case diagram (15 use cases such as Transaction, Withdrawal, GetPIN, CardNotReadable) – each use case being associated with a sequence diagram. Most of the sequence diagrams contain from 3 to 7 messages (e.g., sequence diagrams for use cases ATMStartUp and ATMShutOff contain 7 and 3 messages respectively), the sequence diagram for use case Transaction being the most complicated one with 22 messages. 15 attributes and 18 operations appear in the class diagram, and classes are related by inheritance (4), association (11) and dependency (3) relationships.

We made 10 realistic, logical changes to the original version of the UML diagrams. These logical changes are of three types: requirements changes (5), design improvements (2), and error corrections (3). They result in 70 model element changes¹⁰ (described in Appendix E), out of which 54 have shown impacted elements. Let us take a few examples of logical changes and describe them. One logical change stems from our need

¹⁰ The distribution of these changes for the elements in the taxonomy can be found in Appendix E.

to be able to keep track of how many times per session a user attempts to enter the PIN – after 3 invalid PIN’s the card will be retained. This logical change translates into 11 model element changes. Another logical change is to change the ATM state’s representation from an integer to an enumeration class, and results into 34 model element changes. Two other logical changes concern changes in the legal states of the system and translate into new association end multiplicities in the class diagram: (1) An account can be owned by at most two customers and at least one customer (a multiplicity is changed from $1..*$ to $1,2$); (2) A customer must belong to a bank and a customer can only belong to one bank (a multiplicity is changed from $0..*$ to 1). A last logical change example (design) consists in making class `Account` abstract since only its subclasses are instantiated (e.g., `Saving`). A complete description of all logical changes can be found in Appendix E.

Let us consider the impacted operations, when accounting for all changed model elements taken together, and their distance to the changed model elements. Figure 9 plots, for each of the 54 model element changes, a curve/point representing the cumulative number of impacted operations (y-axis) for each distance value (x-axis). A first, clearly visible result is that only four curves are visible as only four changes propagate impacts further than a distance of 2. The reason is that the impacted elements at distance one and two that do not propagate are classes, and the class is not an impact related element of any other element in a class diagram (see Definition 2). The rationale is that the propagation of impacts from class to class is already addressed since operations and attributes are impact related elements of the operations that call/use them.

More importantly, when there is propagation of impacts, Figure 9 clearly shows that the curves are not exponential, as suggested in [1], but rather linear. This is important as it suggests that our impact analysis rules rather precise. Also, the maximum distance for impacted elements is limited to six. Though more case studies are necessary to draw definitive conclusions, we can state that these results are probably due to our use of semantic-based impact rules, instead of connectivity graphs (see [1]), that allow a more refined identification of impacted elements and reduce false-positives.

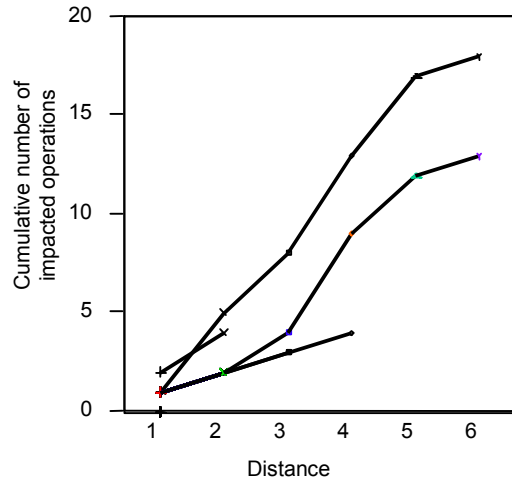


Figure 9 – Cumulative number of impacted operations vs. distance

In the analysis above we perform an overall impact analysis for all logical changes but, if we were in a situation where we would have to decide on which logical changes to implement in a next release, we might want to perform the same analysis for each logical change in isolation to evaluate its individual cost. Also, we only looked at the cumulative number of operations but the same graph could be plotted for classes or even for all model elements impacted together. We provide such diagrams in Appendix E and the results clearly show that the curves are very similar to the one in Figure 9 (though with significantly different scales on the Y-axis).

10. CONCLUSIONS

We present in this report a methodology supported by a prototype tool (iACMTool) to tackle the impact analysis and change management of analysis/design documents in the context of UML-based development. Consistency rules between UML diagrams, automated change identification and classification between two versions of a UML model, as well as impact analysis rules have been formally defined by means of OCL constraints on an adaptation of the UML meta-model.

Our impact analysis methodology and tool are assessed through a case study, thus providing an initial demonstration of its feasibility and practicality. Results are encouraging as it is shown that, with impact rules based carefully 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.

We also define a distance measure to be able to sort impacts, according to their likelihood of occurrence, based on the distance between changed model elements and impacted elements. Whether this measure is a good heuristic will have to be empirically validated.

Though we made a conscious effort 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 we gain more experience, especially by applying our change impact analysis strategy to additional case studies. All three types of rules have been defined using OCL on the UML metamodel and we expect that such precise and formal definitions will help refine and evolve our methodology.

ACKNOWLEDGEMENTS

This work was partly supported by Telcordia Technologies. Lionel Briand and Yvan Labiche were further supported by NSERC operational grants. This work is part of a larger project (TOTEM) on testing object-oriented systems with the UML (TOTEM: www.sce.carleton.ca/Squall/Totem). We would like to thank Karin Buist for her help in defining and implementing (in the UML diagrams) the logical changes.

REFERENCES

- [1] S. A. Bohner, "Software Change Impacts - An Evolving Perspective," *Proc. IEEE International Conference on Software Maintenance*, Montreal, Canada, pp. 263-272, 3-6 October, 2002.
- [2] S. A. Bohner and R. S. Arnold, "An Introduction to Software Change Impact Analysis," in S. A. Bohner and R. S. Arnold, Eds., *Software Change Impact Analysis*, IEEE Computer Society, 1996, pp. 1-25.
- [3] S. A. Bohner and R. S. Arnold, *Software Change Impact Analysis*, IEEE Computer Society Press, 1996.
- [4] L. Briand, Y. Labiche and 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, a short version appeared in the proceedings of ICSM 2002.
- [5] L. C. Briand, Y. Labiche and G. Soccar, "Automating Impact Analysis and Regression Test Selection Base on UML Designs," *Proc. IEEE International Conference on Software Maintenance (ICSM)*, Montreal (Canada), IEEE Computer Society, pp. 252-261, October 3-6, 2002.
- [6] L. C. Briand, J. Wust and H. Lounis, "Using Coupling Measurement for Impact Analysis in Object-Oriented Systems," *Proc. IEEE International Conference on Software Maintenance*, Oxford, England, pp. 475-482, September, 1999.
- [7] B. Bruegge and A. H. Dutoit, *Object-Oriented Software Engineering - Conquering Complex and Challenging Systems*, Prentice Hall, 2000.
- [8] T. J. Grose, S. A. Brodsky and G. C. Doney, *Mastering XMI: Java Programming with XMI, XML, and UML*, John Wiley & Sons, 2002.
- [9] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima and C. Chen, "Change Impact Identification in Object Oriented Software Maintenance," *Proc. IEEE International Conference on Software Maintenance*, IEEE, pp. 202-211, 1994.
- [10] OMG, "Unified Modeling Language (UML)," Object Management Group V1.4, www.omg.org/technology/uml/, 2001.
- [11] TogetherSoftTM, "Together", www.togethersoft.com.
- [12] A. Von Mayrhauser and N. Zhang, "Automated Regression Testing using DBT and Sleuth," *Journal of Software Maintenance*, vol. 11 (2), pp. 93-116, 1999.
- [13] J. Warmer and A. Kleppe, *The Object Constraint Language*, Addison-Wesley, 1999.

Appendix A System Model

The system models are presented in Figure A1 to Figure A22 below.

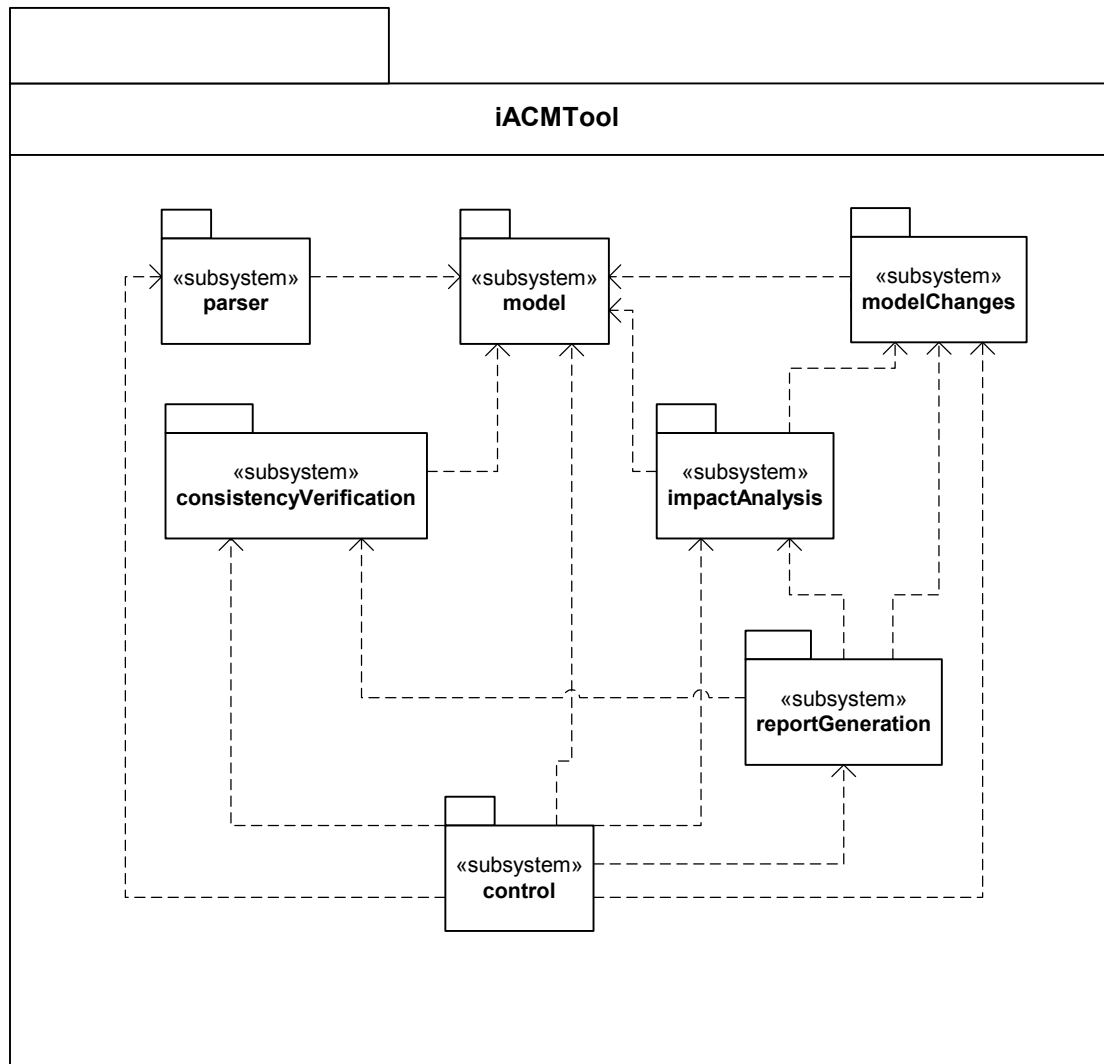


Figure A1: iACMTool – Main packages.

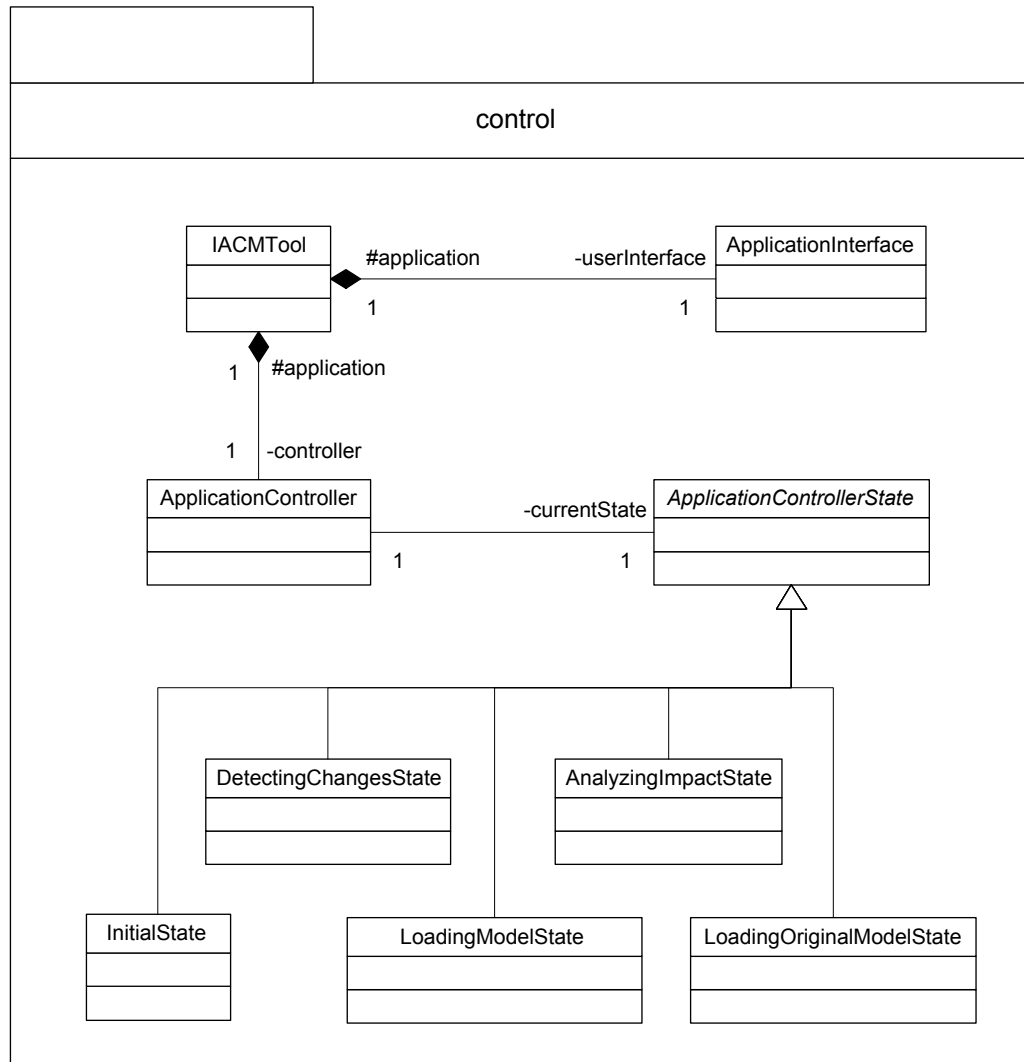


Figure A2: *iACMTool::control package.*

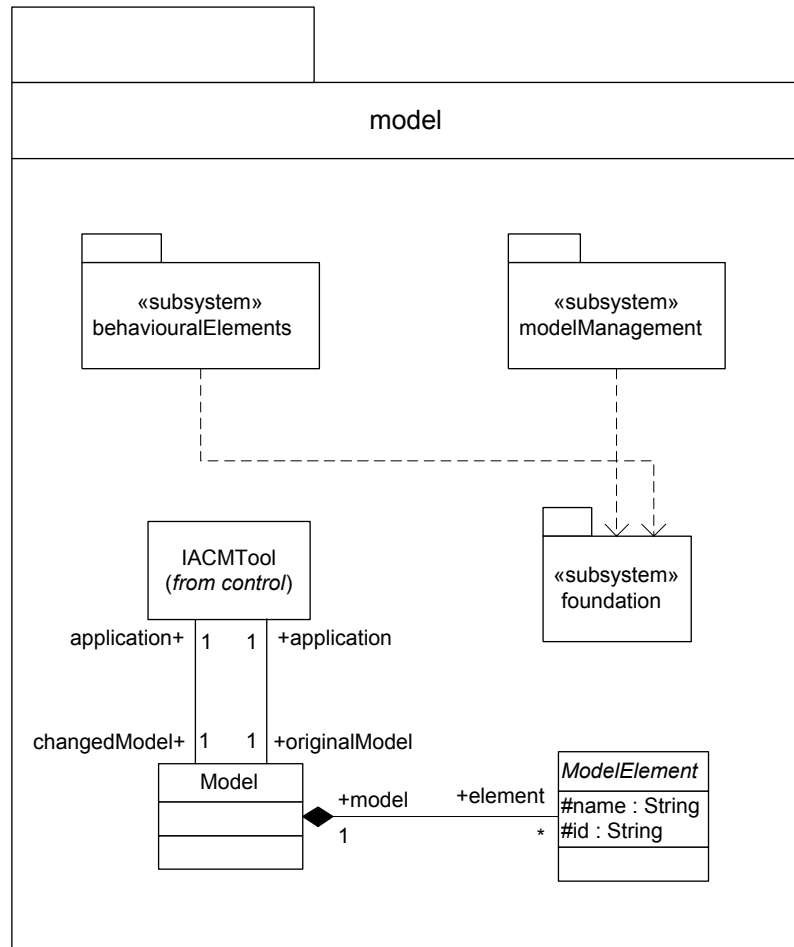


Figure A3: *iACMTool::model package.*

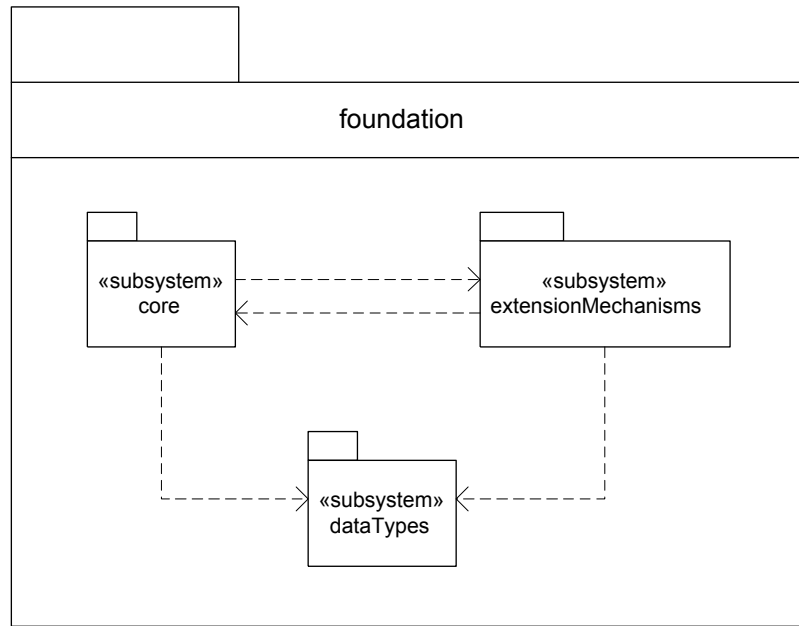


Figure A4: `iACMTool::model::foundation` *package*.

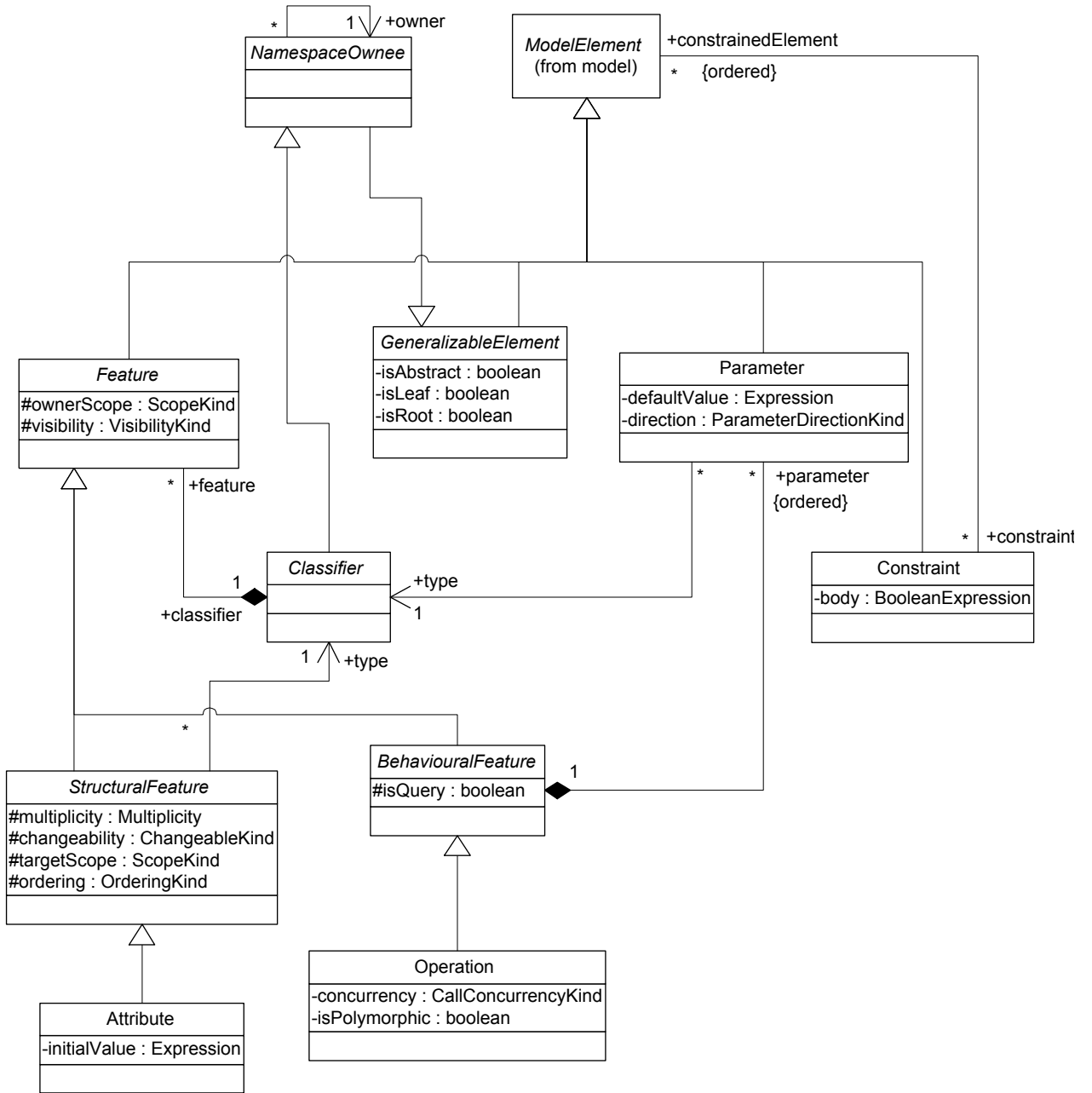


Figure A5: iACMTool::model::foundation::core – main.

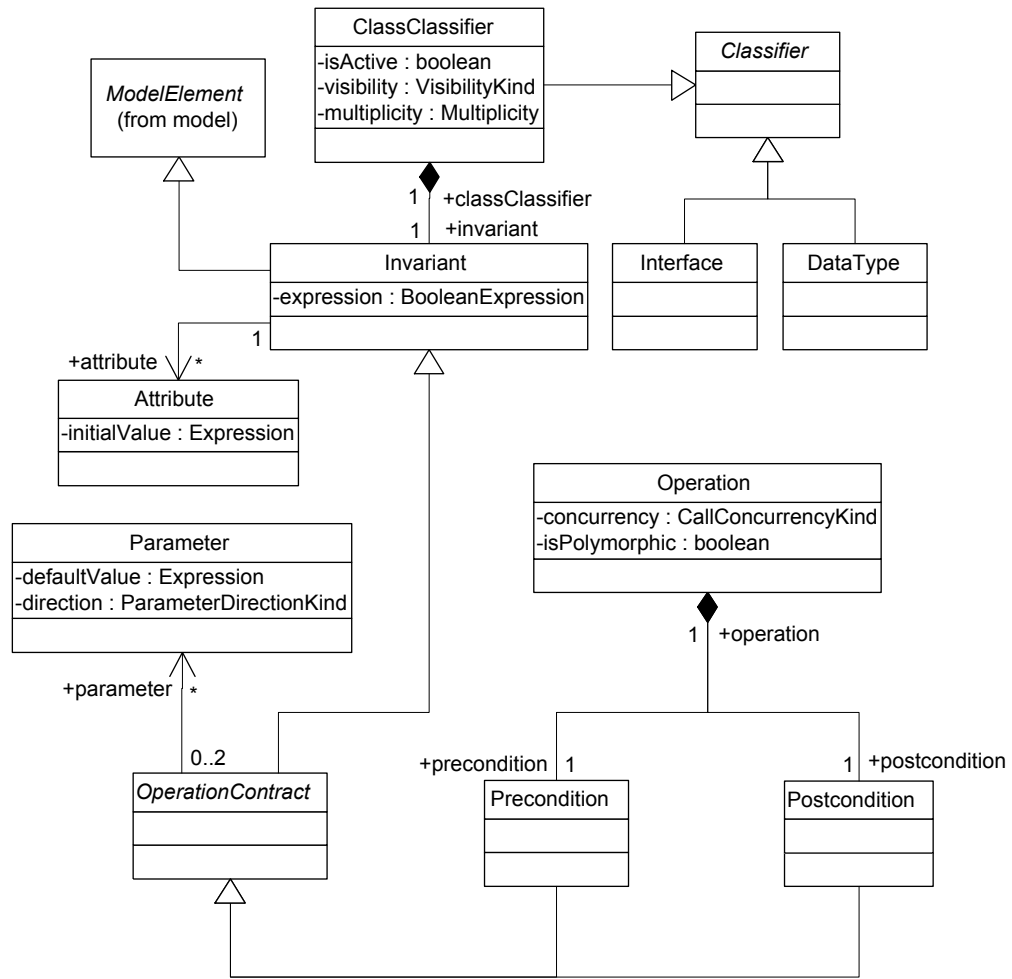


Figure A6: iACMTool::model::foundation::core – classifiers.

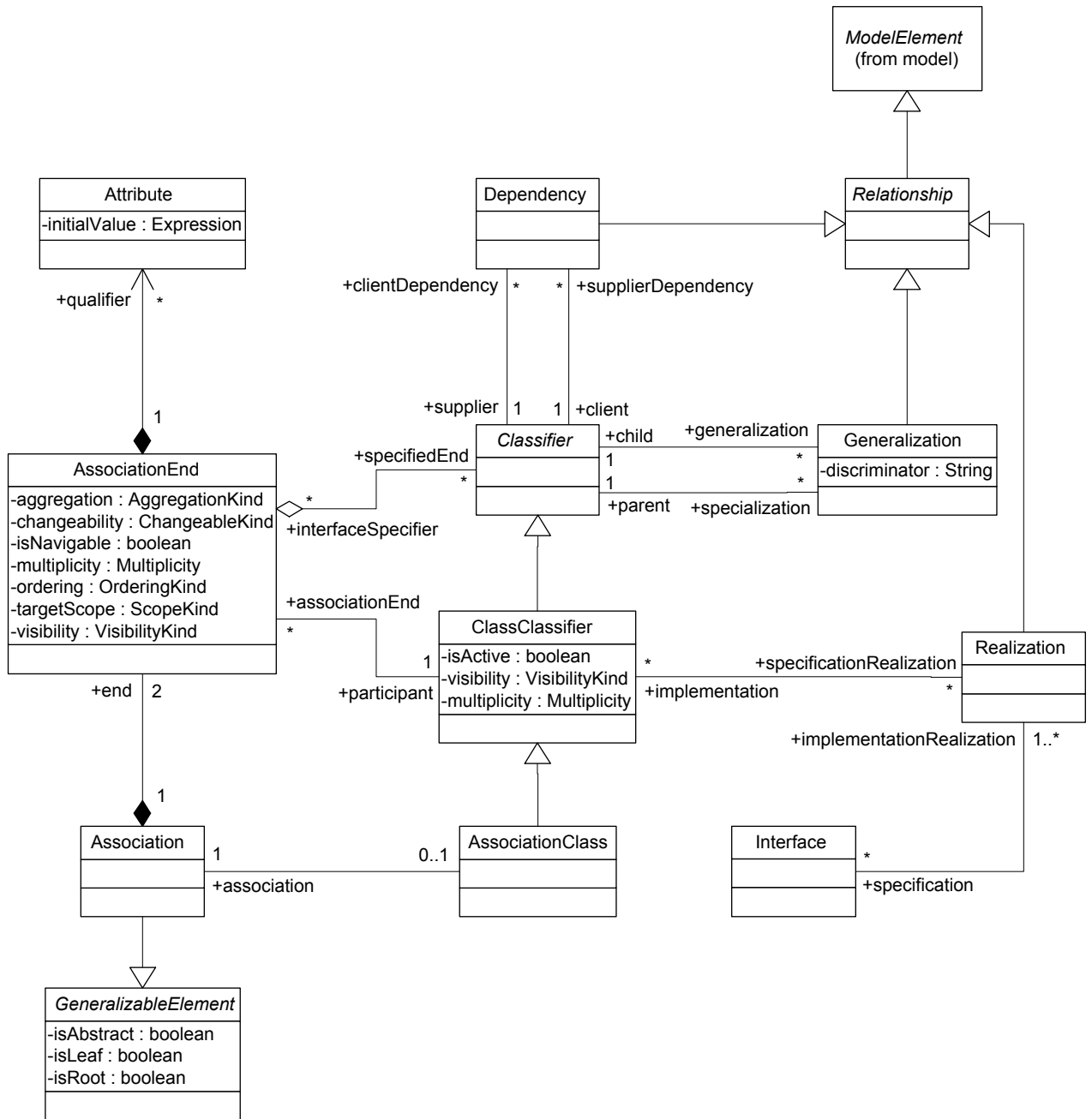


Figure A7: iACMTool::model::foundation::core – *relationships*.

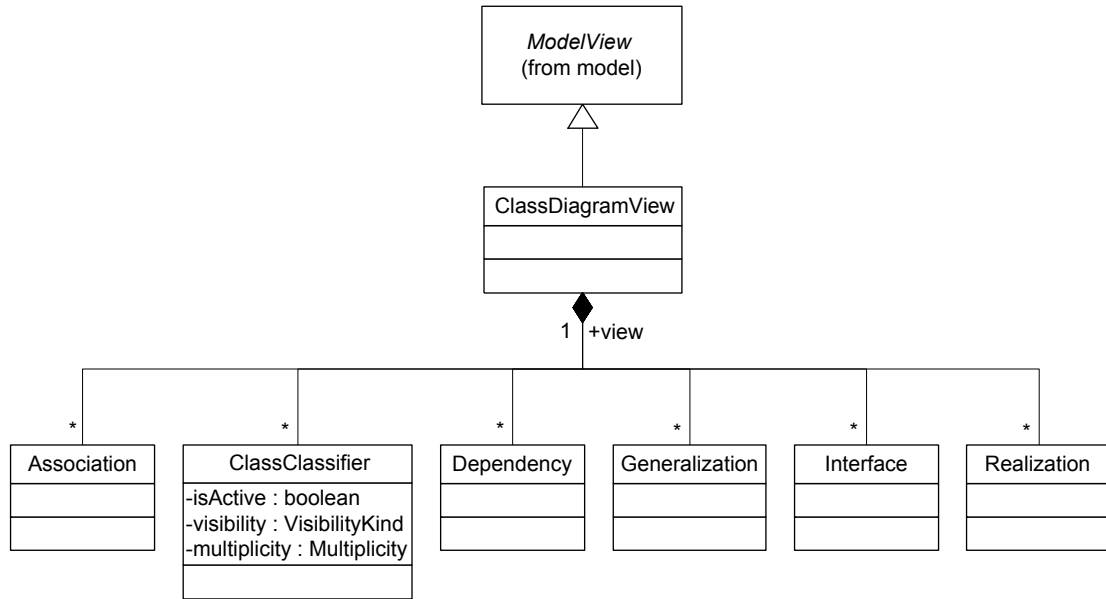


Figure A8: `iACMTool::model::foundation::core` – *class diagram view*.

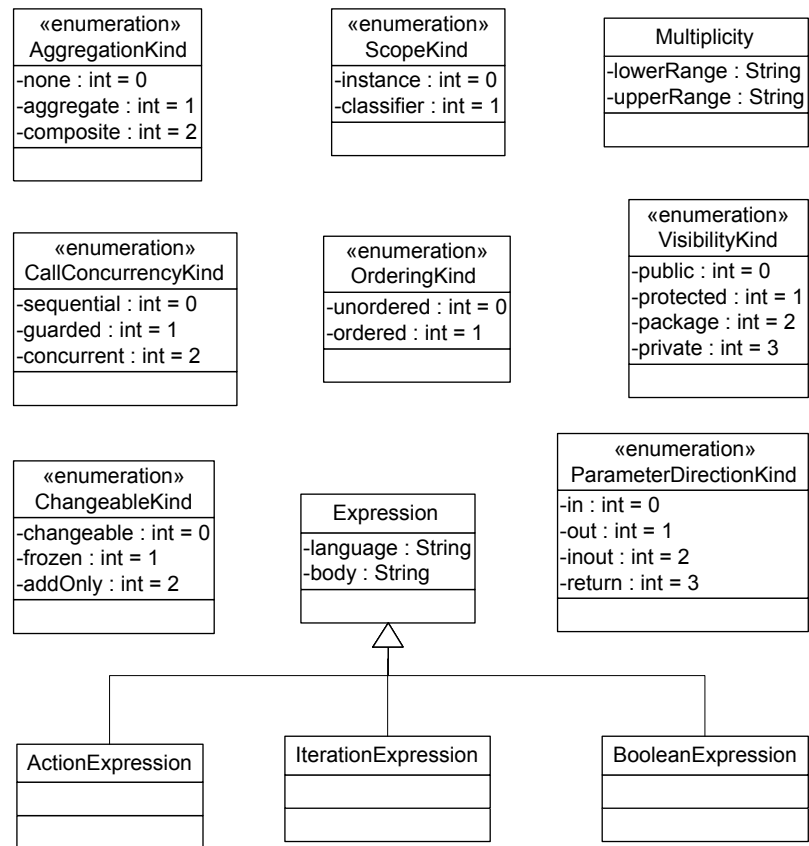


Figure A9: `iACMTool::model::foundation::dataTypes` package.

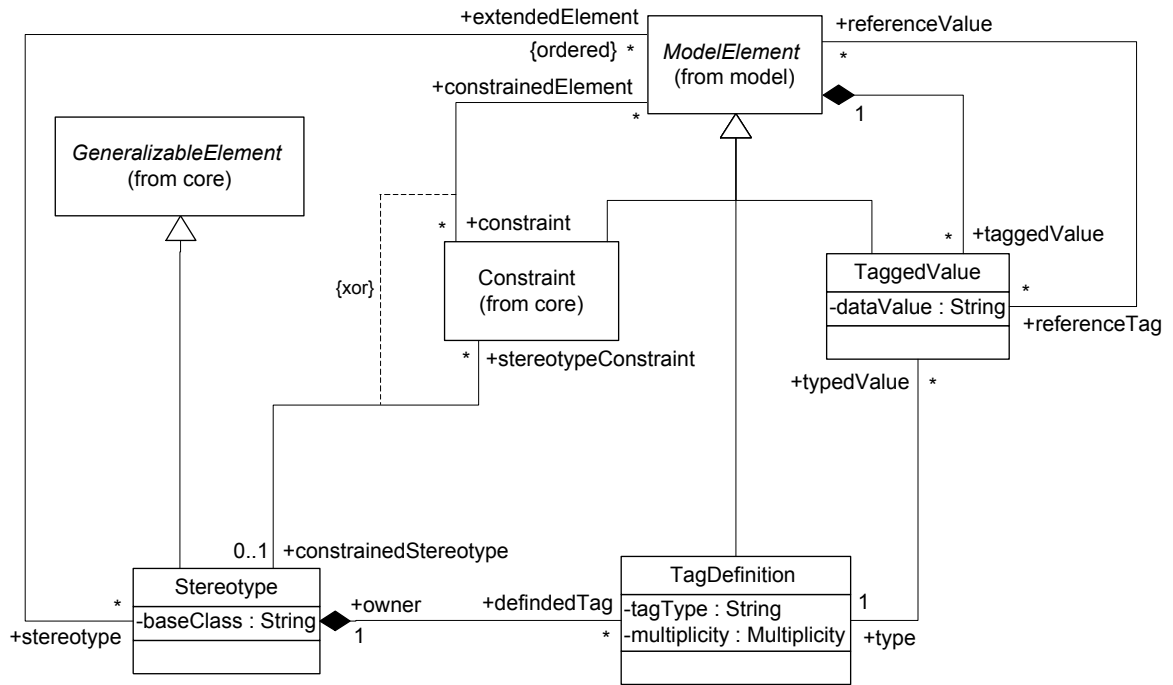


Figure A10: `iACMTool::model::foundation::extensionMechanism` package.

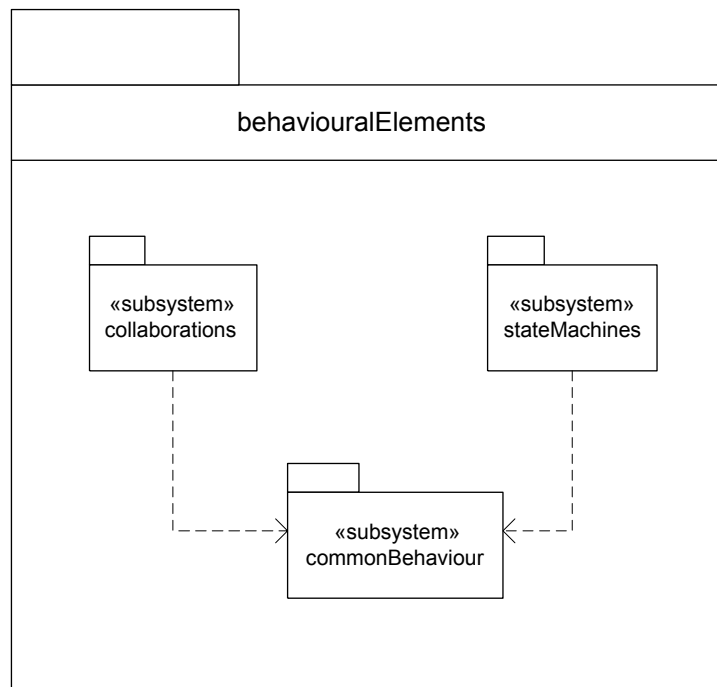


Figure A11: `iACMTool::model::behaviouralElements` package.

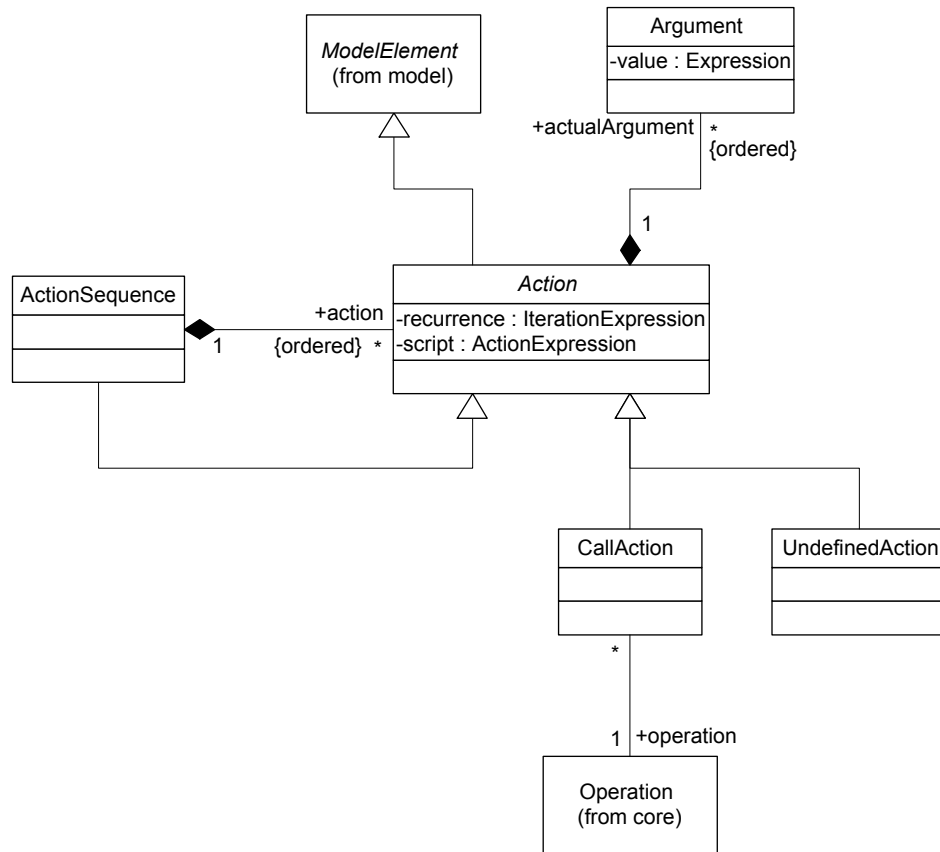


Figure A12: `iACMTool::model::behaviouralElements::commonBehaviour` package.

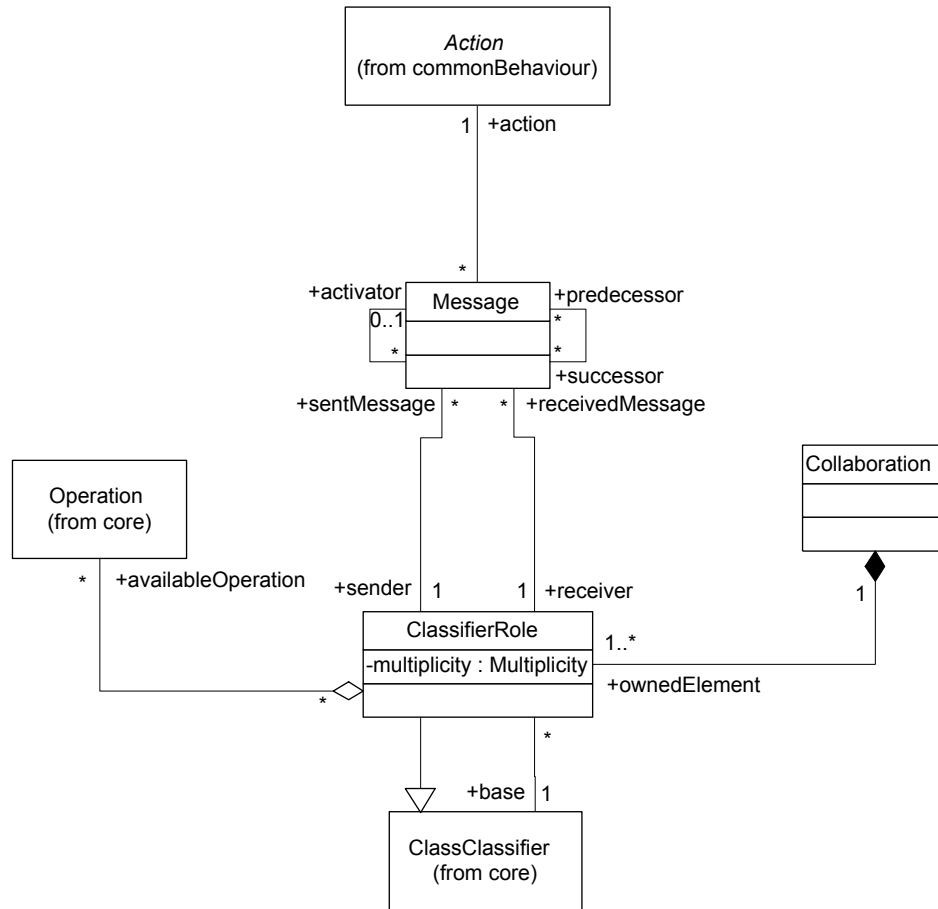


Figure A13: iACMTool::model::behaviouralElements:: collaborations - roles.

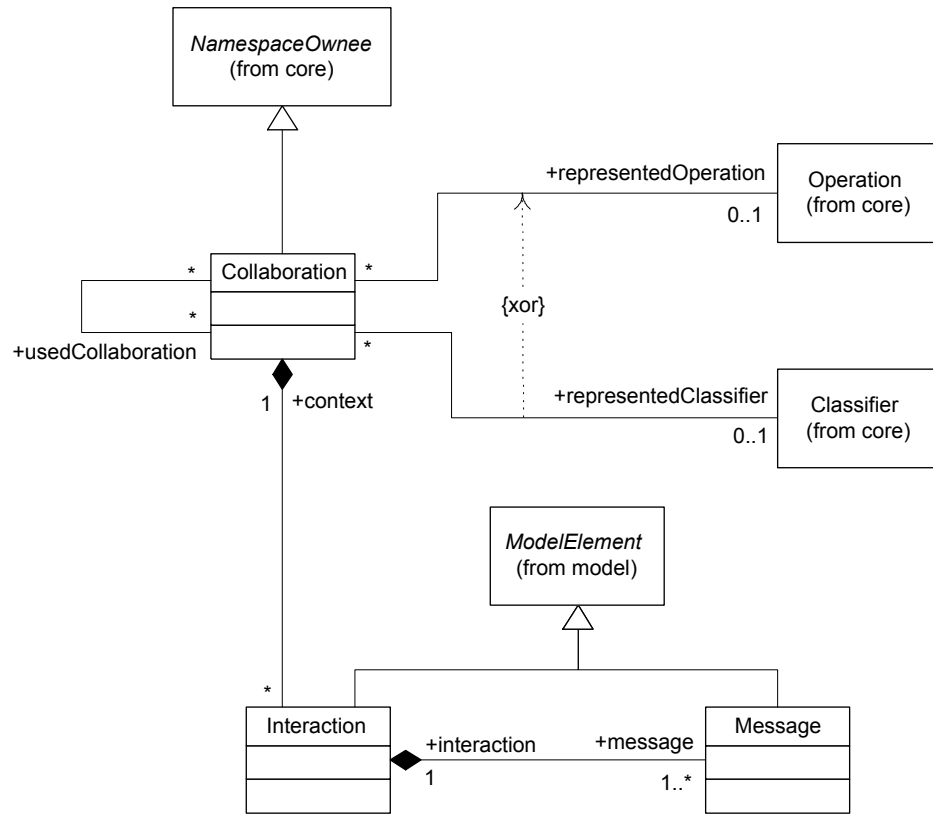


Figure A14: `iACMTool::model::behaviouralElements::collaborations` - *interactions*.

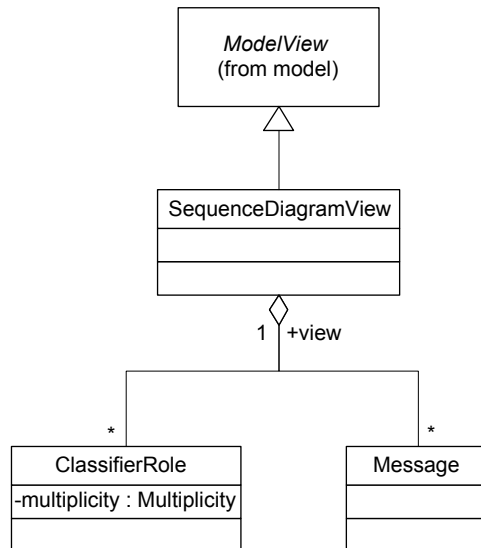


Figure A15: `iACMTool::model::behaviouralElements::collaborations` - *sequence diagram view*.

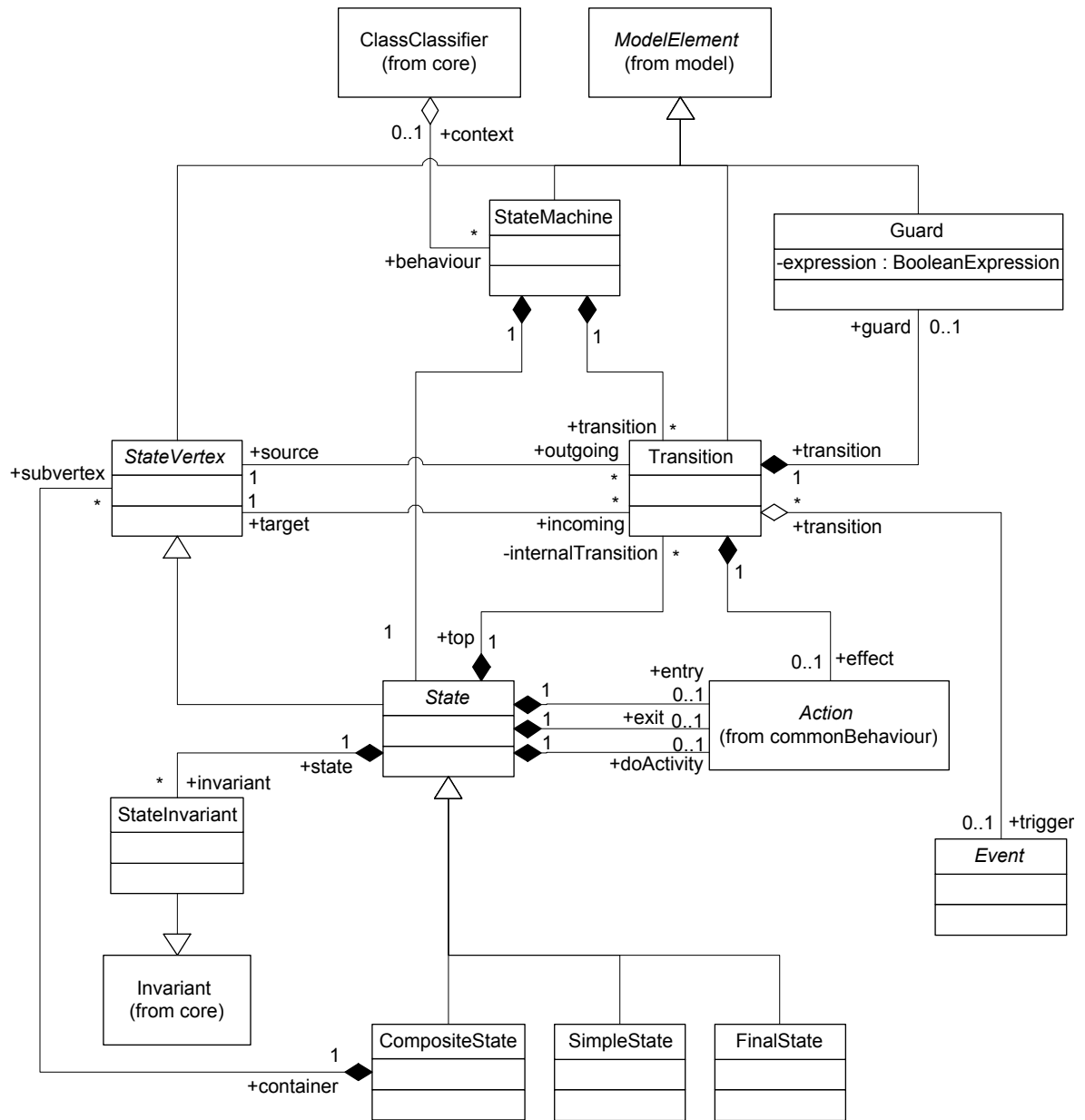


Figure A16: iACMTool::model::behaviouralElements::stateMachines - main.

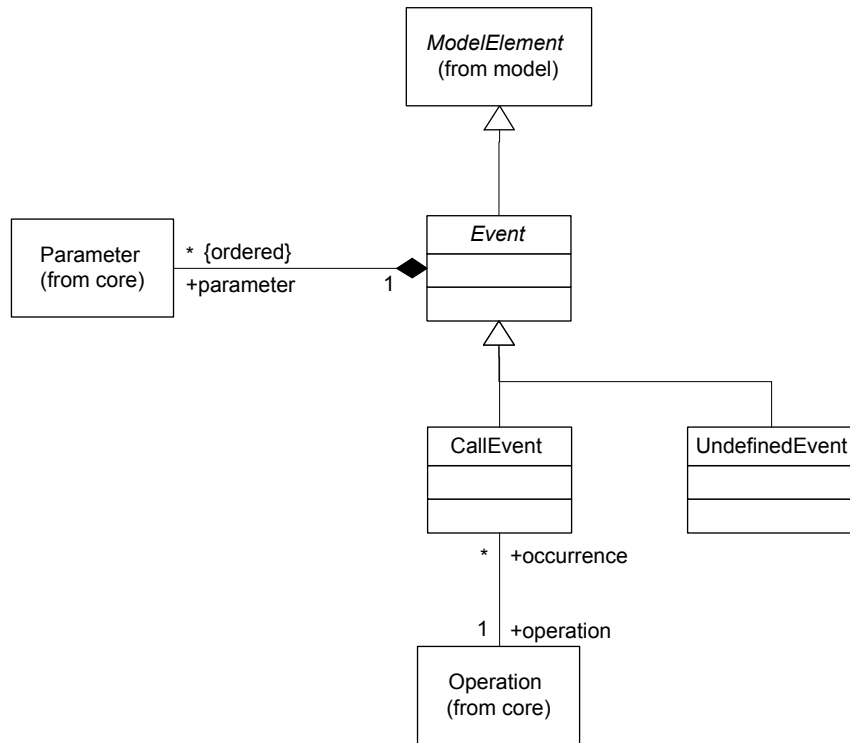


Figure A17: `iACMTool::model::behaviouralElements::stateMachines - events.`

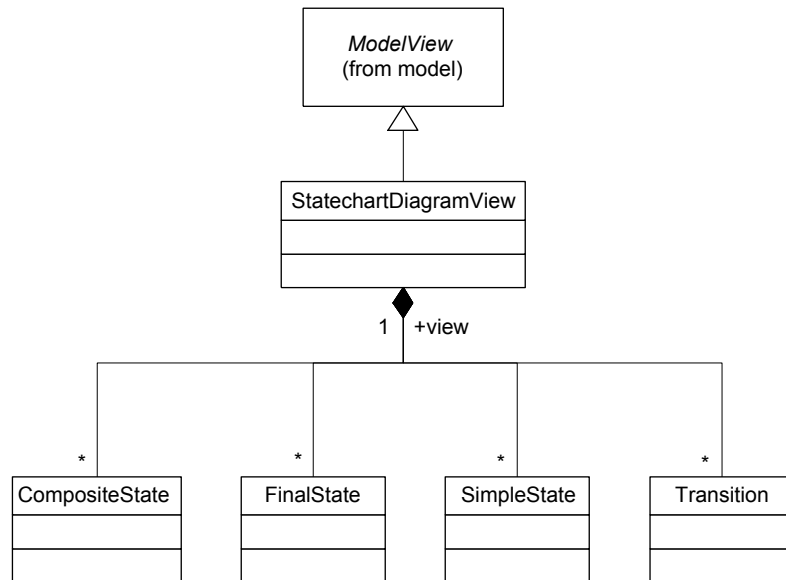


Figure A18: `iACMTool::model::behaviouralElements::stateMachines - statechart diagram view.`

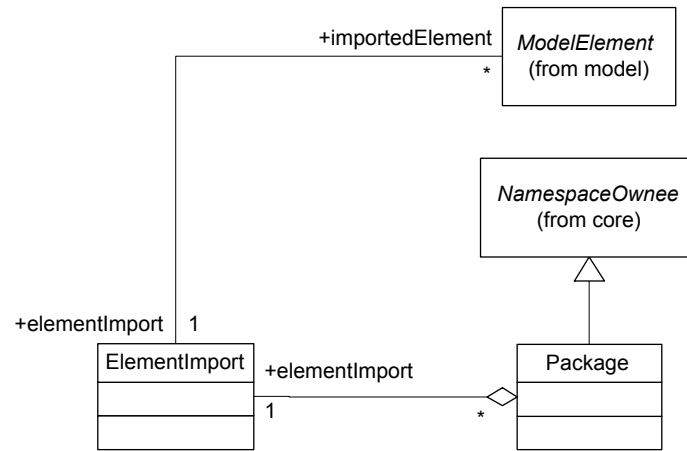


Figure A19: `iACMTool::model::modelManagement` package.

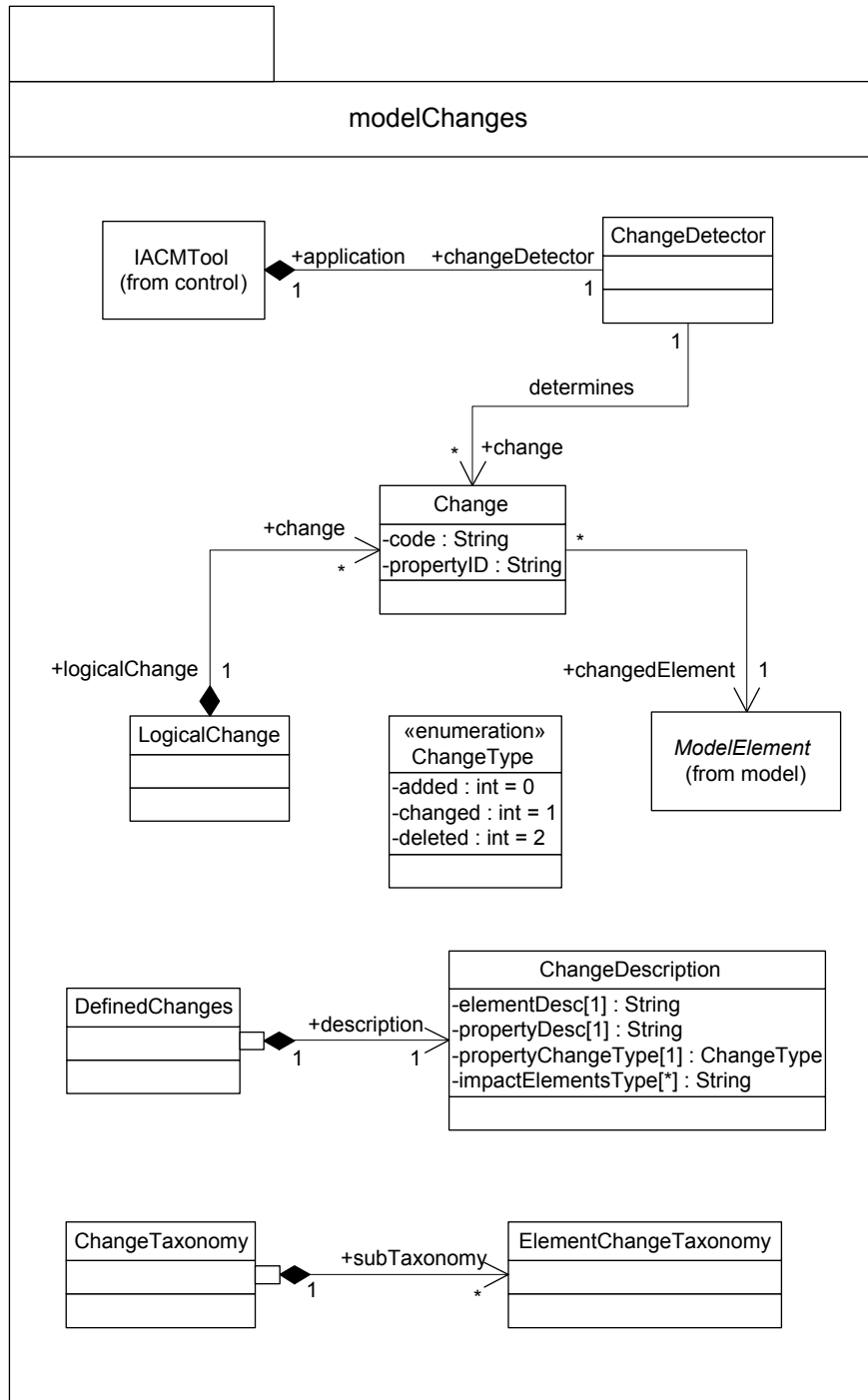


Figure A20: *iACMTool::modelChanges package.*

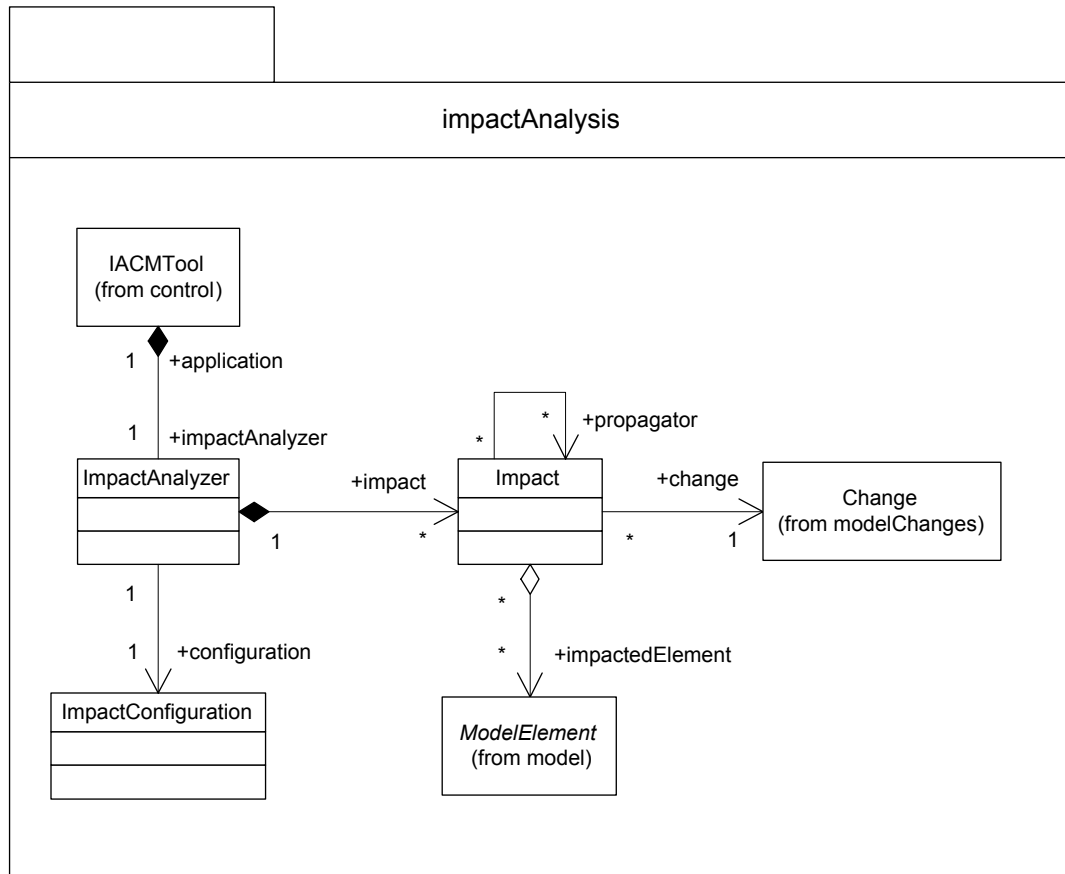


Figure A21: `iACMTool::impactAnalysis` package.

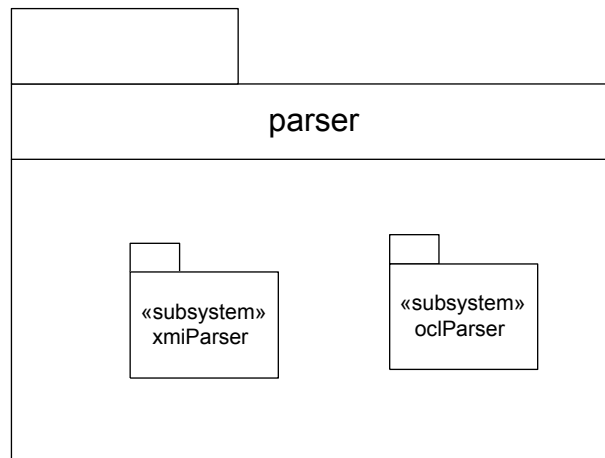


Figure A22: `iACMTool::parser` package.

Appendix B Change Detection

B.1 Change Taxonomy

A conceptual model of the change taxonomy is presented in Figure B1 to Figure B12 below. This is followed by the description of the changes (leaf nodes in the conceptual model).

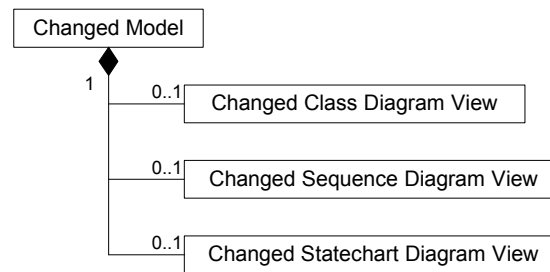


Figure B1: *Changed Model.*

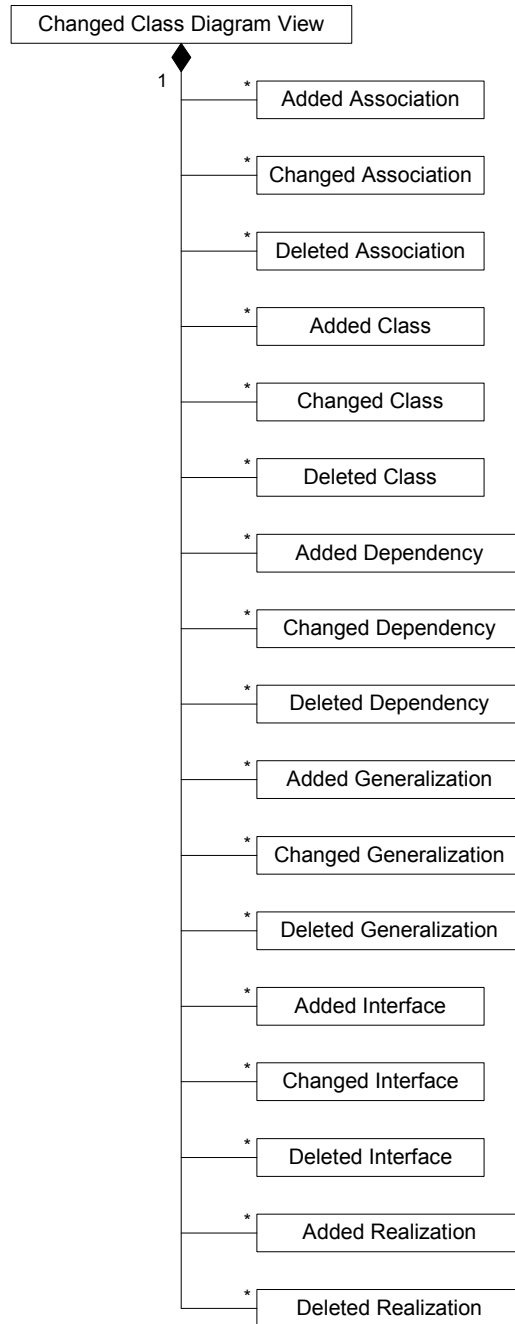


Figure B2: *Changed Class Diagram View.*

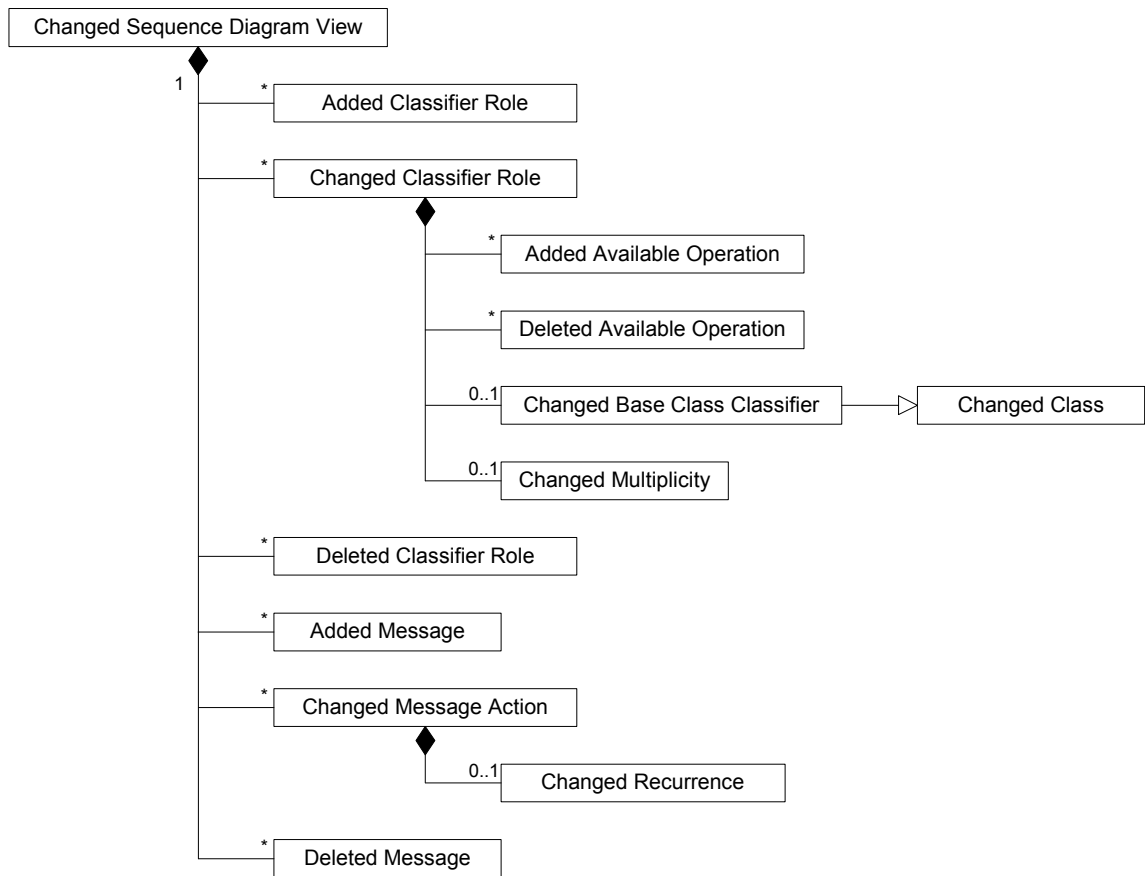


Figure B3: *Changed Sequence Diagram View.*

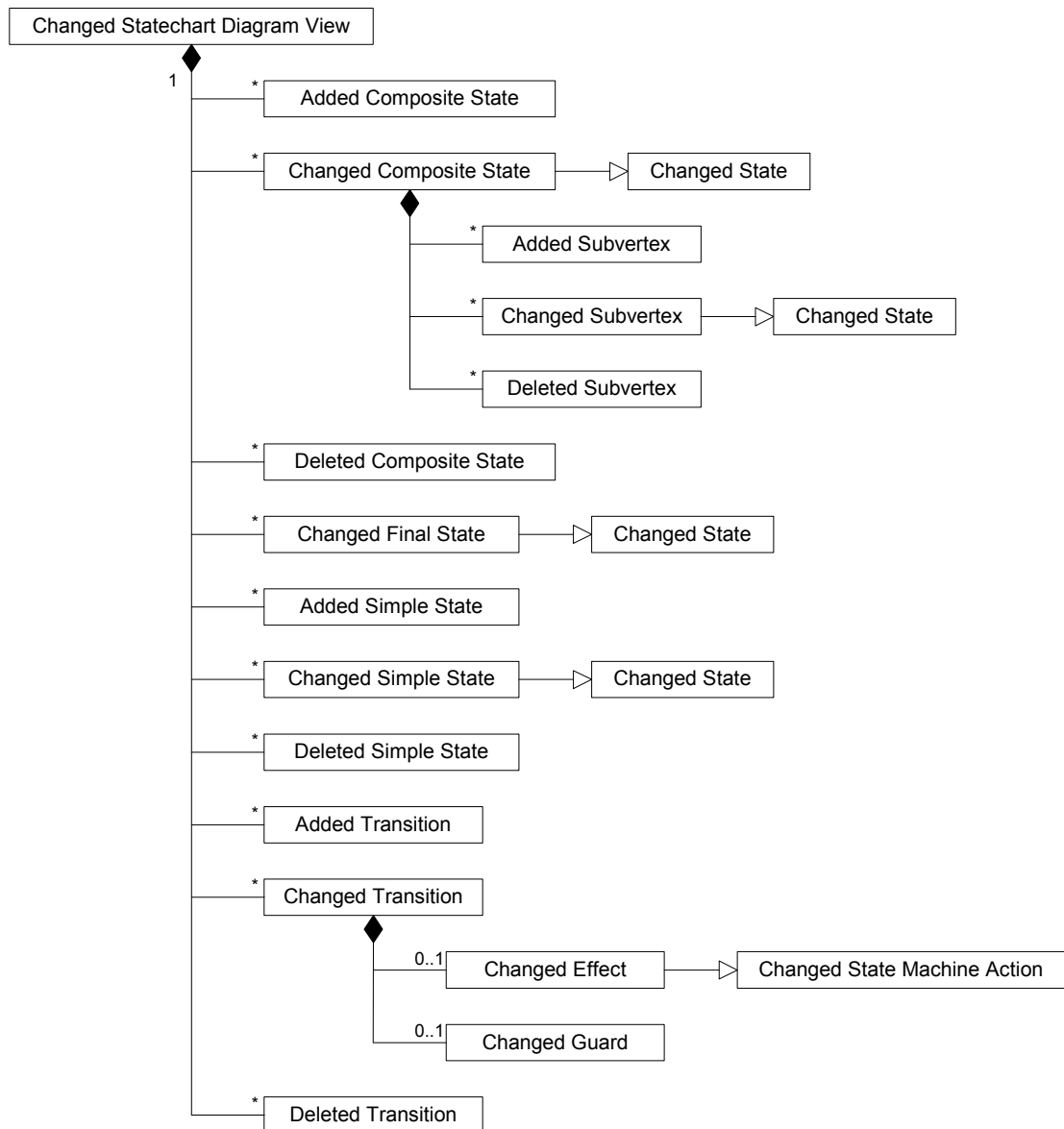


Figure B4: *Changed Statechart Diagram View.*

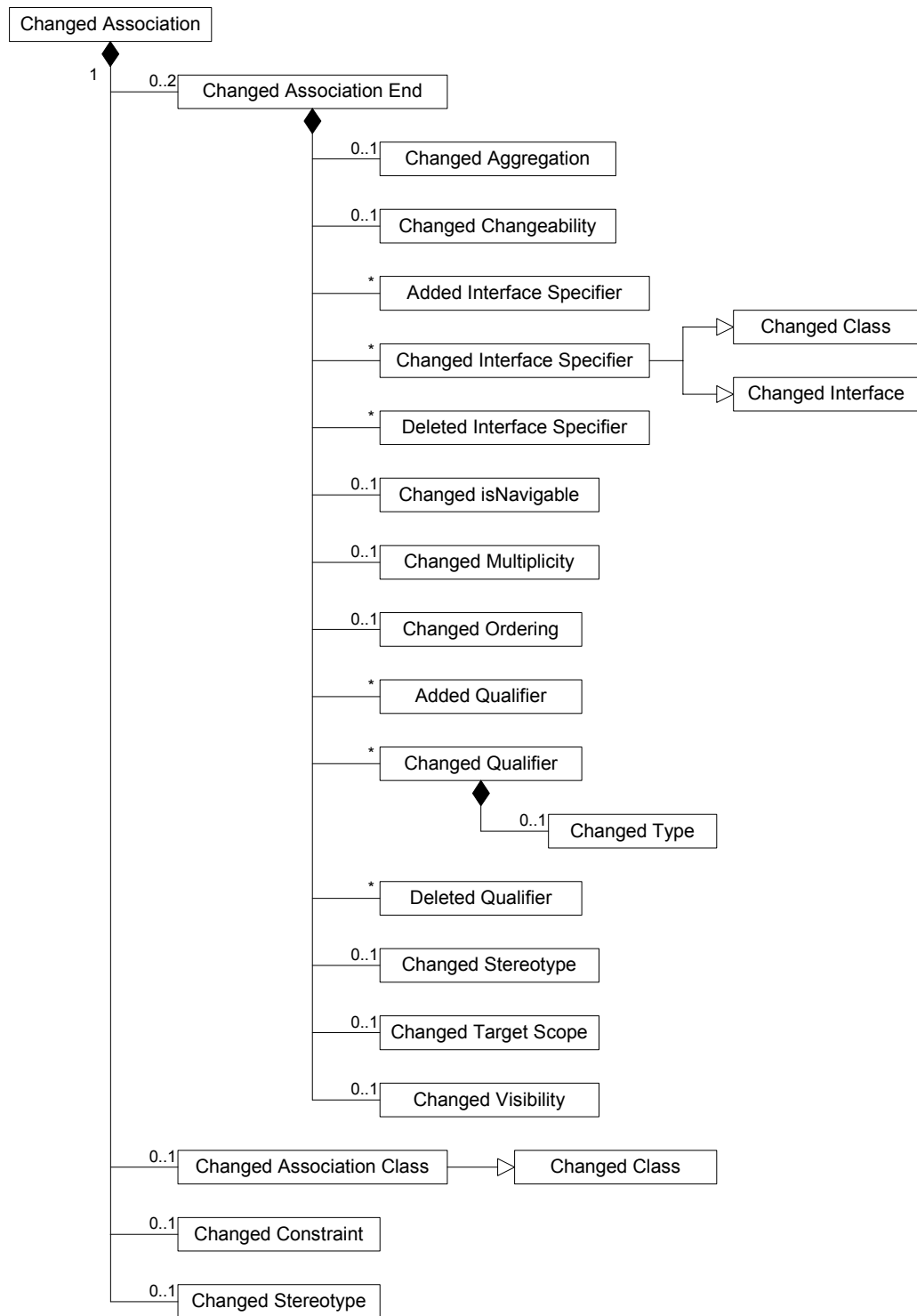


Figure B5: *Changed Association.*

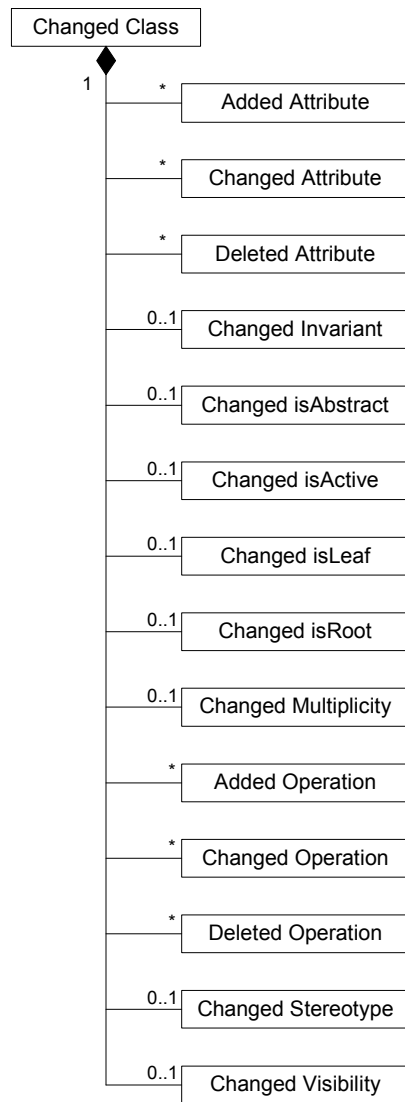


Figure B6: *Changed Class.*

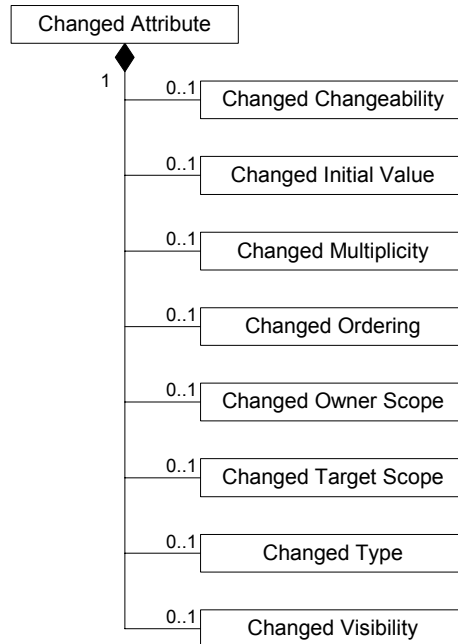


Figure B7: *Changed Attribute.*

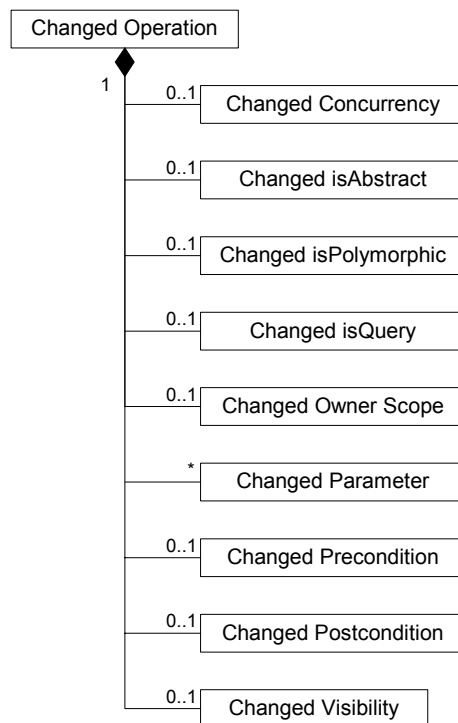


Figure B8: *Changed Operation.*

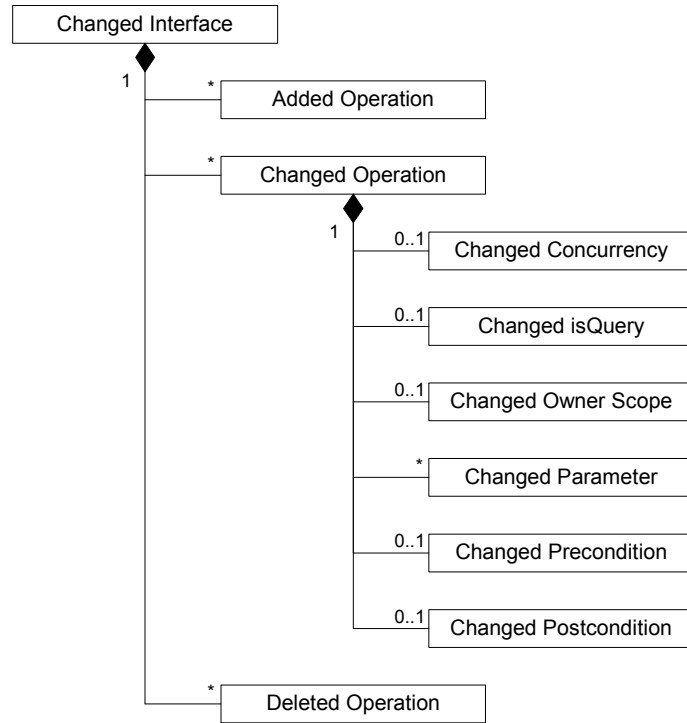


Figure B9: *Changed Interface.*

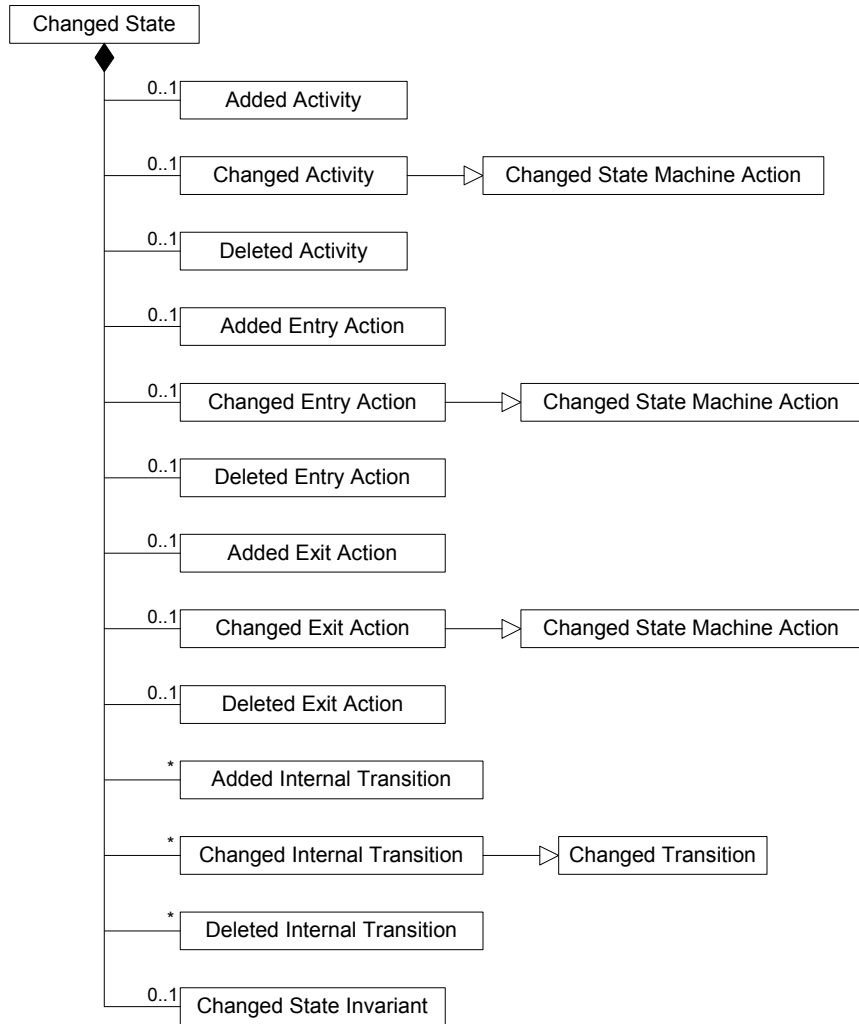


Figure B10: *Changed State.*

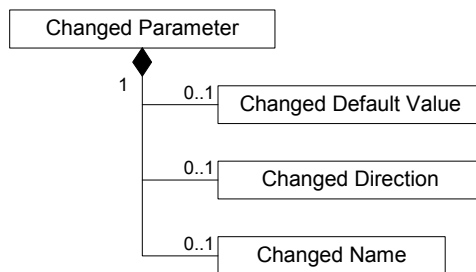


Figure B11: *Changed Parameter.*

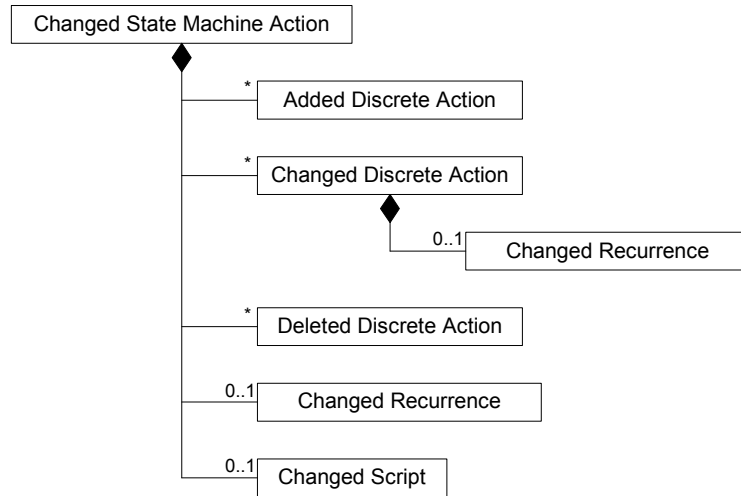


Figure B12: *Changed State Machine Action.*

B.2 Change Detection Rules

Here a brief discussion for each change is provided, followed by an OCL expression that defines the change. The modified version of the model is the version context for the rules below.

1. Changed Class Diagram View – Added Association

Change Code: CCDVAA

Description: In the modified model version there exists an association relationship that does not exist in the original version.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.association->exists(a1:Association|not self.model.
application.originalModel.classDiagramView.
association->exists(a2:Association|
a1.getIDStr() = a2.getIDStr()))`

2. Changed Class Diagram View – Deleted Association

Change Code: CCDVDA

Description: In the original model version there exists an association relationship that does not exist in the modified version.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.model.application.originalModel.classDiagramView.
association->exists(a1:Association|not
self.association->exists(a2:Association|
a1.getIDStr() = a2.getIDStr()))`

3. Changed Class Diagram View – Added Class

Change Code: CCDVAC

Description: In the modified model version there exists a class that does not exist in the original version.

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

4. Changed Class Diagram View – Deleted Class

Change Code: CCDVDC

Description: In the original model version there exists a class that does not exist in the modified version.

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

5. Changed Class Diagram View – Added Dependency

Change Code: CCDVAD

Description: In the modified model version there exists a dependency relationship that does not exist in the original version.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.dependency->exists(d1:Dependency|not
self.model.application.originalModel.classDiagramView.
dependency->exists(d2:Dependency|
d1.getIDStr() = d2.getIDStr()))`

6. Changed Class Diagram View – Deleted Dependency

Change Code: CCDVDD

Description: In the original model version there exists a dependency relationship that does not exist in the modified version.

OCL Expression: **context** model::foundation::core::ClassDiagramView
self.model.application.originalModel.classDiagramView.
dependency->exists(d1:Dependency|not self.
dependency->exists(d2:Dependency|
d1.getIDStr() = d2.getIDStr()))

7. Changed Class Diagram View – Added Generalization

Change Code: CCDVAG

Description: In the modified model version there exists a generalization relationship that does not exist in the original version.

OCL Expression: **context** model::foundation::core::ClassDiagramView
self.generalization->exists(g1:Generalization|not
self.model.application.originalModel.classDiagramView.
generalization->exists(g2:Generalization|
g1.getIDStr() = g2.getIDStr()))

8. Changed Class Diagram View – Deleted Generalization

Change Code: CCDVDG

Description: In the original model version there exists a generalization relationship that does not exist in the modified version.

OCL Expression: **context** model::foundation::core::ClassDiagramView
self.model.application.originalModel.classDiagramView.
generalization->exists(g1:Generalization|not
self.generalization->exists(g2:Generalization|
g1.getIDStr() = g2.getIDStr()))

9. Changed Class Diagram View – Added Interface

Change Code: CCDVAI

Description: In the modified model version there exists an interface that does not exist in the original version.

OCL Expression: **context** model::foundation::core::ClassDiagramView
self.interface->exists(i1:Interface|not self.model.application.
originalModel.classDiagramView.interface->exists(i2:Interface|
i1.getPathname() = i2.getPathname()))

10. Changed Class Diagram View – Deleted Interface

Change Code: CCDVDI

Description: In the original model version there exists an interface that does not exist in the modified version.

OCL Expression: **context** model::foundation::core::ClassDiagramView
self.model.application.originalModel.classDiagramView.
interface->exists(i1:Interface|not self.
interface->exists(i2:Interface|
i1.getPathname() = i2.getPathname()))

11. Changed Class Diagram View – Added Realization

Change Code: CCDVAR

Description: In the modified model version there exists a realization relationship that does not exist in the original version.

OCL Expression: **context** model::foundation::core::ClassDiagramView
self.realization->exists(r1:Realization|not self.model.
application.originalModel.classDiagramView.
realization->exists(r2:Realization|
r1.getIDStr() = r2.getIDStr()))

12. Changed Class Diagram View – Deleted Realization

Change Code: CCDVDR

Description: In the original model version there exists a realization relationship that does not exist in the modified version.

OCL Expression: `context model::foundation::core::ClassDiagramView
self.model.application.originalModel.classDiagramView.
realization->exists(r1:Realization|not
self.realization->exists(r2:Realization|
r1.getIDStr() = r2.getIDStr()))`

13. Changed Sequence Diagram View – Added Classifier Role

Change Code: CSDVACR

Description: In the modified model version there exists a classifier role that does not exist in the original version.

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

14. Changed Sequence Diagram View – Deleted Classifier Role

Change Code: CSDVDCR

Description: In the original model version there exists a classifier role that does not exist in the modified version.

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

15. Changed Sequence Diagram View – Added Message

Change Code: CSDVAM

Description: In the modified model version there exists a message that does not exist in the original version.

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

16. Changed Sequence Diagram View – Deleted Message

Change Code: CSDVDM

Description: In the original model version there exists a message that does not exist in the modified version.

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

17. Changed Statechart Diagram View – Added Composite State

Change Code: CStDVACS

Description: In the modified model version there exists a composite state that does not exist in the original version.

OCL Expression: `context model::behaviouralElements::stateMachines::
StatechartDiagramView
self.compositeState->exists(cs1:CompositeState|not self.model.
application.originalModel.statechartDiagramView.
compositeState->exists(cs2:CompositeState|
cs1.getName() = cs2.getName()))`

18. Changed Statechart Diagram View – Deleted Composite State

Change Code: CStDVDCS

Description: In the original model version there exists a composite state that does not exist in the modified version.

OCL Expression: `context model::behaviouralElements::stateMachines::
StatechartDiagramView
self.model.application.originalModel.statechartDiagramView.
compositeState->exists(cs1:CompositeState|not self.
compositeState->exists(cs2:CompositeState|
cs1.getName() = cs2.getName()))`

19. Changed Statechart Diagram View – Added Simple State

Change Code: CStDVASS

Description: In the modified model version there exists a simple state that does not exist in the original version.

OCL Expression: `context model::behaviouralElements::stateMachines::
StatechartDiagramView
self.simpleState->exists(ss1:SimpleState|not self.model.
application.originalModel.statechartDiagramView.
simpleState->exists(ss2:SimpleState|
ss1.getName() = ss2.getName()))`

20. Changed Statechart Diagram View – Deleted Simple State

Change Code: CStDVDSS

Description: In the original model version there exists a simple state that does not exist in the modified version.

OCL Expression: `context model::behaviouralElements::stateMachines::
StatechartDiagramView
self.model.application.originalModel.statechartDiagramView.
simpleState->exists(ss1:SimpleState|not self.
simpleState->exists(ss2:SimpleState|
ss1.getName() = ss2.getName()))`

21. Changed Statechart Diagram View – Added Transition

Change Code: CStDVAT

Description: In the modified model version there exists a transition that does not exist in the original version.

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

22. Changed Statechart Diagram View – Deleted Transition

Change Code: CStDVDT

Description: In the original model version there exists a transition that does not exist in the modified version.

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

23. Changed Association End – Changed Aggregation

Change Code: CAECA

Description: There exists an association end in the model such that its aggregation property is not the same in the two model versions.

OCL Expression: **context** model::foundation::core::AssociationEnd
self.aggregation <> self.association.view.model.application.
originalModel.classDiagramView.getAssociation(self.association.
getIDStr()).getEnd(self.getIDStr()).aggregation

24. Changed Association End – Changed Changeability

Change Code: CAECC

Description: There exists an association end in the model such that its changeability property is not the same in the two model versions.

OCL Expression: **context** model::foundation::core::AssociationEnd
self.changeability <> self.association.view.model.application.
originalModel.classDiagramView.getAssociation(self.association.
getIDStr()).getEnd(self.getIDStr()).changeability

25. Changed Association End – Added Interface Specifier

Change Code: CAE AIS

Description: There exists an association end in the model such that in the modified version it has an interfaceSpecifier that it doesn't have in the original version.

OCL Expression: **context** model::foundation::core::AssociationEnd
self.interfaceSpecifier->exists(s1:Classifier|
not self.association.view.model.application.
originalModel.classDiagramView.
getAssociation(self.association.getIDStr()).
getEnd(self.getIDStr()).
interfaceSpecifier->exists(s2:Classifier|
s1.getPathname() = s2.getPathname()))

26. Changed Association End – Changed Interface Specifier

Change Code: CAECIS

Description: There exists an association end in the model such that it has an interfaceSpecifier (interface or class) that is not the same in the two 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::AssociationEnd
self.interfaceSpecifier->exists(s1:Classifier|
self.association.view.model.application.changeDetector.
change.changedElement->exists(s2:Classifier|
s1.getPathname() = s2.getPathname()))

27. Changed Association End – Deleted Interface Specifier

Change Code: CAEDIS

Description: There exists an association end in the model such that in the original version it has an `interfaceSpecifier` that it doesn't have in the modified version.

OCL Expression:

```
context model::foundation::core::AssociationEnd
self.association.view.model.application.originalModel.
classDiagramView.getAssociation(self.association.
getIDStr()).getEnd(self.getIDStr()).
interfaceSpecifier->exists(s1:Classifier|not
self.interfaceSpecifier->exist(s2:Classifier|
s1.getPathname() = s2.getPathname()))
```

28. Changed Association End – Changed isNavigable

Change Code: CAECiN

Description: There exists an association end in the model such that its `isNavigable` property is not the same in the two model versions.

OCL Expression:

```
context model::foundation::core::AssociationEnd
self.isNavigable <> self.association.view.model.application.
originalModel.classDiagramView.getAssociation(self.association.
getIDStr()).getEnd(self.getIDStr()).isNavigable
```

29. Changed Association End – Changed Multiplicity

Change Code: CAECM

Description: There exists an association end in the model such that its `multiplicity` is not the same in the two model versions.

OCL Expression:

```
context model::foundation::core::AssociationEnd
not self.multiplicity.equals(self.association.view.model.
application.originalModel.classDiagramView.getAssociation(self.
association.getIDStr()).getEnd(self.getIDStr()).multiplicity)
```

30. Changed Association End – Changed Ordering

Change Code: CAECO

Description: There exists an association end in the model such that its `ordering` property is not the same in the two model versions.

OCL Expression:

```
context model::foundation::core::AssociationEnd
self.ordering <> self.association.view.model.application.
originalModel.classDiagramView.getAssociation(self.association.
getIDStr()).getEnd(self.getIDStr()).ordering
```

31. Changed Association End – Added Qualifier

Change Code: CAEAQ

Description: There exists an association end in the model such that in the modified version it has a `qualifier` that it doesn't have in the original version.

OCL Expression:

```
context model::foundation::core::AssociationEnd
self.qualifier->exists(q1:Attribute|not self.association.view.
model.application.originalModel.classDiagramView.
getAssociation(self.association.getIDStr()).getEnd(self.
getIDStr()).qualifier->exists(q2:Attribute|
q1.getName() = q2.getName()))
```

32. Changed Association End – Changed Qualifier Type

Change Code: CAECQT

Description: There exists an association end in the model such that the `type` property of one of its qualifiers is not the same in the two model versions.

OCL Expression:

```
context model::foundation::core::AssociationEnd
self.qualifier->exists(q:Attribute|not q.type.equals(self.
association.view.model.application.originalModel.
classDiagramView.getAssociation(self.association.getIDStr()).
getEnd(self.getIDStr()).qualifier->asSequence->at(self.
getQualifierPosition(q)).type))
```


33. Changed Association End – Deleted Qualifier

Change Code: CAEDQ

Description: There exists an association end in the model such that in the original version it has a qualifier that it doesn't have in the modified version.

OCL Expression: `context model::foundation::core:AssociationEnd
self.association.view.model.application.originalModel.
classDiagramView.getAssociation(self.association.getIDStr()).
getEnd(self.getIDStr()).qualifier->exists(q1:Attribute|not
self.qualifier->exists(q2:Attribute|
q1.getName() = q2.getName()))`

34. Changed Association End – Changed Target Scope

Change Code: CAECTS

Description: There exists an association end in the model such that its targetScope is not the same in the two model versions.

OCL Expression: `context model::foundation::core:AssociationEnd
self.targetScope <> self.association.view.model.application.
originalModel.classDiagramView.getAssociation(self.association.
getIDStr()).getEnd(self.getIDStr()).targetScope`

35. Changed Association End – Changed Visibility

Change Code: CAECV

Description: There exists an association end in the model such that its visibility is not the same in the two model versions.

OCL Expression: `context model::foundation::core:AssociationEnd
self.visibility <> self.association.view.model.application.
originalModel.classDiagramView.getAssociation(self.association.
getIDStr()).getEnd(self.getIDStr()).visibility`

36. Changed Class – Added Attribute

Change Code: CCAA

Description: There exists a class in the model such that in the modified version it has an attribute that it doesn't have in the original version.

OCL Expression: `context model::foundation::core::ClassClassifier
self.getAttributes()->exists(a1:Attribute|not self.view.model.
application.originalModel.classDiagramView.
getClassClassifier(self.getPathname()).
getAttributes()->exists(a2:Attribute|
a1.getIDStr() = a2.getIDStr()))`

37. Changed Class – Deleted Attribute

Change Code: CCDA

Description: There exists a class in the model such that in the original version it has an attribute that it doesn't have in the modified version.

OCL Expression: `context model::foundation::core::ClassClassifier
self.view.model.application.originalModel.classDiagramView.
getClassClassifier(self.getPathname()).
getAttributes()->exists(a1:Attribute|not self.
getAttributes()->exists(a2:Attribute|
a1.getIDStr() = a2.getIDStr()))`

38. Changed Class – Changed Invariant

Change Code: CCCI

Description: There exists a class in the model such that its invariant is not the same in the two model versions.

OCL Expression: `context model::foundation::core::ClassClassifier
not self.invariant.equals(self.view.model.application.
originalModel.classDiagramView.getClassClassifier(self.
getPathname()).invariant)`

39. Changed Class – Changed isAbstract

Change Code: CCCiAbs

Description: There exists a class in the model such that its isAbstract property is not the same in the two model versions.

OCL Expression: `context model::foundation::core::ClassClassifier
self.isAbstract <> self.view.model.application.originalModel.
classDiagramView.getClassClassifier(self.getPathname()).
isAbstract`

40. Changed Class – Changed isActive

Change Code: CCCiA

Description: There exists a class in the model such that its isActive property is not the same in the two model versions.

OCL Expression: `context model::foundation::core::ClassClassifier
self.isActive <> self.view.model.application.originalModel.
classDiagramView.getClassClassifier(self.getPathname()).
isActive`

41. Changed Class – Changed isLeaf

Change Code: CCCiL

Description: There exists a class in the model such that its isLeaf property is not the same in the two model versions.

OCL Expression: `context model::foundation::core::ClassClassifier
self.isLeaf <> self.view.model.application.originalModel.
classDiagramView.getClassClassifier(self.getPathname()).isLeaf`

42. Changed Class – Changed isRoot

Change Code: CCCiR

Description: There exists a class in the model such that its isRoot property is not the same in the two model versions.

OCL Expression: `context model::foundation::core::ClassClassifier
self.isRoot <> self.view.model.application.originalModel.
classDiagramView.getClassClassifier(self.getPathname()).isRoot`

43. Changed Class – Changed Multiplicity

Change Code: CCCM

Description: There exists a class in the model such that its multiplicity property is not the same in the two model versions.

OCL Expression: `context model::foundation::core::ClassClassifier
not self.multiplicity.equals(self.view.model.application.
originalModel.classDiagramView.getClassClassifier(self.
getPathname()).multiplicity)`

44. Changed Class – Added Operation

Change Code: CCAO

Description: There exists a class in the model such that in the modified version it has an operation that it doesn't have in the original version.

OCL Expression: `context model::foundation::core::ClassClassifier
self.getOperations()->exists(o1:Operation|not self.view.model.
application.originalModel.classDiagramView.
getClassClassifier(self.getPathname()).
getOperations()->exists(o2:Operation|
o1.getSignature() = o2.getSignature()))`

45. Changed Class – Deleted Operation

Change Code: CCDO

Description: There exists a class in the model such that in the original version it has an operation that it doesn't have in the modified version.

OCL Expression: `context model::foundation::core::ClassClassifier
self.view.model.application.originalModel.classDiagramView.
getClassClassifier(self.getPathname()).
getOperations()->exists(o1:Operation|not self.
getOperations()->exists(o2:Operation|
o1.getSignature() = o2.getSignature()))`

46. Changed Class – Changed Visibility

Change Code: CCCV

Description: There exists a class in the model such that its visibility property is not the same in the two model versions.

OCL Expression: `context model::foundation::core::ClassClassifier
self.visibility <> self.view.model.application.originalModel.
classDiagramView.getClassClassifier(self.getPathname()).
visibility`

47. Changed Interface – Added Operation

Change Code: CIAO

Description: There exists an interface in the model such that in the modified version it has an operation that it doesn't have in the original version.

OCL Expression: `context model::foundation::core::Interface
self.getOperations()->exists(o1:Operation|not self.view.model.
application.originalModel.classDiagramView.getInterface(self.
getPathname()).getOperations()->exists(o2:Operation|
o1.getSignature() = o2.getSignature()))`

48. Changed Interface – Deleted Operation

Change Code: CIDO

Description: There exists an interface in the model such that in the original version it has an operation that it doesn't have in the modified version.

OCL Expression: `context model::foundation::core::Interface
self.view.model.application.originalModel.classDiagramView.
getInterface(self.getPathname()).
getOperations()->exists(o1:Operation|not self.
getOperations()->exists(o2:Operation|
o1.getSignature() = o2.getSignature()))`

49. Changed Classifier Role – Added Available Operation

Change Code: CCRAAO

Description: There exists a classifier role in the model such that in the modified version it has an available operation that it doesn't have in the original version.

OCL Expression: `context model::behaviouralElements::collaborations::ClassifierRole
self.availableOperation->exists(ao1:Operation|not
self.view.model.application.originalModel.sequenceDiagramView.
getClassifierRole(self.getIDStr()).
availableOperation->exists(ao2:Operation|
ao1.getSignature() = ao2.getSignature()))`

50. Changed Classifier Role – Deleted Available Operation

Change Code: CCRDAO

Description: There exists a classifier role in the model such that in the original version it has an available operation that it doesn't have in the modified version.

OCL Expression: `context model::behaviouralElements::collaborations::ClassifierRole
self.view.model.application.originalModel.sequenceDiagramView.
getClassifierRole(self.getIDStr()).
availableOperation->exists(ao1:Operation|not self.
availableOperation->exists(ao2:Operation|
ao1.getSignature() = ao2.getSignature()))`

51. Changed Classifier Role – Changed Base Class Classifier

Change Code: CCRCBCC

Description: There exists a classifier role in the model such that its base class classifier is not the same in the two model versions. Note that the implementation of this rule assumes that all the changed class classifiers in the model have been previously identified.

OCL Expression: `context model::behaviouralElements::collaborations::ClassifierRole
self.base->exists(b1:ClassClassifier|self.view.model.
application.changeDetector.change.
changedElement->exists(b2:ClassClassifier|
b1.getPathname() = b2.getPathname()))`

52. Changed Classifier Role – Changed Multiplicity

Change Code: CCRCM

Description: There exists a classifier role in the model such that its multiplicity property is not the same in the two model versions.

OCL Expression: `context model::behaviouralElements::collaborations::ClassifierRole
not self.multiplicity.equals(self.view.model.application.
originalModel.sequenceDiagramView.getClassifierRole(self.
getIDStr()).multiplicity)`

The following definition is used in rule 53 below:

```
context model::behaviouralElements::collaborations::Message def:  
  let originalMessageAction:Action = self.view.model.application.originalModel.  
    sequenceDiagramView.getMessage(self.getIDStr()).action
```

53. Changed Message Action – Changed Recurrence

Change Code: CMACR

Description: There exists a message in the model such that the recurrence property of its action is not the same in the two model versions.

OCL Expression: `context model::behaviouralElements::collaborations::Message
not self.action.oclIsTypeOf(ActionSequence) and
not originalMessageAction.oclIsTypeOf(ActionSequence) and
not self.action.recurrence.equals(originalMessageAction.
recurrence)`

54. Changed Composite State – Added Subvertex

Change Code: CCSAS

Description: There exists a composite state in the model such that in the modified version it has a subvertex that it doesn't have in the original version.

OCL Expression: `context model::behaviouralElements::stateMachines::CompositeState
self.subvertex->exists(sv1:StateVertex|not self.view.model.
application.originalModel.statechartDiagramView.
getState(self.getPathname()).subvertex->exists(sv2:StateVertex|
sv1.getName() = sv2.getName()))`

55. Changed Composite State – Deleted Subvertex

Change Code: CCSDS

Description: There exists a composite state in the model such that in the original version it has a subvertex that it doesn't have in the modified version.

OCL Expression: `context model::behaviouralElements::stateMachines::CompositeState
self.view.model.application.originalModel.
statechartDiagramView.getState(self.getPathname()).
subvertex->exists(sv1:StateVertex|not self.
subvertex->exists(sv2:StateVertex|
sv1.getName() = sv2.getName()))`

56. Changed Transition – Changed Guard

Change Code: CTCG

Description: There exists a transition in the model such that its guard condition is not the same in the two model versions.

OCL Expression: `context model::behaviouralElements::stateMachines::Transition
not self.guard.equals(self.view.model.originalModel.
statechartDiagramView.getTransition(self.getIDStr()).guard)`

57. Changed Attribute – Changed Changeability

Change Code: CACC

Description: There exists an attribute in the model such that its changeability property is not the same in the two model versions.

OCL Expression: `context model::foundation::core::Attribute
self.changeability <> self.getClassClassifier().view.model.
application.originalModel.classDiagramView.
getClassClassifier(self.getClassClassifier().getPathname()).
getAttribute(self.name).changeability`

58. Changed Attribute – Changed Initial Value

Change Code: CACIV

Description: There exists an attribute in the model such that its initialValue property is not the same in the two model versions.

OCL Expression: `context model::foundation::core::Attribute
not self.initialValue.equals(self.getClassClassifier().view.
model.application.originalModel.classDiagramView.
getClassClassifier(self.getClassClassifier().getPathname()).
getAttribute(self.name).initialValue)`

59. Changed Attribute – Changed Multiplicity

Change Code: CACM

Description: There exists an attribute in the model such that its multiplicity property is not the same in the two model versions.

OCL Expression: `context model::foundation::core::Attribute
not self.multiplicity.equals(self.getClassClassifier().view.
model.application.originalModel.classDiagramView.
getClassClassifier(self.getClassClassifier().getPathname()).
getAttribute(self.name).multiplicity)`

60. Changed Attribute – Changed Ordering

Change Code: CACO

Description: There exists an attribute in the model such that its ordering property is not the same in the two model versions.

OCL Expression: `context model::foundation::core::Attribute
self.ordering <> self.getClassClassifier().view.model.
application.originalModel.classDiagramView.
getClassClassifier(self.getClassClassifier().getPathname()).
getAttribute(self.name).ordering`

61. Changed Attribute – Changed Owner Scope

Change Code: CACOS

Description: There exists an attribute in the model such that its `ownerScope` is not the same in the two model versions.

OCL Expression: `context model::foundation::core::Attribute
self.ownerScope <> self.getClassClassifier().view.model.
application.originalModel.classDiagramView.
getClassClassifier(self.getClassClassifier().getPathname()).
getAttribute(self.name).ownerScope`

62. Changed Attribute – Changed Target Scope

Change Code: CACTS

Description: There exists an attribute in the model such that its `targetScope` is not the same in the two model versions.

OCL Expression: `context model::foundation::core::Attribute
self.targetScope <> self.getClassClassifier().view.model.
application.originalModel.classDiagramView.
getClassClassifier(self.getClassClassifier().getPathname()).
getAttribute(self.name).targetScope`

63. Changed Attribute – Changed Type

Change Code: CACT

Description: There exists an attribute in the model such that its `type` is not the same in the two model versions.

OCL Expression: `context model::foundation::core::Attribute
self.type.getPathname() <> self.getClassClassifier().view.
model.application.originalModel.classDiagramView.
getClassClassifier(self.getClassClassifier().getPathname()).
getAttribute(self.name).type.getPathname()`

64. Changed Attribute – Changed Visibility

Change Code: CACV

Description: There exists an attribute in the model such that its `visibility` is not the same in the two model versions.

OCL Expression: `context model::foundation::core::Attribute
self.visibility <> self.getClassClassifier().view.model.
application.originalModel.classDiagramView.
getClassClassifier(self.getClassClassifier().getPathname()).
getAttribute(self.name).visibility`

65. Changed Class Operation – Changed Concurrency

Change Code: CCOCC

Description: There exists a class operation in the model such that its `concurrency` property is not the same in the two model versions.

OCL Expression: `context model::foundation::core::Operation
self.concurrency <> self.getClassClassifier().view.model.
application.originalModel.classDiagramView.
getClassClassifier(self.getClassClassifier().getPathname()).
getOperation(self.getSignature()).concurrency`

66. Changed Class Operation – Changed isAbstract

Change Code: CCOCiAbs

Description: There exists a class operation in the model such that its `isAbstract` property is not the same in the two model versions.

OCL Expression: `context model::foundation::core::Operation
self.isAbstract <> self.getClassClassifier().view.model.
application.originalModel.classDiagramView.
getClassClassifier(self.getClassClassifier().getPathname()).
getOperation(self.getSignature()).isAbstract`

67. Changed Class Operation – Changed isPolymorphic

Change Code: CCOCiP

Description: There exists a class operation in the model such that its isPolymorphic property is not the same in the two model versions.

OCL Expression: `context model::foundation::core::Operation
self.isPolymorphic <> self.getClassClassifier().view.model.
application.originalModel.classDiagramView.
getClassClassifier(self.getClassClassifier().getPathname()).
getOperation(self.getSignature()).isPolymorphic`

68. Changed Class Operation – Changed isQuery

Change Code: CCOCiQ

Description: There exists a class operation in the model such that its isQuery property is not the same in the two model versions.

OCL Expression: `context model::foundation::core::Operation
self.isQuery <> self.getClassClassifier().view.model.
application.originalModel.classDiagramView.
getClassClassifier(self.getClassClassifier().getPathname()).
getOperation(self.getSignature()).isQuery`

69. Changed Class Operation – Changed Owner Scope

Change Code: CCOCOS

Description: There exists a class operation in the model such that its ownerScope is not the same in the two model versions.

OCL Expression: `context model::foundation::core::Operation
self.ownerScope <> self.getClassClassifier().view.model.
application.originalModel.classDiagramView.
getClassClassifier(self.getClassClassifier().getPathname()).
getOperation(self.getSignature()).ownerScope`

70. Changed Class Operation – Changed Precondition

Change Code: CCOCPre

Description: There exists a class operation in the model such that its precondition is not the same in the two model versions.

OCL Expression: `context model::foundation::core::Operation
not self.precondition.equals(self.getClassClassifier().view.
model.application.originalModel.classDiagramView.
getClassClassifier(self.getClassClassifier().getPathname()).
getOperation(self.getSignature()).precondition)`

71. Changed Class Operation – Changed Postcondition

Change Code: CCOCpst

Description: There exists a class operation in the model such that its postcondition is not the same in the two model versions.

OCL Expression: `context model::foundation::core::Operation
not self.postcondition.equals(self.getClassClassifier().view.
model.application.originalModel.classDiagramView.
getClassClassifier(self.getClassClassifier().getPathname()).
getOperation(self.getSignature()).postcondition)`

72. Changed Class Operation – Changed Visibility

Change Code: CCOCV

Description: There exists a class operation in the model such that its visibility is not the same in the two model versions.

OCL Expression: `context model::foundation::core::Operation
self.visibility <> self.getClassClassifier().view.model.
application.originalModel.classDiagramView.
getClassClassifier(self.getClassClassifier().getPathname()).
getOperation(self.getSignature()).visibility`

73. Changed Interface Operation – Changed Concurrency

Change Code: CIOCC

Description: There exists an interface operation in the model such that its concurrency is not the same in the two model versions.

OCL Expression: `context model::foundation::core::Operation
self.concurrency <> self.getInterface().view.model.application.
originalModel.classDiagramView.getInterface(self.
getInterface().getPathname()).getOperation(self.
getSignature()).concurrency`

74. Changed Interface Operation – Changed isPolymorphic

Change Code: CIOCiP

Description: There exists an interface operation in the model such that its isPolymorphic property is not the same in the two model versions.

OCL Expression: `context model::foundation::core::Operation
self.isPolymorphic <> self.getInterface().view.model.
application.originalModel.classDiagramView.getInterface(self.
getInterface().getPathname()).getOperation(self.
getSignature()).isPolymorphic`

75. Changed Interface Operation – Changed isQuery

Change Code: CIOCiQ

Description: There exists an interface operation in the model such that its isQuery property is not the same in the two model versions.

OCL Expression: `context model::foundation::core::Operation
self.isQuery <> self.getInterface().view.model.application.
originalModel.classDiagramView.getInterface(self.
getInterface().getPathname()).getOperation(self.
getSignature()).isQuery`

76. Changed Interface Operation – Changed Owner Scope

Change Code: CIOCOS

Description: There exists an interface operation in the model such that its ownerScope is not the same in the two model versions.

OCL Expression: `context model::foundation::core::Operation
self.ownerScope <> self.getInterface().view.model.application.
originalModel.classDiagramView.getInterface(self.
getInterface().getPathname()).getOperation(self.
getSignature()).ownerScope`

77. Changed Interface Operation – Changed Precondition

Change Code: CIOCPre

Description: There exists an interface operation in the model such that its precondition is not the same in the two model versions.

OCL Expression: `context model::foundation::core::Operation
not self.precondition.equals(self.getInterface().view.model.
application.originalModel.classDiagramView.getInterface(self.
getInterface().getPathname()).getOperation(self.
getSignature()).precondition)`

78. Changed Interface Operation – Changed Postcondition

Change Code: CIOCPst

Description: There exists an interface operation in the model such that its postcondition is not the same in the two model versions.

OCL Expression: `context model::foundation::core::Operation
not self.postcondition.equals(self.getInterface().view.model.
application.originalModel.classDiagramView.getInterface(self.
getInterface().getPathname()).getOperation(self.
getSignature()).postcondition)`

79. Changed Class Operation Parameter – Changed Default Value

Change Code: CCOPCDV

Description: There exists a class operation parameter in the model such that its `defaultValue` is not the same in the two model versions.

OCL Expression: `context model::foundation::core::Parameter
not self.defaultValue.equals(self.getOperation().
getClassClassifier().view.model.application.originalModel.
classDiagramView.getClassClassifier(self.getOperation().
getClassClassifier().getPathname()).getOperation(self.
getOperation().getSignature()).
parameter->asSequence->at(self.getOperation().
getParameterPosition(self)).defaultValue)`

80. Changed Class Operation Parameter – Changed Direction

Change Code: CCOPCD

Description: There exists a class operation parameter in the model such that its `direction` is not the same in the two model versions.

OCL Expression: `context model::foundation::core::Parameter
self.direction <> self.getOperation().getClassClassifier().
view.model.application.originalModel.classDiagramView.
getClassClassifier(self.getOperation().getClassClassifier().
getPathname()).getOperation(self.getOperation().
getSignature()).parameter->asSequence->at(self.getOperation().
getParameterPosition(self)).direction`

81. Changed Class Operation Parameter – Changed Name

Change Code: CCOPCN

Description: There exists a class operation parameter in the model such that its `name` is not the same in the two model versions.

OCL Expression: `context model::foundation::core::Parameter
self.name <> self.getOperation().getClassClassifier().view.
model.application.originalModel.classDiagramView.
getClassClassifier(self.getOperation().getClassClassifier().
getPathname()).getOperation(self.getOperation().
getSignature()).parameter->asSequence->at(self.getOperation().
getParameterPosition(self)).name`

82. Changed Interface Operation Parameter – Changed Default Value

Change Code: CIOPCDV

Description: There exists an interface operation parameter in the model such that its `defaultValue` is not the same in the two model versions.

OCL Expression: `context model::foundation::core::Parameter
not self.defaultValue.equals(self.getOperation().
getInterface().view.model.application.originalModel.
classDiagramView.getInterface(self.getOperation().
getInterface().getPathname()).getOperation(self.
getOperation().getSignature()).
parameter->asSequence->at(self.getOperation().
getParameterPosition(self)).defaultValue)`

83. Changed Interface Operation Parameter – Changed Direction

Change Code: CIOPCD

Description: There exists an interface operation parameter in the model such that its `direction` is not the same in the two model versions.

OCL Expression: `context model::foundation::core::Parameter
self.direction <> self.getOperation().getInterface().view.
model.application.originalModel.classDiagramView.
getInterface(self.getOperation().getInterface().
getPathname()).getOperation(self.getOperation().
getSignature()).parameter->asSequence->at(self.getOperation().
getParameterPosition(self)).direction`

84. Changed Interface Operation Parameter – Changed Name

Change Code: CIOPCN

Description: There exists an interface operation parameter in the model such that its name is not the same in the two model versions.

OCL Expression: `context model::foundation::core::Parameter
self.name <> self.getOperation().getInterface().view.
model.application.originalModel.classDiagramView.
getInterface(self.getOperation().getInterface()).
getPathname().getOperation(self.getOperation()).
getSignature().parameter->asSequence->at(self.getOperation().
getParameterPosition(self)).name`

85. Changed State – Added Activity

Change Code: CSAA

Description: There exists a state in the model such that in the modified version it has a `doActivity` property and it doesn't have this property in the original version.

OCL Expression: `context model::behaviouralElements::stateMachines::State
self.doActivity->size = 1 and self.view.model.application.
originalModel.statechartDiagramView.getState(self.
getPathname()).doActivity->size = 0`

86. Changed State – Deleted Activity

Change Code: CSDA

Description: There exists a state in the model such that in the original version it has a `doActivity` property and it doesn't have this property in the modified version.

OCL Expression: `context model::behaviouralElements::stateMachines::State
self.doActivity->size = 0 and self.view.model.application.
originalModel.statechartDiagramView.getState(self.
getPathname()).doActivity->size = 1`

87. Changed State – Added Entry Action

Change Code: CSAEA

Description: There exists a state in the model such that in the modified version it has an `entry` action and it doesn't have this property in the original version.

OCL Expression: `context model::behaviouralElements::stateMachines::State
self.entry->size = 1 and self.view.model.application.
originalModel.statechartDiagramView.getState(self.
getPathname()).entry->size = 0`

88. Changed State – Deleted Entry Action

Change Code: CSDEA

Description: There exists a state in the model such that in the original version it has an `entry` action and it doesn't have this property in the modified version.

OCL Expression: `context model::behaviouralElements::stateMachines::State
self.entry->size = 0 and self.view.model.application.
originalModel.statechartDiagramView.getState(self.
getPathname()).entry->size = 1`

89. Changed State – Added Exit Action

Change Code: CSAExA

Description: There exists a state in the model such that in the modified version it has an `exit` action and it doesn't have this property in the original version.

OCL Expression: `context model::behaviouralElements::stateMachines::State
self.exit->size = 1 and self.view.model.application.
originalModel.statechartDiagramView.getState(self.
getPathname()).exit->size = 0`

90. Changed State – Deleted Exit Action

Change Code: CSDExA

Description: There exists a state in the model such that in the original version it has an `exit` action and it doesn't have this property in the modified version.

OCL Expression: `context model::behaviouralElements::stateMachines::State
self.exit->size = 0 and self.view.model.application.
originalModel.statechartDiagramView.getState(self.
getPathname()).exit->size = 1`

91. Changed State – Added Internal Transition

Change Code: CSAIT

Description: There exists a state in the model such that in the modified version it has an `internalTransition` that it doesn't have in the original version.

OCL Expression: `context model::behaviouralElements::stateMachines::State
self.internalTransition->exists(t1:Transition|not
self.view.model.application.originalModel.
statechartDiagramView.getState(self.getPathname()).
internalTransition->exists(t2:Transition|
t1.getIDStr() = t2.getIDStr()))`

92. Changed State – Deleted Internal Transition

Change Code: CSDIT

Description: There exists a state in the model such that in the original version it has an `internalTransition` that it doesn't have in the modified version.

OCL Expression: `context model::behaviouralElements::stateMachines::State
self.view.model.application.originalModel.
statechartDiagramView.getState(self.getPathname()).
internalTransition->exists(t1:Transition|not
self.internalTransition->exists(t2:Transition|
t1.getIDStr() = t2.getIDStr()))`

93. Changed State – Changed State Invariant

Change Code: CSCSI

Description: There exists a state in the model such that its state invariant is not the same in the two model versions.

OCL Expression: `context model::behaviouralElements::statecharts::State
not self.invariant.equals(self.view.model.application.
originalModel.statechartDiagramView.getState(self.
getPathname()).invariant)`

The following definition is used in rules 94 to 97 inclusive:

```
context model::commonBehaviour::Action def:
  let originalSMAction:Action =
    (if Action.allInstances.transition->size = 1 then
      Action.allInstances.transition.view.model.application.originalModel.
      statechartDiagramView.getTransition(Action.allInstances.transition.
      getIDStr()).effect
    else -- the else clause assumes that Action.allInstances.state->size = 1
      Action.allInstances.state.view.model.application.originalModel.
      statechartDiagramView.getState(Action.allInstances.state.getPathname()).
      action
      -- here the action returned by the expression refers to
      -- the entry, exit and doActivity properties of State.
    endif)
```

94. **Changed State Machine Action – Added Discrete Action**

Change Code: CSMAADA

Description: There exists an action in the model such that in the modified version it has a discrete action that it doesn't have in the original version.

OCL Expression: **context** model::commonBehaviour::Action
 Action.allInstances.oclIsTypeOf(ActionSequence) and
 originalSMAction.oclIsTypeOf(ActionSequence) and
 Action.allInstances.action->exists(a1:Action|not
 originalSMAction.action->exists(a2:Action|
 a1.getIDStr() = a2.getIDStr()))

95. **Changed State Machine Action – Deleted Discrete Action**

Change Code: CSMADDA

Description: There exists an action in the model such that in the original version it has a discrete action that it doesn't have in the modified version.

OCL Expression: **context** model::commonBehaviour::Action
 Action.allInstances.oclIsTypeOf(ActionSequence) and
 originalSMAction.oclIsTypeOf(ActionSequence) and
 originalSMAction.action->exists(a1:Action|not
 Action.allInstances.action->exists(a2:Action|
 a1.getIDStr() = a2.getIDStr()))

96. **Changed State Machine Action – Changed Recurrence**

Change Code: CSMACR

Description: There exists an action in the model such that its recurrence property is not the same in the two model versions.

OCL Expression: **context** model::commonBehaviour::Action
 not Action.allInstances.oclIsTypeOf(ActionSequence) and
 not originalSMAction.oclIsTypeOf(ActionSequence) and
 not Action.allInstances.recurrence.
 equals(originalSMAction.recurrence)

97. **Changed State Machine Action – Changed Script**

Change Code: CSMACS

Description: There exists an action in the model such that its script property is not the same in the two model versions.

OCL Expression: **context** model::commonBehaviour::Action
 not Action.allInstances.oclIsTypeOf(ActionSequence) and
 not originalSMAction.oclIsTypeOf(ActionSequence) and
 not Action.allInstances.script.equals(originalSMAction.script)

Appendix C Consistency Verification

Each consistency rule is briefly described in Natural Language. The rules are being formalized using OCL. Each of the rules is a Boolean expression. There are also some consistency warnings listed below, these indicate consistency conditions that should be checked. These arise because further work is needed to make extract the required information from the model so that they can be consistency rules.

1. No protected operation can be called (in a sequence diagram) by an operation belonging to a class that is not a descendent of the container class
2. No protected operation can be called (in a statechart) by an operation belonging to a class that is not a descendent of the container class
3. No private operation can be called (in a sequence diagram) by an operation belonging to another class
4. No private operation can be called (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 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 that owns the attribute
32. An operation with the “leaf” property may not be overridden
33. Each attribute that is called in a statechart transition must be defined in the corresponding class diagram
34. Each operation that is invoked in a sequence message must be defined in a class diagram
35. Each operation that is invoked in a state transition must be defined in a class diagram
36. 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 superclass is the whole.
37. 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, have the same name and the same rolenames.
38. A class cannot be a part in more than one composition – no composite part may be shared by two composite objects.
39. Each base classifier that appears in a sequence diagram must be defined in the static view of the model.
40. Each operation invoked on a classifier role in a sequence diagram must be defined in the static view of the model.
41. No attribute of a class and a qualifier or rolename of an associated class can have the same name.
42. 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.

43. 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.
44. In a sequence diagram, a class role can only invoke an operation on another role if it has a navigable association to the target role.
45. 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.
46. 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.
47. No two class in the same package can have the same name.
48. No two attributes in a class can have the same name.
49. A sequence message cannot update an attribute if the attribute's changeability is not "changeable".
50. A transition's action list cannot update an attribute if the attribute's changeability is not "changeable".
51. The postcondition of an operation must not *possibly update* an attribute whose changeability is not "changeable".
52. The multiplicity range for an attribute must be adhered to by all elements that access it.
53. A static operation cannot access an instance attribute
54. A static operation cannot invoke an instance operation
55. For every class in which operations use another class there must be a dependency relationship between the two classes in the class diagram.
56. Each concrete class must implement all the abstract operations of its super class(es).
57. Each class that contains at least one abstract operation must be declared abstract.
58. An abstract class cannot be instantiated.
59. A class's multiplicity must not be violated by the multiplicity of any association end in which it is the participant.
60. A class's multiplicity must not be violated by the multiplicity of any classifier role in which it is the base.
61. A class's package visibility should be observed, especially for associations between classes of different packages.
62. A class that realizes an interface must declare all the operations in the interface.
63. A classifier role cannot have an available operation that is not declared in its base class.
64. An operation cannot be invoked on a classifier role that is not present in its set of available operations.
65. An abstract operation cannot be invoked.

66. If an operation is not polymorphic then it cannot be overridden in subclasses of its class.
67. 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.
68. 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.
69. For all the class operations that realize an interface operation, their concurrency values must be the same as that of the interface operation.
70. For all the class operations that realize an interface operation, their polymorphic properties must be the same as that of the interface operation.
71. For all the class operations that realize an interface operation, their query properties must be the same as that of the interface operation.
72. For all the class operations that realize an interface operation, their owner scope values must be the same as that of the interface operation.
73. For all the class operations that realize an interface operation, their precondition must be the same as that of the interface operation.
74. For all the class operations that realize an interface operation, their postcondition must be the same as that of the interface operation.
75. There must not exist two classifier roles in the sequence diagram view such that both roles have the same pathnames.
76. 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 superclass is the whole.
77. 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, have the same name and the same rolenames.
78. A class cannot be a part in more than one composition – no composite part may be shared by two composite objects.
79. Each base classifier that appears in a sequence diagram must be defined in the static view of the model.
80. Each operation invoked on a classifier role in a sequence diagram must be defined in the static view of the model.
81. No attribute of a class and a qualifier or rolename of an associated class can have the same name.
82. 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.
83. 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.

84. In a sequence diagram, a class role can only invoke an operation on another role if it has a navigable association to the target role.
85. 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.
86. 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.
87. No two class in the same package can have the same name.
88. No two attributes in a class can have the same name.
89. A sequence message cannot update an attribute if the attribute's changeability is not "changeable".
90. A transition's action list cannot update an attribute if the attribute's changeability is not "changeable".
91. The postcondition of an operation must not *possibly update* an attribute whose changeability is not "changeable".
92. The multiplicity range for an attribute must be adhered to by all elements that access it.
93. A static operation cannot access an instance attribute
94. A static operation cannot invoke an instance operation
95. For every class in which operations use another class there must be a dependency relationship between the two classes in the class diagram.
96. Each concrete class must implement all the abstract operations of its super class(es).
97. Each class that contains at least one abstract operation must be declared abstract.
98. An abstract class cannot be instantiated.
99. A class's multiplicity must not be violated by the multiplicity of any association end in which it is the participant.
100. A class's multiplicity must not be violated by the multiplicity of any classifier role in which it is the base.
101. A class's package visibility should be observed, especially for associations between classes of different packages.
102. A class that realizes an interface must declare all the operations in the interface.
103. A classifier role cannot have an available operation that is not declared in its base class.
104. An operation cannot be invoked on a classifier role that is not present in its set of available operations.
105. An abstract operation cannot be invoked.
106. If an operation is not polymorphic then it cannot be overridden in subclasses of its class.

107. 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.
108. 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.
109. For all the class operations that realize an interface operation, their concurrency values must be the same as that of the interface operation.
110. For all the class operations that realize an interface operation, their polymorphic properties must be the same as that of the interface operation.
111. For all the class operations that realize an interface operation, their query properties must be the same as that of the interface operation.
112. For all the class operations that realize an interface operation, their owner scope values must be the same as that of the interface operation.
113. For all the class operations that realize an interface operation, their precondition must be the same as that of the interface operation.
114. For all the class operations that realize an interface operation, their postcondition must be the same as that of the interface operation.
115. There must not exist 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 Impact Analysis (Side Effect) Rules

The impact analysis rules described below correspond to the changes defined in the change taxonomy defined in Appendix B. The fields for each rule are described below. The format for the title of each rule is as follows: “Changed *Element* – Added/Changed/Deleted *Property*”. Where *Element* is the changed element’s class, and *Property* is the property of the element that has changed. If the property is an added/deleted link, then the target object’s class is used.

Note that some fields may not appear in a particular rule – this is because these fields do not have a value, thus there is no point in including the field. For example, if no elements were impacted by a particular change then there would be no *Rationale*, no *Resulting Changes* and no *OCL Expressions* since these would not have any value. Note also, that the rules assume that the model is consistent.

Some rules require the `getProperty(propertyID:String)` operation. This operation is defined for each model element class that requires it. The operation returns the element’s property given the property’s ID. A data dictionary will be included later providing detail information on all the operations that appear in the rules.

Change Code:	This is the code that is assigned to the rule and is an acronym for the change type.
Changed Element:	The pathname of the class (in the meta-model) that has been changed. Note that is field shows the type for changedElement type (in the OCL expressions below) is of type
Added Property:	The rolename of the association end containing the class (in the meta-model) for the added link’s target object; or the pathname of the class (in the meta-model) for the added link’s target object. Note that a property is an attribute or a link to a changed element.
Changed Property:	The changed attribute of the changed element.
Deleted Property:	The rolename of the association end containing the class (in the meta-model) for the added link’s target object; or the pathname of the class (in the meta-model) for the deleted link’s target object.

Impacted Element:	Specifies the pathname of the impacted element's class (in the meta-model).
Description:	States, in natural language, what elements have been impacted by the change and under what conditions.
Rationale:	Explains the reason(s) for the impacts.
Resulting Change:	States the changes that may be needed to accomplish a change. These changes are the changes for which no impact analysis rules are defined, such as changes to the implementation, for example.
Invoked Rules:	The rules to be invoked by the current rule. That is, a rule may result in a set of changes – there may be impact analysis rules defined for these changes. Each of these invoked rules may subsequently invoke other rules. Therefore, this field provides information about the <i>transitive closure</i> for a particular change.
OCL Expression:	OCL expression stating how the impacted elements are determined. The OCL expression represents only the logic of how to determine the impacted elements, and thus the implementation may be different.

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

```
def:
    let null:ModelElement = Set{}
```

1. Changed Class Diagram View – 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 bag of impacted classes is such that each impacted class in the bag is a participant (in the meta-model) of one of the added association's ends, and each impacted class can navigate to the classifier at the other end of the association via the added association.

Rationale: Each of the impacted classes 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. Also, 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).getProperty(self.propertyID).
        oclAsType(Association)

    addedAssociation.end->select(e:AssociationEnd|
        addedAssociation.getOtherEnd(e.getIDStr()).
        isNavigable = true).participant
```

2. Changed Class Diagram View – Deleted Association

Change Code: CCDVDA

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

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

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

Description: The bag of impacted classes is such that each impacted class in the bag was a participant (in the meta-model) of one of the deleted association's ends, and each impacted class was able to navigate to the classifier at the other end of the association via the deleted association.

Rationale: Each of the impacted classes has lost a navigable association.

Resulting Change: Methods (implementations of the operations) may have to be updated and/or deleted to reflect the change. Also, 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).getProperty(self.propertyID).
        oclAsType(Association)

    deletedAssociation.end->select(e:AssociationEnd|
        addedAssociation.getOtherEnd(e.getIDStr()).
        isNavigable = true).participant->select(p:ClassClassifier|
        p.view.model.application.modifiedModel.classDiagramView.
        classClassifier->exists(c:ClassClassifier|
        c.getPathname() = p.getPathname()))
```

3. Changed Class Diagram View – Added Class

Change Code: CCDVAC
Changed Element: model::foundation::core::ClassDiagramView
Added Property: model::foundation::core::ClassClassifier
Description: No impact since the model is assumed to be consistent.

4. Changed Class Diagram View – Deleted Class

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: The bag of impacted classes is such that each impacted class in the bag had a navigable association to, or dependency relationship (as a client) with the deleted class. The descendants of the deleted class are also impacted. The interfaces that have a dependency relationship (as a client) with the deleted class are also impacted.
Rationale: Each of the impacted classes 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. Also, since the model is assumed to be consistent, there should be no parameter nor attribute that has the deleted class as its type.

OCL Expressions:

```
context modelChanges::Change def:
  let deletedClass:ClassClassifier = self.changedElement.
    oclAsType(ClassDiagramView).getProperty(self.propertyID).
    oclAsType(ClassClassifier)

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 Class Diagram View – 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) .
getProperty(self.propertyID).oclAsType(Dependency).client
```

6. Changed Class Diagram View – 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: Same as that of CCDVAD.

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 Class Diagram View – 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 subclassifier 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.oclAsType(ClassDiagramView) .
getProperty(self.propertyID).oclAsType(Generalization).child
```

8. Changed Class Diagram View – 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 subclassifier of its former superclassifier.

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 Class Diagram View – Added Interface

Change Code: CCDVAI
Changed Element: model::foundation::core::ClassDiagramView
Added Property: model::foundation::core::Interface
Description: No impact since the model is assumed to be consistent.

10. Changed Class Diagram View – Deleted Interface

Change Code: CCDVDI
Changed Element: model::foundation::core::ClassDiagramView
Deleted Property: model::foundation::core::Interface
Impacted Elements: model::foundation::core::ClassClassifier
model::foundation::core::Interface
Description: The bag of impacted classes is such that each impacted class in the bag had a navigable association to, or dependency relationship (as a client) with the deleted interface. 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.
OCLE Expressions:

```
context modelChanges::Change def:
  let deletedInterface:Interface = self.changedElement.
    oclAsType(ClassDiagramView).oclAsType(ClassDiagramView).
    getProperty(self.propertyID).oclAsType(Interface)

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 -- subinterfaces
  deletedInterface.specialization.child
```


11. Changed Class Diagram View – 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:

```
context modelChanges::Change
self.changedElement.oclassType(ClassDiagramView).
getProperty(self.propertyID).oclassType(Realization).
implementation
```

12. Changed Class Diagram View – Deleted Realization

Change Code: CCDVDR
Changed Element: `model::foundation::core::ClassDiagramView`
Deleted Property: `model::foundation::core::Realization`
Impacted Element: `model::foundation::core::ClassClassifier`
Description: Same as that of CCDVAR.
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 Sequence Diagram View – Added Classifier Role

Change Code: CSDVACR
Changed Element: `model::behaviouralElements::collaborations::SequenceDiagramView`
Added Property: `model::behaviouralElements::collaborations::ClassifierRole`
Description: No impact since the model is assumed to be consistent.

14. Changed Sequence Diagram View – Deleted Classifier Role

Change Code: CSDVDCR

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

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

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

Description: The bag of impacted classes is such that each impacted class in the bag is a base class of at least one of the classifier roles in the sequence diagram view of the model, and each impacted class sent at least one message, to the deleted classifier role, that did not invoke an operation (on the deleted classifier role). The bag of impacted operations is such that each impacted operation sent at least one message to the deleted classifier role.

Rationale: The impacted classes no longer send messages to the deleted classifier role. The impacted operations no longer send messages to the deleted classifier role.

Resulting Changes: The implementation of the impacted classes as well as that of the impacted operations may have to be changed.

OCL Expressions:

```
context modelChanges::Change def:
  let deletedRole:ClassifierRole = self.changedElement.
    oclAsType(SequenceDiagramView).getProperty(self.
      propertyID).oclAsType(ClassifierRole)

  let senderRoles:ClassifierRole = deletedRole.receivedMessage.
    sender

context modelChanges::Change -- classes
  senderRoles->select(sr:ClassifierRole|sr.
    sentMessage->select(sm:Message|deletedRole.
      receivedMessage->includes(sm))->exists(m:Message|
        not m.activator.action.oclIsTypeOf(CallAction))).base

context modelChanges::Change -- operations
  senderRoles.base.operation->select(od:Operation|
    deletedRole.receivedMessage.activator.action.
    operation->exists(oi:Operation|od.equals(oi)))
  -- od = defined operation
  -- oi = invoked operation
```

15. Changed Sequence Diagram View – 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 if the message that sends the added message does not invoke an operation, i.e. its action is not a *call action*; else, the operation, *o*, that sends the added message is impacted. The postcondition of *o* is also impacted.

Rationale: The impacted class now performs one more action. The impacted operation also now performs one more action. 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 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).getProperty(self.
            propertyID).oclAsType(Message)

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

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

context modelChanges::Change -- operation
    sendingOperation

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

16. Changed Sequence Diagram View – 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 if the message that sent the deleted message does not invoke an operation; else, the operation, *o*, that sent the deleted message is impacted. The postcondition of *o* is also impacted.

Rationale: The impacted class now performs one less action. The impacted operation also now performs one less action. The impacted postcondition may be invalidated since one action has been deleted from the sequence of actions performed by the postcondition's 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 State Diagram View – Added Composite State

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) .
getProperty(self.propertyID).oclAsType(CompositeState) .
stateMachine.context
```

18. Changed State Diagram View – Deleted Composite State

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 State Diagram View – Added Simple State

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.oclAsType(StatechartDiagramView).
    getProperty(self.propertyID).oclAsType(SimpleState).
    stateMachine.context
```

20. Changed State Diagram View – Deleted Simple State

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 State Diagram View – 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.oclAsType(StatechartDiagramView).
    getProperty(self.propertyID).oclAsType(Transition).
    stateMachine.context
```

22. Changed State Diagram View – 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 Association End – 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 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 life time 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 life time dependencies of the parts in the composition relationship, for example, constructors and destructors may have to be changed.

OCL Expression:

```
context modelChanges::Change
if (self.changedElement.oclasType(AssociationEnd).
    getAggregation() = #composite) or (self.changedElement.
    association.view.model.application.originalModel.
    classDiagramView.getAssociation(self.changedElement.
    association.getIDStr()).getEnd(self.changedElement.
    getIDStr()).getAggregation() = #composite) then
    self.changedElement.participant
else
    null
endif)
```

24. Changed Association End – 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 verify 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. Also, 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.
        association.getOtherEnd(self.changedElement.getIDStr()).
        participant

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

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

25. Changed Association End – Added Interface Specifier

Change Code: CAEAIIS

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

Added Property: `interfaceSpecifier`

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

26. Changed Association End – Changed Interface Specifier

Change Code: CAECIS

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

Changed Property: `interfaceSpecifier`

Description: “Changed Interface Specifier” 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 Association End – Deleted Interface Specifier

Change Code: CAEDIS

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

Deleted Property: `interfaceSpecifier`

Description: 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. Thus, deleting the interface specifier does not result in any impacts.

28. Changed Association End – 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.isNavigable = true then
    self.changedElement.association.getOtherEnd(self.
      changedElement.getIDStr()).participant
  else
    null
  endif
```


29. Changed Association End – 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 Association End – 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. Also, 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 Association End – 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 Association End – 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 CAECO.
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 CAECO.
OCL Expressions: -- Same as that of CAECO.

33. Changed Association End – 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 CAECO.
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 CAECO.
OCL Expressions: -- Same as that of CAECO.

34. Changed Association End – Changed Target Scope

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 CAECO.
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 CAECO.
OCL Expressions: -- Same as that of CAECO.

35. Changed Association End – Changed Visibility

Change Code: CAECV
Changed Element: `model::foundation::core::AssociationEnd`
Changed Property: `visibility`
Description: No impact since the model is assumed to be consistent.

36. Changed Class – 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 Class – Deleted Attribute

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

38. Changed Class – Changed Invariant

Change Code: CCCI
Changed Element: `model::foundation::core::ClassClassifier`
Changed Property: `invariant`
Description: No impact since the model is assumed to be consistent.

39. Changed Class – 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.oclIsTypeOf(ClassClassifier)))
```

40. Changed Class – Changed isActive

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

41. Changed Class – Changed isLeaf

Change Code: CCCiL

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

Changed Property: `isLeaf`

Description: 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 Class – Changed isRoot

Change Code: CCCiR

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

Changed Property: `isRoot`

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

43. Changed Class – Changed Multiplicity

Change Code: CCCM

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

Changed Property: `multiplicity`

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

44. Changed Class – Added Operation

Change Code: CCAO

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

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

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

45. Changed Class – Deleted Operation

Change Code: CCDO

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

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

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

46. Changed Class – Changed Visibility

Change Code: CCCV

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

Changed Property: `visibility`

Description: 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: 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 realizes 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:	<pre>context modelChanges::Change self.changedElement.implementationRealization. implementation->select(c:ClassClassifier c.operation->includes(self.changedElement. oclAsType(Interface).getOperation(self.propertyID). oclAsType(Operation)))</pre>

49. Changed Classifier Role – Added Available Operation

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

50. Changed Classifier Role – Deleted Available Operation

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

51. Changed Classifier Role – Changed Base Class Classifier

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 Classifier Role – 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 if the message that sends the changed message does not invoke an operation; else, the operation, *o*, that sends the changed message is impacted. The postcondition of *o* 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

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

context modelChanges::Change -- class
    if not changedMessage.activator.oclIsTypeOf(CallAction) then
        changedMessage.sender.base
    else
        null
    endif

context modelChanges::Change -- operation
    sendingOperation

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

54. Changed Composite State – 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 Composite State – 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:	<pre>context modelChanges::Change Def: let affectedClass:ClassClassifier = self.changedElement. getClassClassifier() context modelChanges::Change -- class affectedClass context modelChanges::Change -- operations if self.changedElement.oclAsType(Attribute).getProperty(self. propertyID).oclAsType(ChangeableKind) <> #changeable then affectedClass.getOperations()->select(o:Operation o.postcondition.possiblyUpdatesVariable(self. changedElement.name)) else null endif</pre>

58. Changed Attribute – Changed Initial Value

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 Owner Scope

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. Also, 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 Target Scope

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: No impact since the model is assumed to be consistent.

65. Changed Class Operation – Changed Concurrency

Change Code: CCOCC
Changed Element: `model::foundation::core::Operation`
Changed Property: `concurrency`
Description: No impact. The method of the changed operation may have to be modified since concurrent access to it has changed.

66. Changed Class Operation – Changed isAbstract

Change Code: CCOCiAbs
Changed Element: `model::foundation::core::Operation`
Changed Property: `isAbstract`
Description: 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 Class Operation – Changed isPolymorphic

Change Code: CCOCiP
Changed Element: `model::foundation::core::Operation`
Changed Property: `isPolymorphic`
Description: 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 Class Operation – Changed isQuery

Change Code: CCOCiQ
Changed Element: `model::foundation::core::Operation`
Changed Property: `isQuery`
Description: 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 Class Operation – Changed Owner Scope**
Change Code: CCOCOS
Changed Element: `model::foundation::core::Operation`
Changed Property: `ownerScope`
Description: 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 Class 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 Class 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 methods of the impacted operations may now have different effects. The impacted postconditions may have to be changed.
Resulting Change: The methods of the impacted operations have to be checked and possibly changed.
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 Class Operation – Changed Visibility**
Change Code: CCOCV
Changed Element: `model::foundation::core::Operation`
Changed Property: `visibility`
Description: 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: 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: 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: No impact since the model is assumed to be consistent.

76. Changed Interface Operation – Changed Owner Scope

Change Code: CIOCOS

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

Changed Property: `ownerScope`

Description: 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: 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: 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 Default Value

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 Default Value

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: 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 Activity

Change Code: CSAA
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 Activity

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

87. Changed State – Added Entry Action

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

88. Changed State – Deleted Entry Action

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

89. Changed State – Added Exit Action

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

90. Changed State – Deleted Exit Action

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

91. Changed State – Added Internal Transition

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

92. Changed State – Deleted Internal Transition

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

93. Changed State – Changed State Invariant

Change Code: CSCSI
Changed Element: `model::behaviouralElements::stateMachines::State`
Changed Property: `invariant`
Impacted Element: `model::foundation::core::ClassClassifier`
Description: Same as that of CSAA.
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 CSAA.

94. Changed State Machine 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 State Machine 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 State Machine 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 State Machine 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 E Case Study

E.1 Logical Changes

Eight (8) logical changes were made in the case study. This translated into 70 changes.

The logical The logical changes are described below.

Change # 1

We want to be able to keep track of how many times per session a user attempts to enter the PIN – after 3 invalid PIN's the card will be 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)1.1.2: resetNumTries() (CardInsert)
 - b. (CSDVAM)3: 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 then 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 ‘4: 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)PrintingReceipt – added myState: StateEnum
 23. (CSDVAM)PrintingRecipet – 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)CardNoReadable – added message ‘3: setStateWaitingForCard()’

30. (CSDVACR)FailedTransaction – added myState:StateEnum
31. (CSDVAM)FailedTransaction – added message ‘ 4: [err=2 or err=3]setToGettingTransAmountState()’
32. (CSDVCR)PINInvalid – added myState:StateEnum
33. (CSDVAM)PINInvliad – added message ‘ 4: [err = 1 and try < 3] setToGettingPINState()’
34. (CSDVAM)PINInvalid – added message ‘ 6:[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 and a customer can only belong to one bank

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

Change # 5

Class Account is changed to an Abstract class – Rationale: you will never have an instance of class Account since accounts are always either Savings or Chequeings 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

Change #8

Error corrections, as follows:

- Changed invariant of the Savings class
- Changed initial value fo the transAmount attribute in the ATM class
- Corrected a syntax error in the postcondition of the setPIN operation in the Transaction class
- Changed the message condition for message 1.1.2 in the Inquiry interaction

E.2 Change Distribution

Table 1 below presents the change distribution for the ATM case study. Only 16 of the 97 (leaf) changes in the change taxonomy was used in this case study.

Change Code	Number of Changes
CCDVAA	1
CCDVAC	1
CCDVAI	1
CCDVAR	1
CSDVACR	2
CSDVAM	35
CSDVDM	12
CAECM	2
CCAA	2
CCDA	1
CCCiAbs	1
CCAO	4
CCRCBCC	4
CMACR	1
CACIV	1
CCOCPst	1

Table 1: Change Distribution for ATM Case Study

E.3 Impacts Vs Distance Graphs

Figure E1 below presents the cumulative number of all impacted elements for the ATM case study while Figure E2 below presents the cumulative number of classes impacted for the same case study.

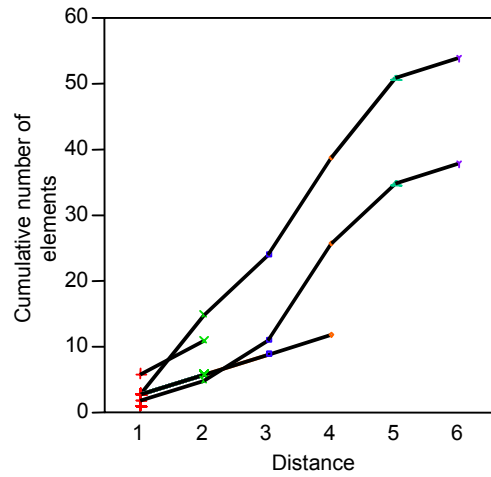


Figure E1: Cumulative number of all impacted elements vs. distance.

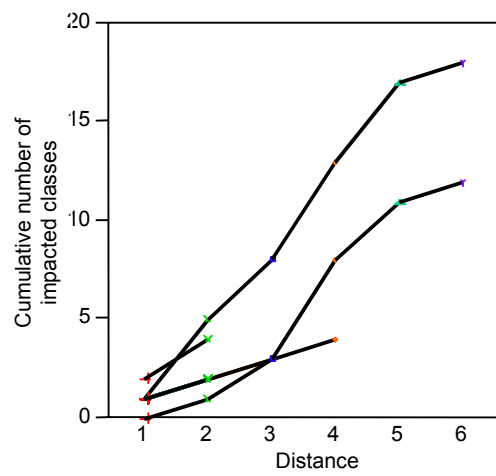


Figure E2: Cumulative number of impacted classes vs. distance

E.4 UML Model (Original)

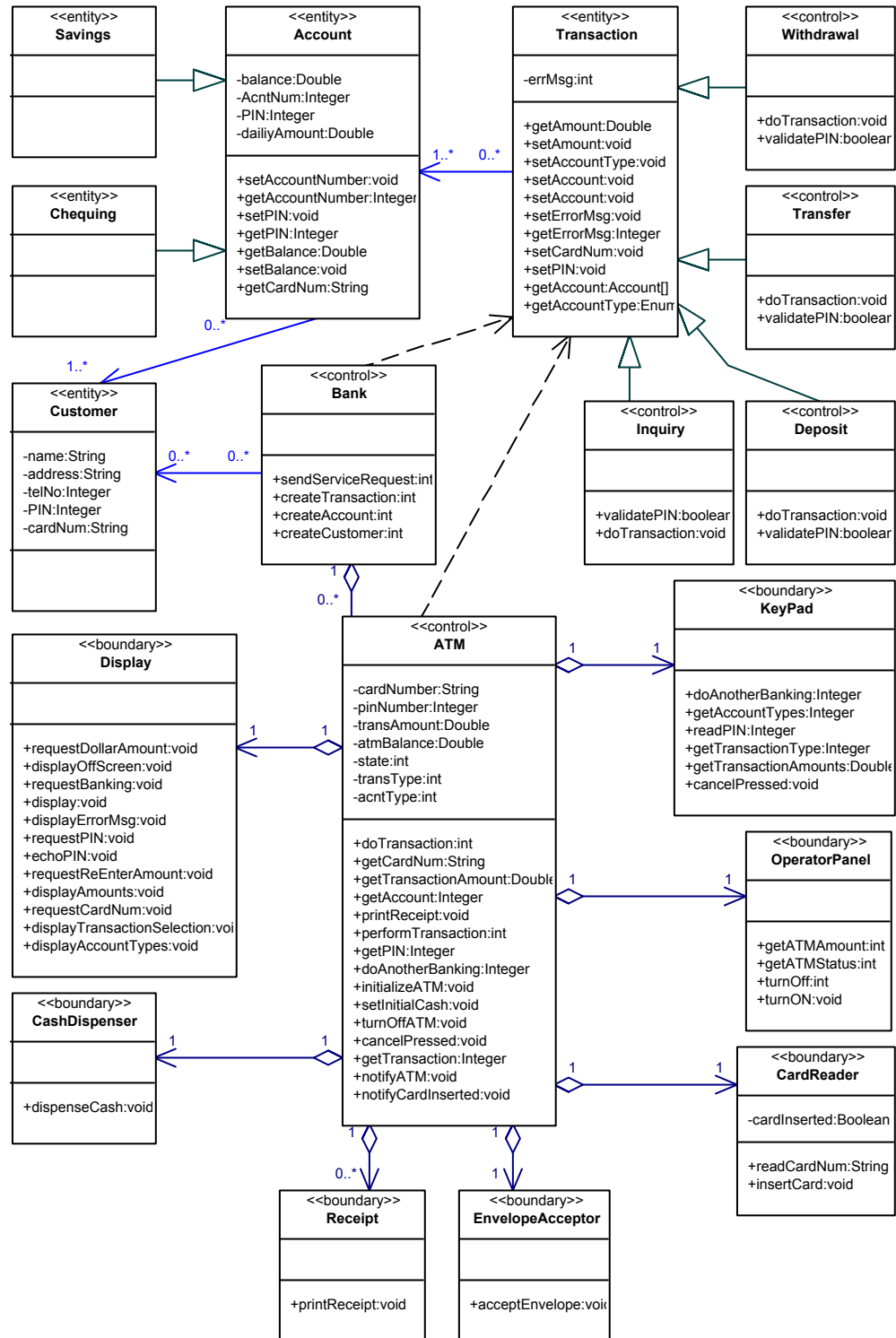


Figure E3: Class Diagram

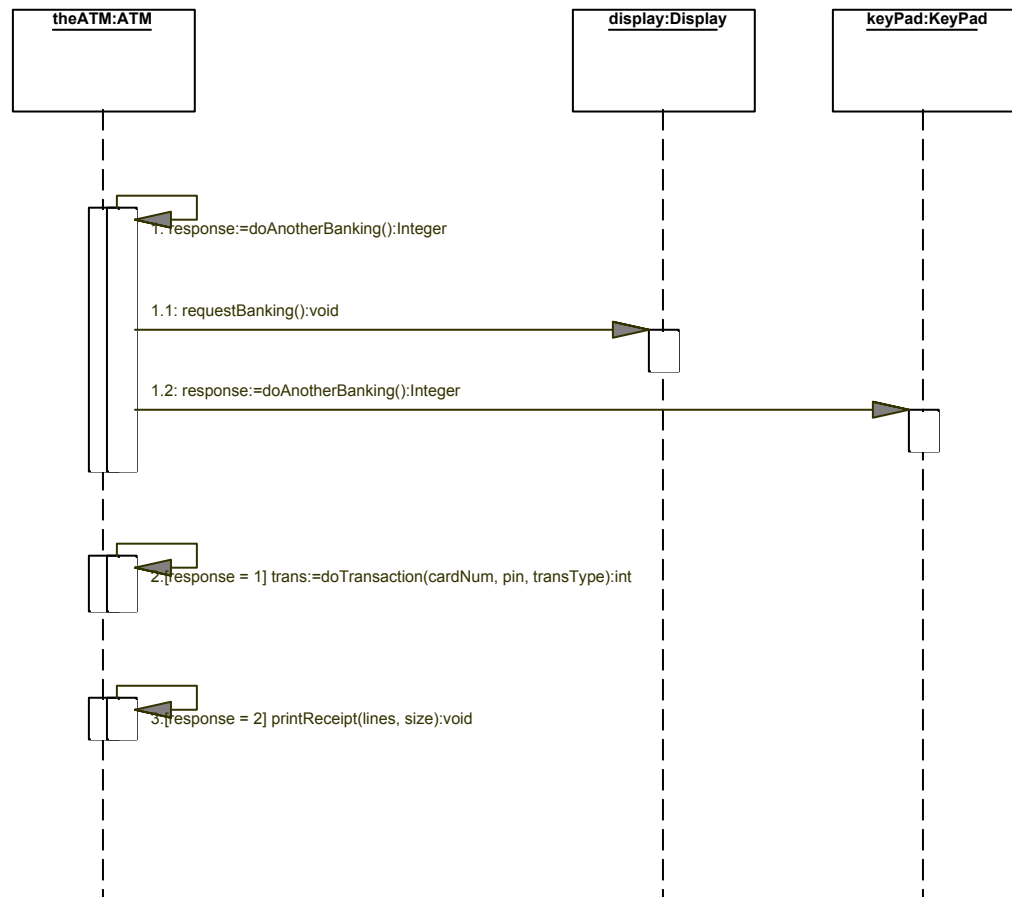


Figure E4: Sequence (Interaction) Diagram for the *AskingDoAnother* Use Case

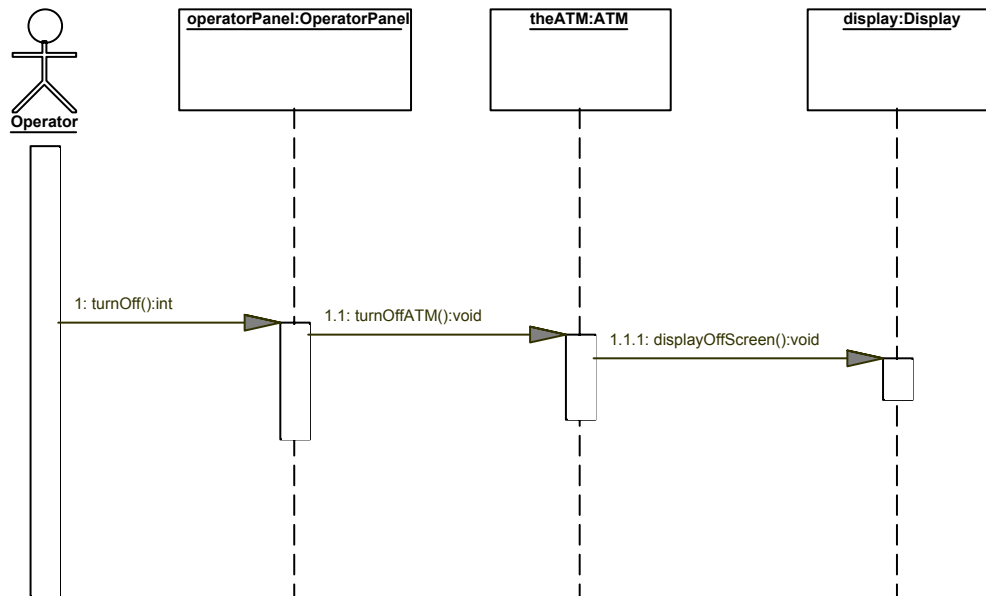


Figure E5: Sequence (Interaction) Diagram for the *ATMShutOff* Use Case

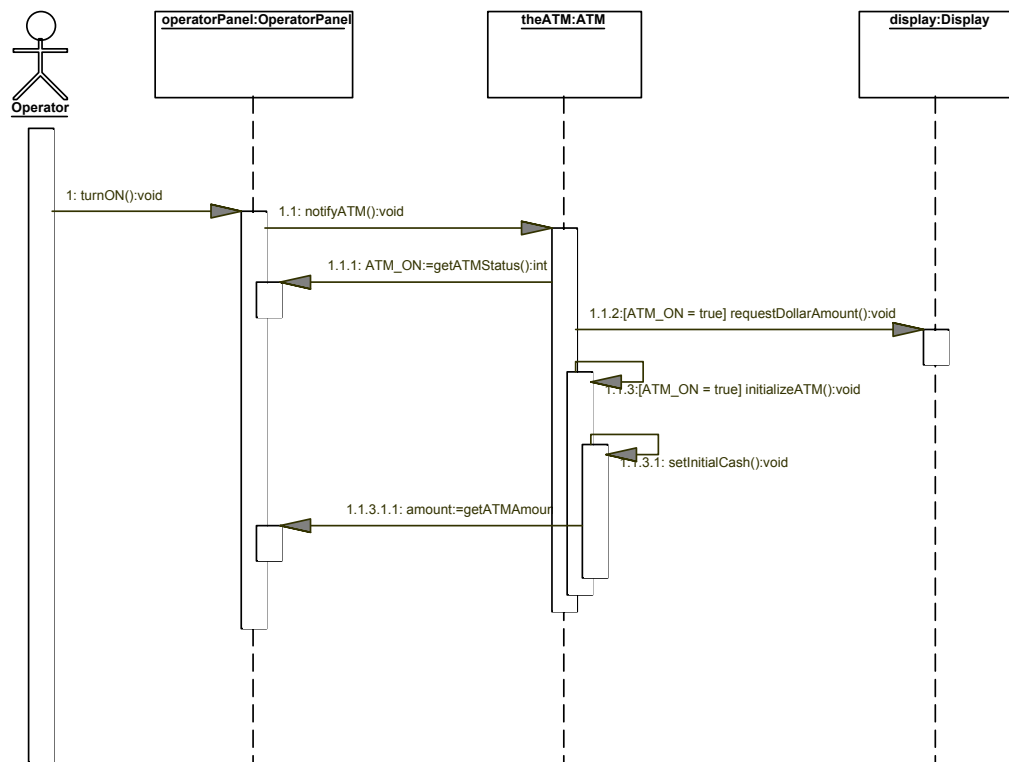


Figure E6: Sequence (Interaction) Diagram for the *ATMStartUp* Use Case

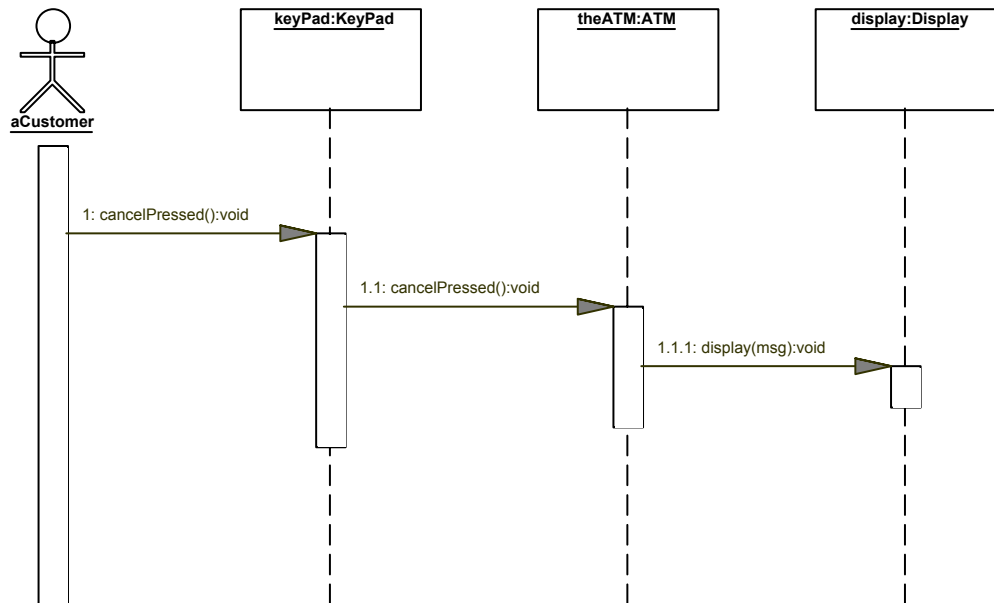


Figure E7: Sequence (Interaction) Diagram for the *Cancel* Use Case

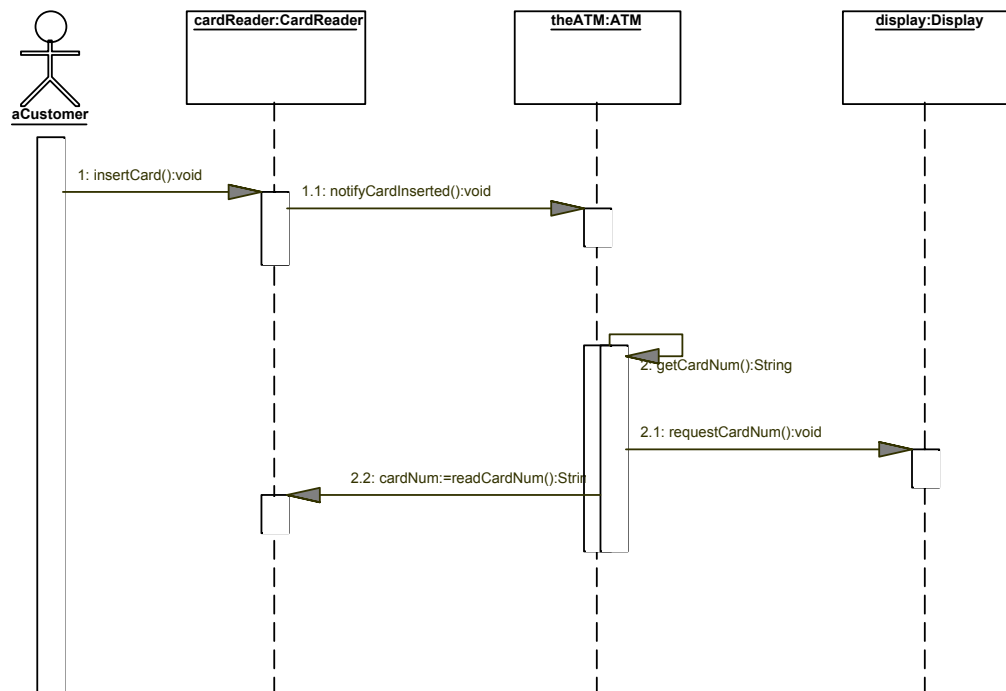


Figure E8: Sequence (Interaction) Diagram for the *CardInsert* Use Case

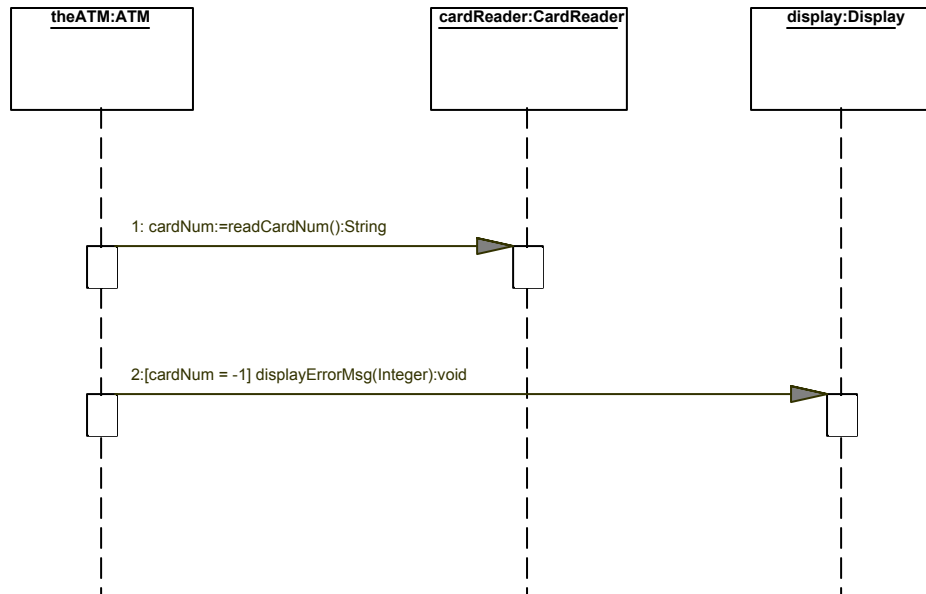


Figure E9: Sequence (Interaction) Diagram for the *CardNotReadable* Use Case

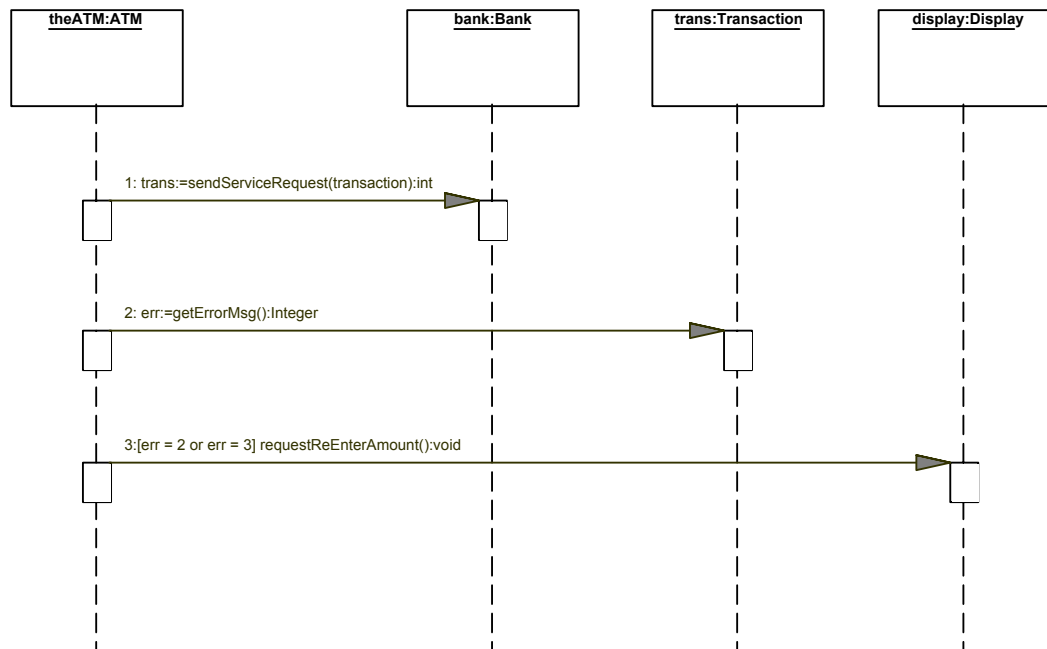


Figure E10: Sequence (Interaction) Diagram for the *FailedTransaction* Use Case

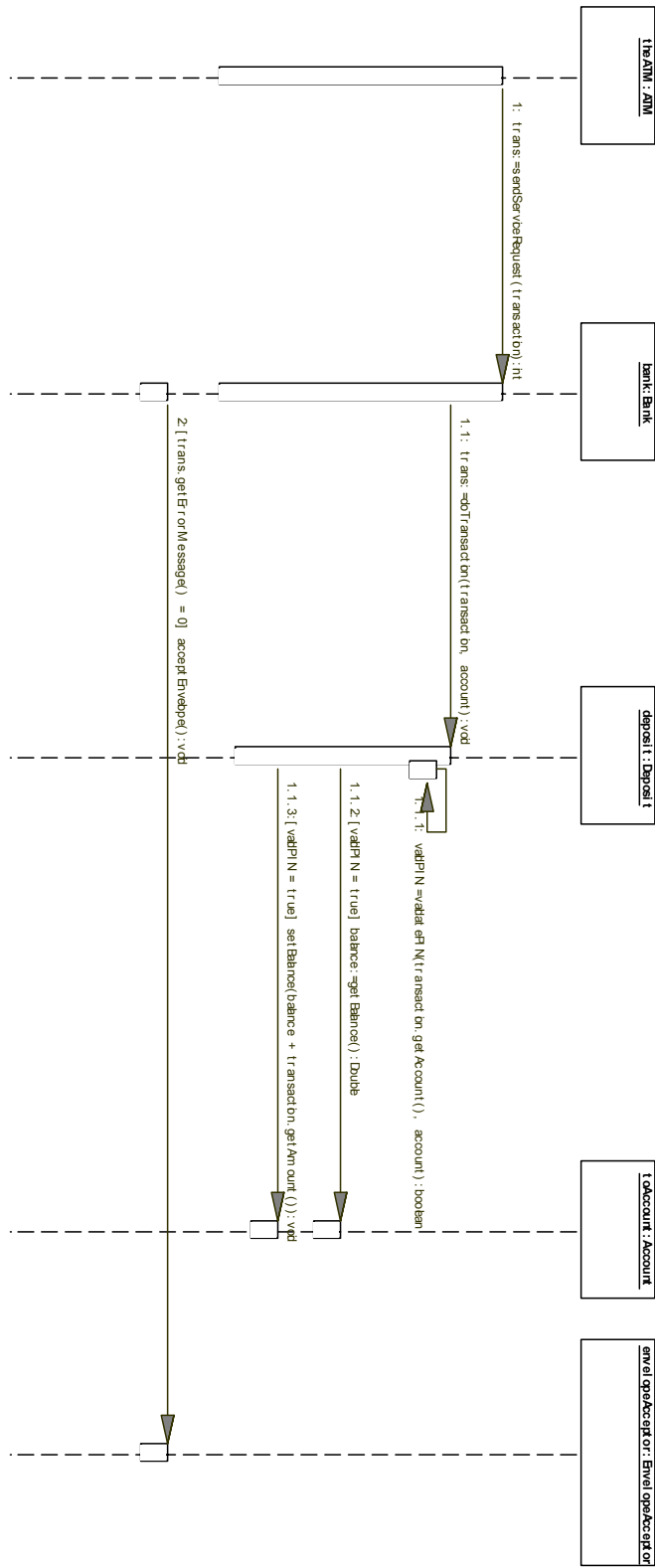


Figure E11: Sequence (Interaction) Diagram for the *Deposit* Use Case

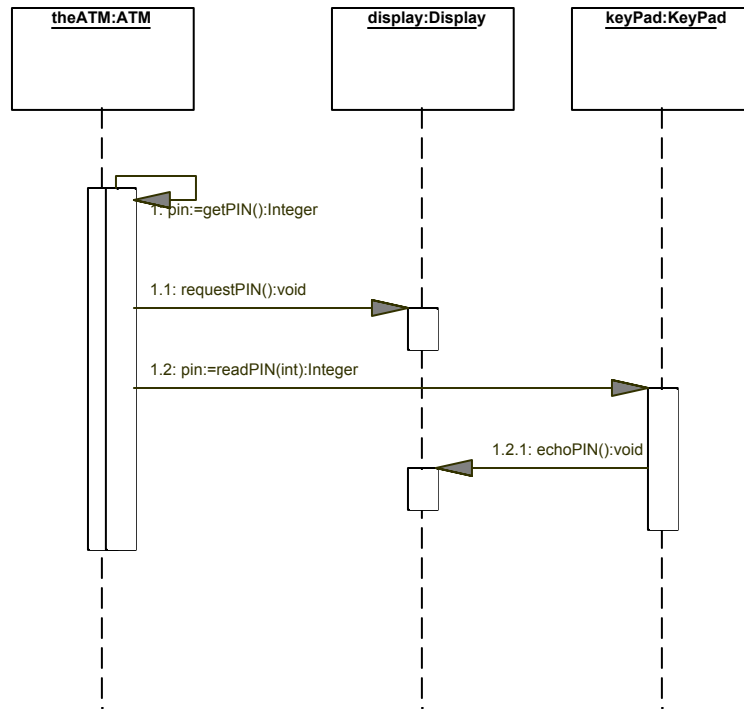


Figure E12: Sequence (Interaction) Diagram for the *GetPIN* Use Case

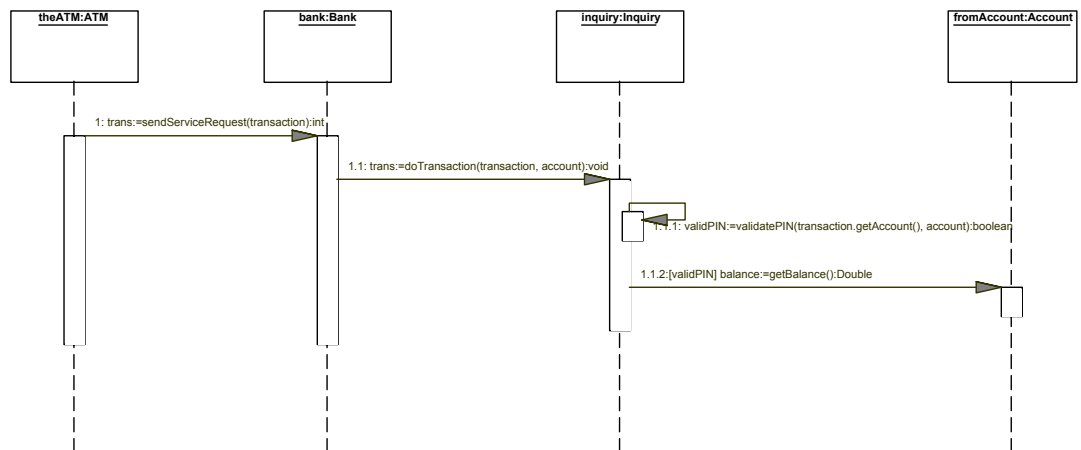


Figure E13: Sequence (Interaction) Diagram for the *Inquiry* Use Case

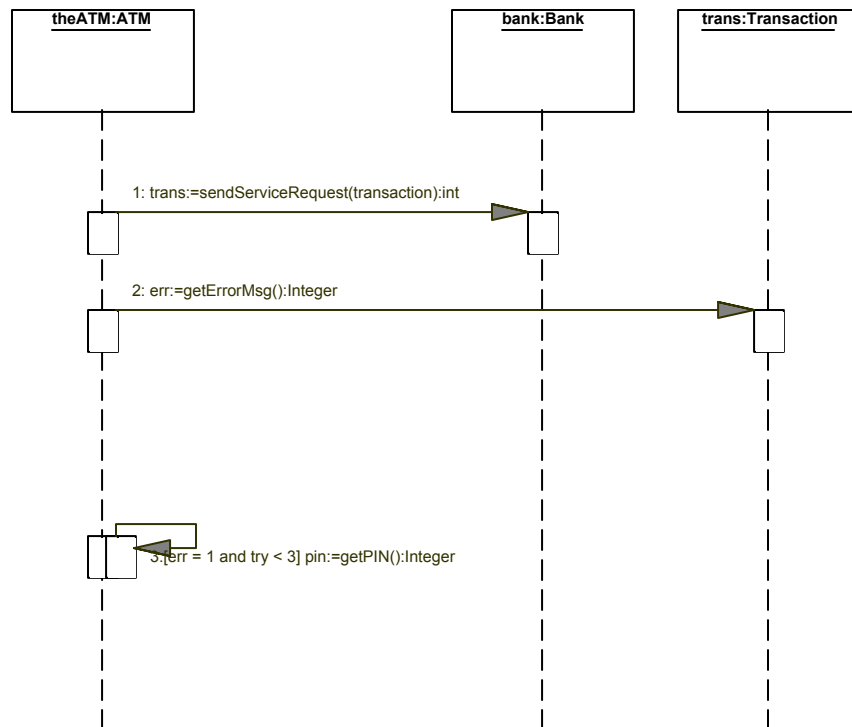


Figure E14: Sequence (Interaction) Diagram for the *PINInvalid* Use Case

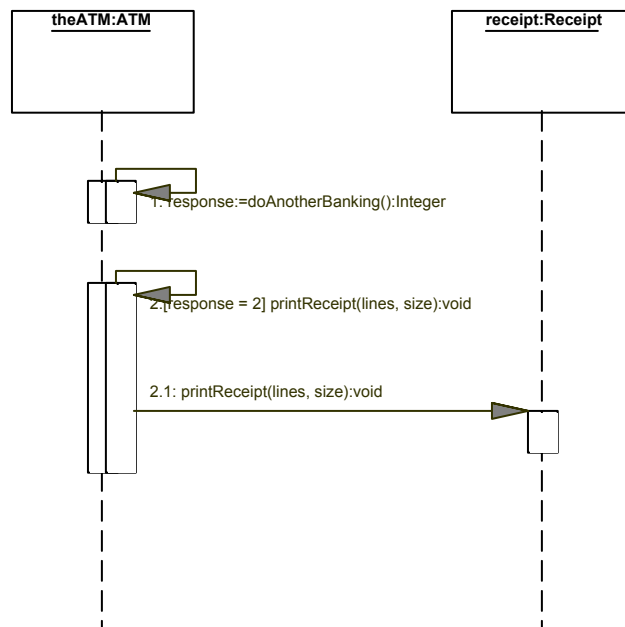


Figure E15: Sequence (Interaction) Diagram for the *PrintReceipt* Use Case

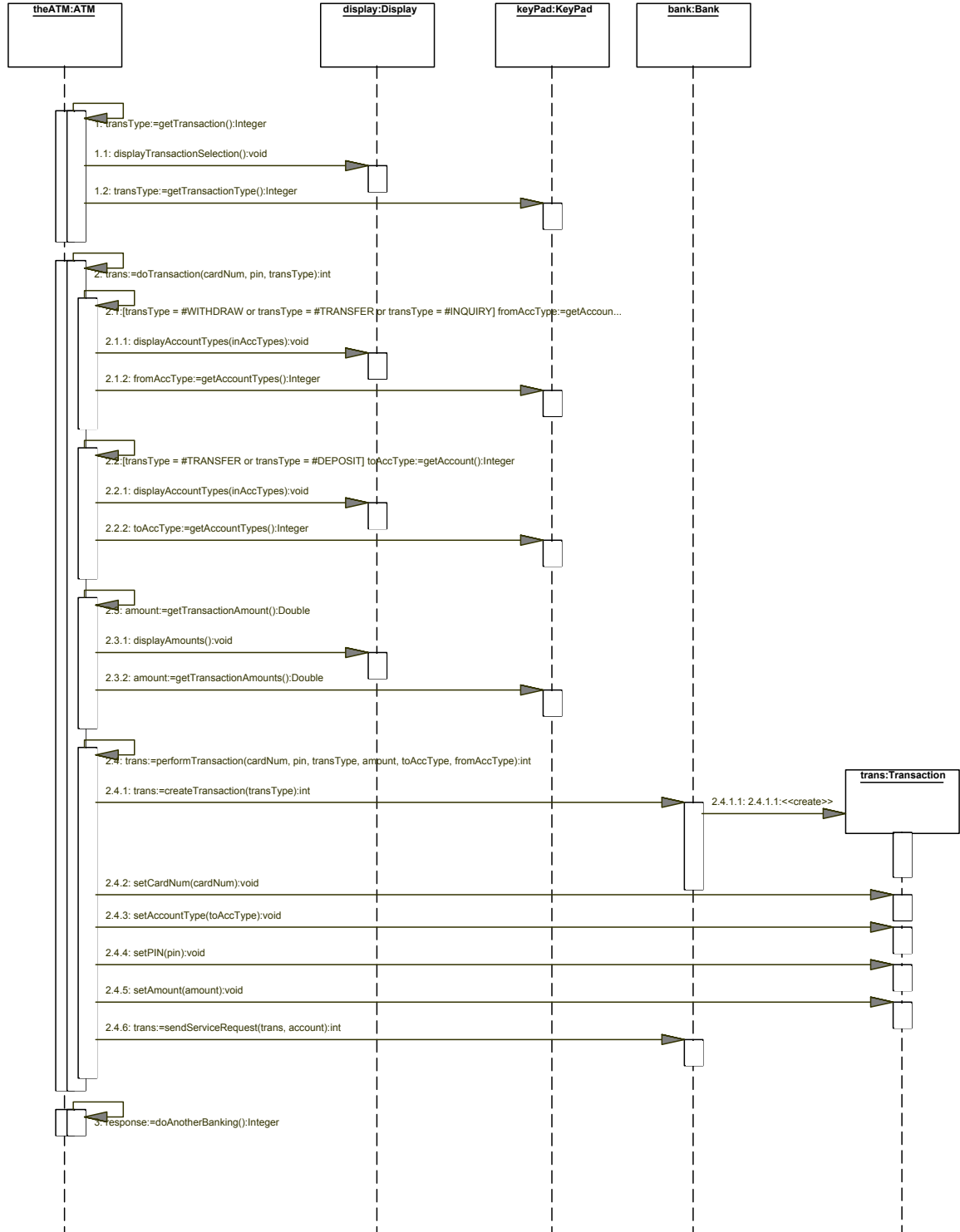


Figure E16: Sequence (Interaction) Diagram for the *Transaction* Use Case

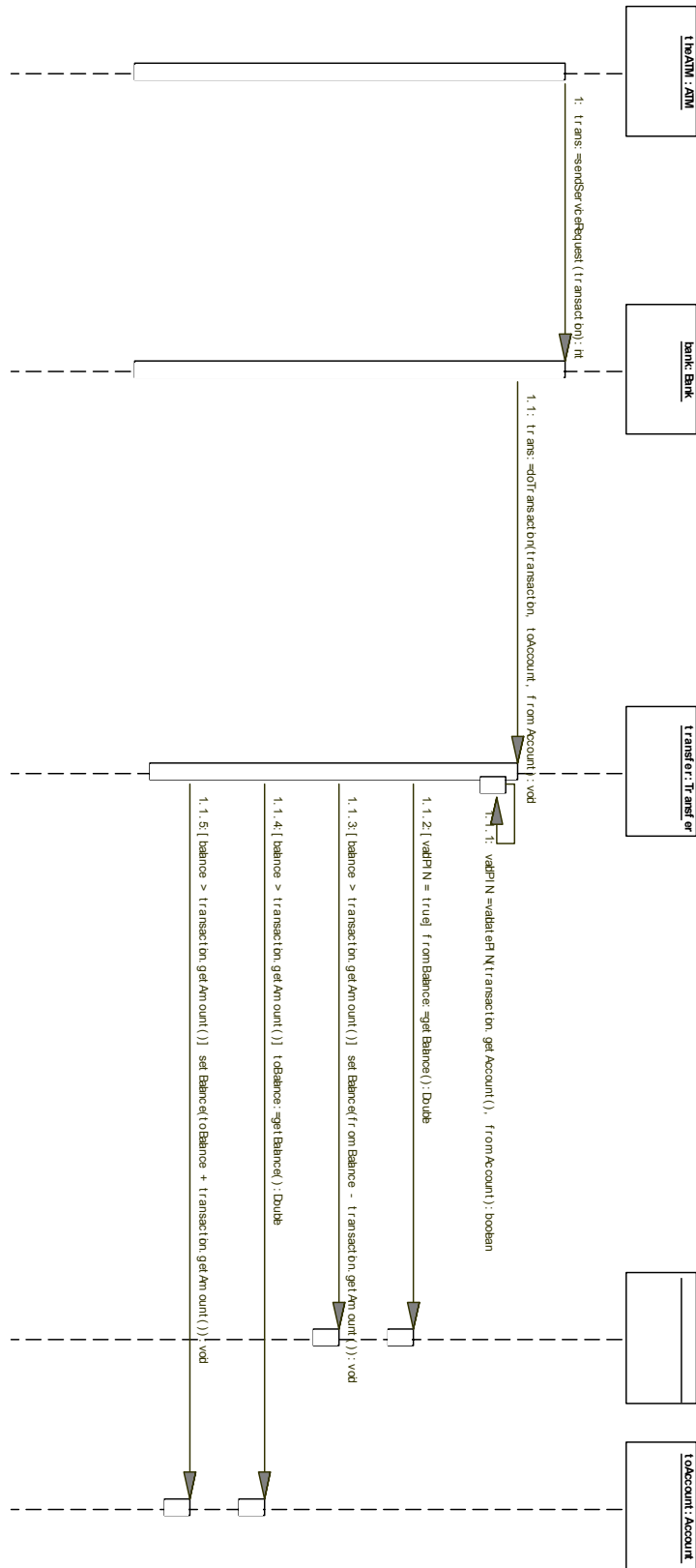


Figure E17: Sequence (Interaction) Diagram for the *Transfer* Use Case

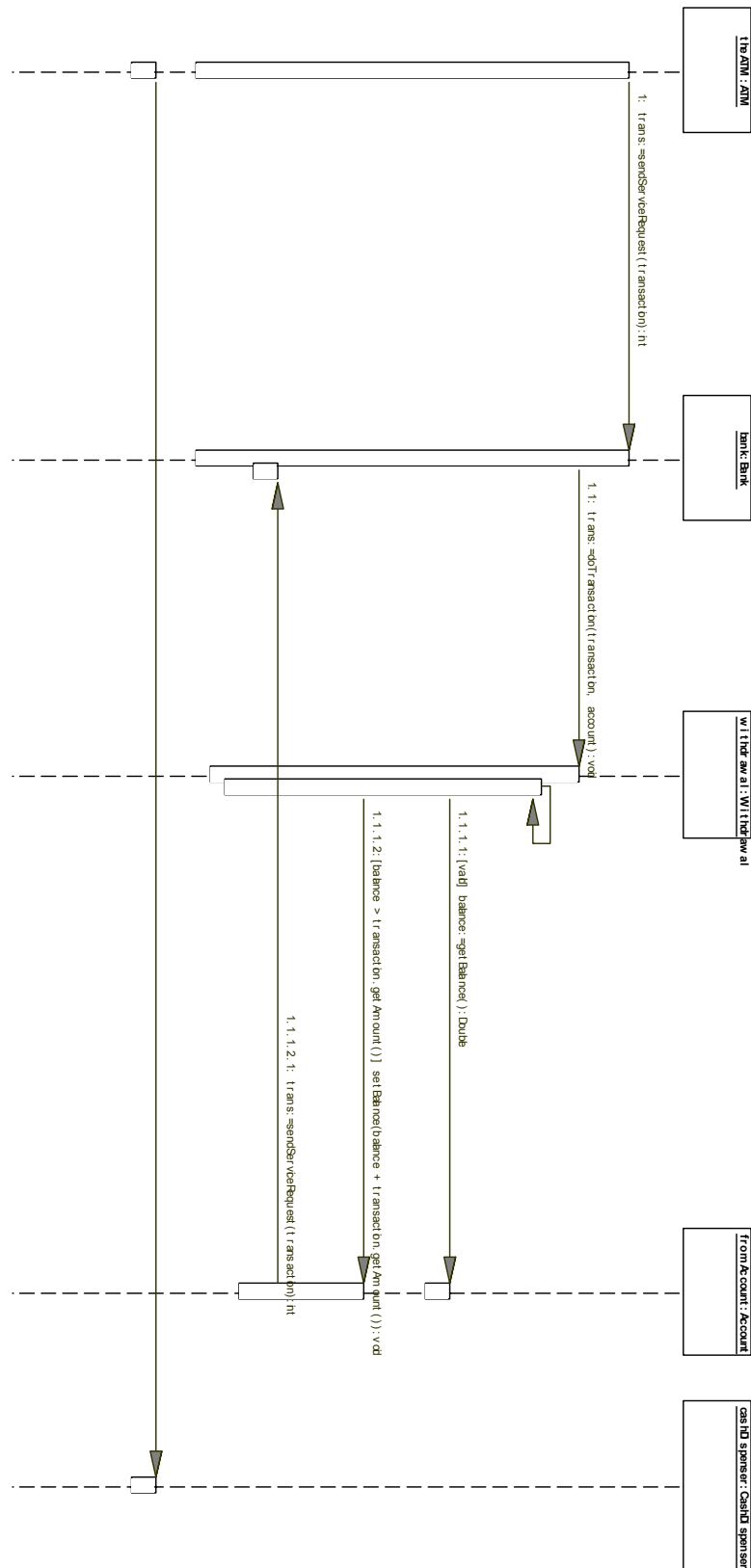


Figure E18: Sequence (Interaction) Diagram for the *Withdraw* Use Case