

# 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  
{briand, labiche, leeosul}@sce.carleton.ca

## Abstract

*The use of Unified Modeling 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.*

## 1. Introduction

The use of UML (Unified Modeling Language) analysis/design models [8] on large projects leads to a large number of inter-dependent UML diagrams (that may also contain OCL [14] constraints, e.g., contracts, guard conditions). 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 implementation of such changes. Then a proper decision can be made earlier—before any change detailed implementation is considered—on whether to implement a particular (set of) change(s) based on what design elements are likely to get impacted and thus on the likely change cost. Earlier decision-making and change planning is clearly important in the context of rigorous change management. On the other hand, since UML models describe the system at a higher level of abstraction than the code, model-based approaches may provide less precise results than code-based ones. For example, it may be possible that new, unexpected impacts appear at

implementation time. This is an issue that requires further investigation but will not be addressed in this article.

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™ [12], 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 [14]) 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 paper 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 detail each of the most important aspects of the approach and provide 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]), whereas a few of them are model-based. Kung et al. [10] 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 discusses the impact of object-oriented characteristics (e.g., encapsulation, inheritance, polymorphism, dynamic binding) on such an analysis. In [13], 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 [6]. 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 role<sup>1</sup>) in

---

<sup>1</sup> 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).

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 properties may change)<sup>2</sup>. 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. [7] have explored the use of coupling measures and predictive statistical model for that purpose.

## 4. Overview of the approach

First note that, due to space constraints, we do not present all the details of our change impact analysis strategy. Rather we concentrate on the important notions, providing excerpts for all the four steps that are involved in the strategy: consistency checking, change detection, change impact analysis, prioritization of impacts. Further details can be found in [5].

As mentioned above, the identification of model inconsistencies is important to ensure that the impact analysis algorithms we use yield correct results.

---

<sup>2</sup> Even when no model property changes, the model element implementation may require change.

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 analysis on UML diagrams. We have identified 120 consistency rules<sup>3</sup>. For example, one simple rule we use can be described informally as<sup>4</sup>:

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 categories<sup>3</sup> (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 rules<sup>3</sup>.

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 a measure 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.

---

<sup>3</sup> 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.

<sup>4</sup> Note that it can also be expressed using OCL on the meta-model.

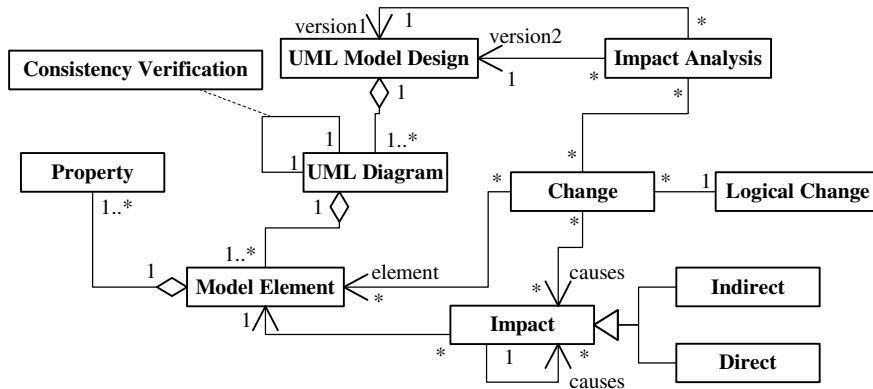


Figure 1 Conceptual model

## 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, reportGeneration, consistencyVerification, impactAnalysis, and control. The subsystem decomposition is shown in Figure 2 with packages and dependencies among them. More architectural details can be found in [5]. 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 [9]) 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 [11] 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 [5] and, due to space constraints, only 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.

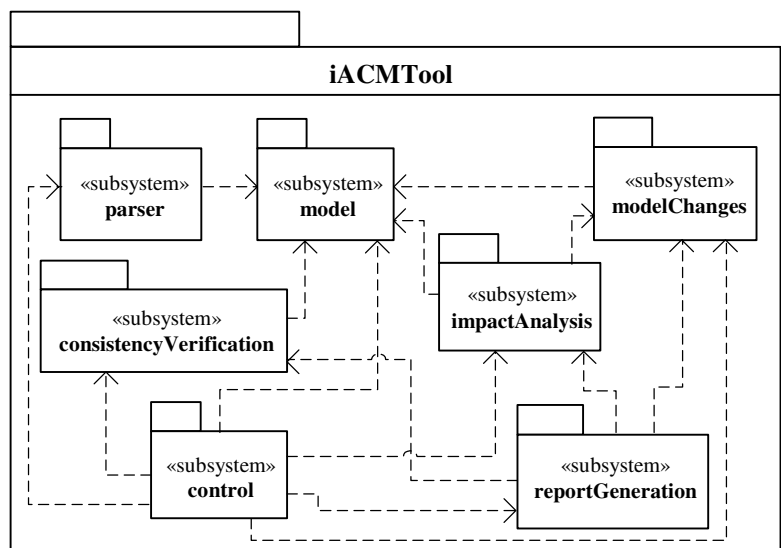
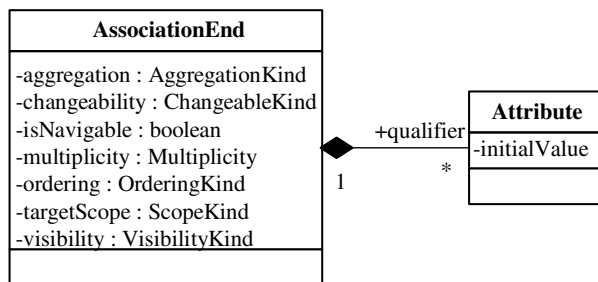


Figure 2 Impact analysis tool subsystems

## 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 paper.

### 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 [5]. 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.

#### Changed Sequence Diagram View – Added Message

**Change Code:** CSDVAM

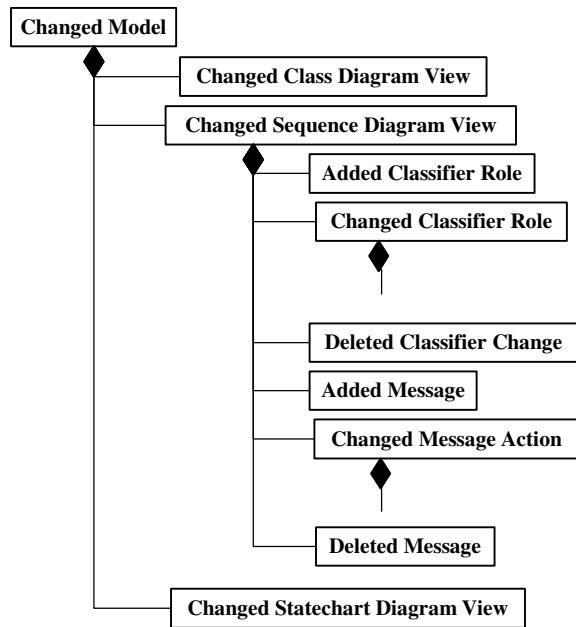
**Description:** In the modified model version there exists a message that does not exist in the original version.

#### OCL Expression:

```

context model::behaviouralElements::collaborations::SequenceDiagramView
self.message->select(
    exists(
        mNew:Message | not self.model.application.originalModel.
sequenceDiagramView.message->exists(
        mOld:Message | mNew.getIDStr() = mOld.getIDStr()
    )
)
)
  
```

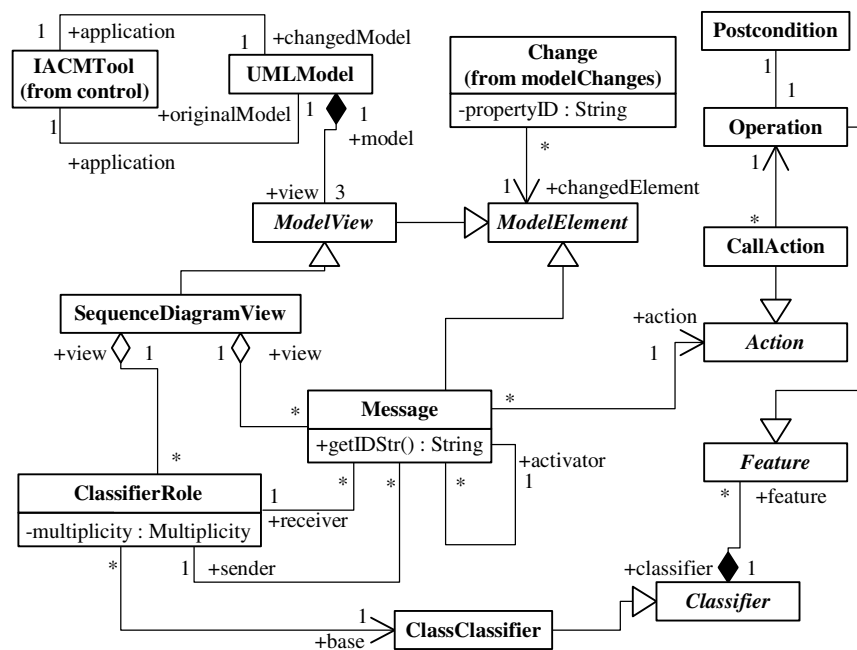
**Figure 4 Example change type**



### Figure 5 Excerpt of Change Taxonomy

Figure 6 shows an excerpt of the model subsystem class diagram (with a link to the `Change` class in `modelChanges`) 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 which are not shown here and are available in [5]. The taxonomy has been designed so that we could define precise impact analysis rules for every leaf change category.



**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 bags) of elements, corresponding to elements of different types (e.g., classes, operations), that are potentially impacted by a particular change (e.g., added message). These collections are bags, i.e., collections with possibly several occurrences of an element [14], because it is possible that an element is impacted in several ways by a particular change. 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  $\forall i \in I, \exists e \in E \setminus E'$  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.

**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:**

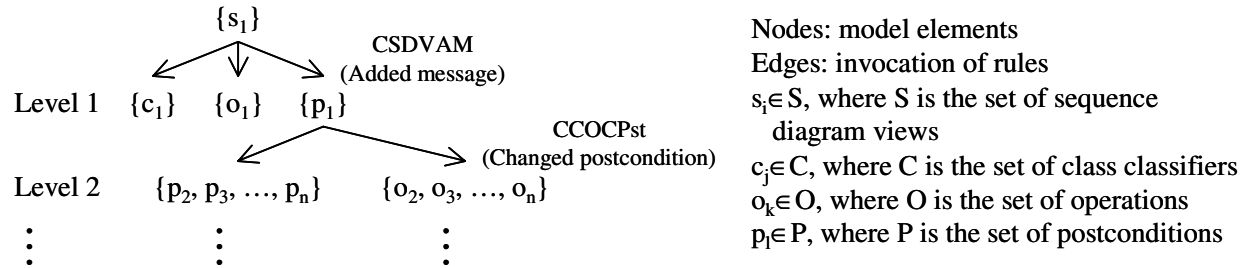
```
context modelChanges::Change def:
  let addedMessage:Message = self.changedElement.oclAsType(SequenceDiagramView).
    Message->select(m:Message | m.getIDStr()=self.propertyID)
  let sendingOperation:Operation = (
    if addedMessage.activator.action.oclIsTypeOf(CallAction) then
      addedMessage.sender.base.operation->select(o:Operation |
        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
```

**Figure 7 Impact Analysis Rule Example**

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 (CSDVAM – see example of change detection rule in Figure 4), 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 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`).

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



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

identifies the base class of the classifier role 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 diagram<sup>5</sup>. This identification involves selecting (the `select` operator) the operation declaration in the class that corresponds to the invoked operation in the sequence diagram, and is realized by the `equals()` operation in the OCL expression. This operation's complexity stems from overloaded operations, as described in [4, 6], 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 assumption<sup>6</sup> 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 ( $p_1$ ), 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, 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).

## 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 perform 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 (see [5] for all the diagrams) contains a class diagram (19 classes such as ATM, Bank, Withdrawal) and a use case diagram (15 use cases such as Transaction, Withdrawal, CardNotReadable, GetPIN) – each use case being associated with a sequence diagram. Most of the sequence diagrams contain between 3 and 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

<sup>5</sup> 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.

<sup>6</sup> This fundamental assumption seems reasonable, but empirical investigations are warranted to validate it.

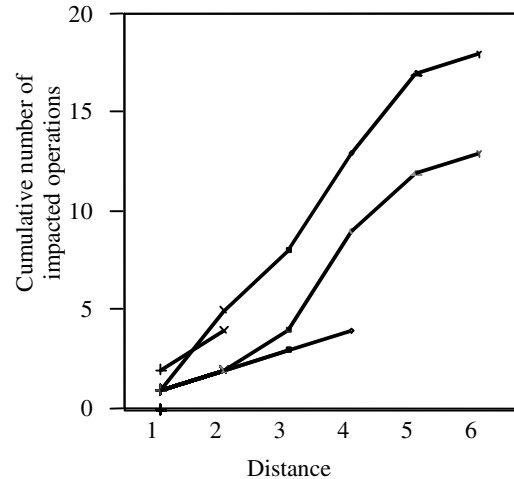


of three types: requirements changes (5), design improvements (2), and error corrections (3). They result in 70 model element changes, out of which 54 have shown impacted elements (the distribution of these changes for the elements in the taxonomy can be found in [5]). Let us take a few examples of logical changes and describe them. One logical change stems from our need 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 [5].

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 farther than a distance of 2. The reason is that the impacted elements at distance one and two that do not propagate are classes, and a 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 are 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.

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



**Figure 9 Cumulative number of impacted operations vs. distance**

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 [5] 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 paper 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, and a Canada Research Chair grant. 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](http://www.sce.carleton.ca/Squall/Totem)). We would like to thank Ashish Jain, Devasis Bassu, and Rabih Zbib, from Telcordia Technologies, for their valuable feedback during all stages of the project. 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](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 L. O'Sullivan, "Impact Analysis and Change Management of UML Models," Carleton University, Technical Report SCE-03-01, [http://www.sce.carleton.ca/Squall/Articles/TR\\_SCE-03-01.pdf](http://www.sce.carleton.ca/Squall/Articles/TR_SCE-03-01.pdf), February, 2003.
- [6] 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.
- [7] 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.
- [8] B. Bruegge and A. H. Dutoit, *Object-Oriented Software Engineering - Conquering Complex and Challenging Systems*, Prentice Hall, 2000.
- [9] T. J. Grose, S. A. Brodsky and G. C. Doney, *Mastering XML: Java Programming with XML, XML, and UML*, John Wiley & Sons, 2002.
- [10] 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.
- [11] OMG, "Unified Modeling Language (UML)," Object Management Group V1.4, [www.omg.org/technology/uml/](http://www.omg.org/technology/uml/), 2001.
- [12] TogetherSoft™, "Together", [www.togethersoft.com](http://www.togethersoft.com).
- [13] A. Von Mayrhauser and N. Zhang, "Automated Regression Testing using DBT and Sleuth," *Journal of Software Maintenance*, vol. 11 (2), pp. 93-116, 1999.
- [14] J. Warmer and A. Kleppe, *The Object Constraint Language*, Addison-Wesley, 1999.