



Automated impact analysis of UML models

L.C. Briand *, Y. Labiche, L. O'Sullivan, M.M. Sówka

*Software Quality Engineering Laboratory, Systems and Computer Engineering Department, Carleton University,
1125 Colonel By drive, Ottawa, Ontario, Canada K1S5B6*

Received 16 February 2005; received in revised form 22 April 2005; accepted 1 May 2005

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, UML diagrams undergo changes that address error corrections and changed requirements. Those changes can in turn lead to subsequent changes to other elements in the UML diagrams. 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 that change. In this article, we propose a UML model-based approach to impact analysis that can be applied before implementation of changes, thus allowing early decision-making and change planning. We first verify that the UML diagrams in a design model are consistent. Then the changes between two different versions of UML models are automatically identified according to a change taxonomy. Next, model elements which are directly or indirectly impacted by the changes (i.e., may undergo changes) are determined using formally defined impact analysis rules (defined with the 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, and two case studies that validate both the methodology and the tool. Empirical results confirm that distance helps determine the likelihood of change in the code.

© 2005 Elsevier Inc. All rights reserved.

Keywords: Impact analysis; UML; Design

1. Introduction

The use of UML (Unified Modeling Language) analysis/design models (Bruegge and Dutoit, 2000) on large projects leads to a large number of inter-dependent UML diagrams (that may also contain OCL (Warmer and Kleppe, 1999) constraints, e.g., contracts, guard conditions). As software systems evolve, UML diagrams undergo changes. Such changes to a diagram may lead to subsequent changes to other elements of the model's

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 on the system models and code. This can in turn help predict the cost and complexity of changes and help decide whether to implement them in a new release (Bohner and Arnold, 1996a).

In the context of large software development teams, the above problems are even more acute as diagrams may undergo changes in a concurrent manner as 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.

* Corresponding author. Tel.: +1 613 520 26 00; fax: +1 613 520 57 27.

E-mail address: briand@sce.carleton.ca (L.C. Briand).

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 (Bohner and Arnold, 1996a).

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. A proper decision can therefore be made—before any detailed implementation of the change is considered—on whether to implement a particular (set of) change(s) based on what design elements are likely to be 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 their code-based counterparts. 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 article.

Another assumption made by any model-based impact analysis method is that the model is up-to-date and therefore consistent with the code. This is often an issue in many software development organizations. There are several reasons for this problem:

1. Current UML tools do not provide many functionalities that would allow people to exploit models, for example in terms of test case specification or impact analysis. If there is no significant ways to benefit from models once the system is developed, then there is no incentive to maintain them.
2. Even if there were the motivation to maintain models, current tools are not always effective at supporting change. They do not always provide ways to check the consistency of diagrams (according to configurable rules) or perform impact analysis on diagrams. However, the functionality to manage traceability and consistency between design models and code is now available in some UML CASE tools. For example, Together® (Borland, 2003), 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.
3. And there is the usual problem of training and education. Most software engineers are not yet familiar with model-driven development, though this is the case of most new graduates.

However, even with this state of practice, we believe that our work is still relevant because it can address points 1 and 2 above. It provides one more way to exploit models (early impact analysis) and it provides a mechanism to help changes in models (through model impact analysis).

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 (Warmer and Kleppe, 1999)) and analyzing changes in actual case studies. Note that our work started long before the recent adoption of the new UML 2.0 standard (OMG, 2003). However, though minor adjustments are required, the overall feasibility and principles of our approach are not affected.
- A prototype tool implements the above principles using a carefully thought-out architecture and an extensible design.
- Two case studies have been performed to assess the feasibility and practical challenges of our approach.

This article describes the methodological framework and the fundamental principles underlying the change detection and impact analysis rules,¹ and reports on two case studies. Section 2 discusses related works. Section 3 provides a precise description of the problems we address, the objectives of our research, and an overview of the approach. Section 4 through Section 6 detail each of the most important aspects of the approach and provide examples. Section 7 presents the case studies and the last section, Section 8, outlines our main conclusions and future work.

¹ Details on our tool (iACMTool) are not provided in this article but can be found in Briand et al. (2003).

2. Related works

Bohner (2002) 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) overestimating 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 Bohner and Arnold (1996b)), whereas a few of them are model-based. In Kung et al. (1994), the authors describe 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. The analysis, however, is rough as only static information (the class diagram) is used. In Von Mayrhauser and Zhang (1999), the authors use a functional model (referred to as “domain model”, which is not based on UML) of the system under consideration to generate test cases, build a mapping between changes to the domain model and the impact it has on test cases, and classify the test cases according to the mapping. Another method for regression test selection, based on UML models (class and sequence diagrams), is presented in Briand et al. (2002a). 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.

A recent work, performed concurrently to ours, addresses the identification of impacts on UML models when system requirements (e.g., use case descriptions) undergo changes (von Kethen and Grund, 2003). The user derives a first set of impacted elements by following traceability links between textual use case descriptions and model elements. Then impact analysis classifies impacted elements into *primary impacts* (elements that have to be changed), *secondary impacts* (elements that have to be changed with a high probability), and *tertiary impacts* (elements that may have to be changed). However, the three categories of impacts are not described with enough technical details (e.g., what is a high probability and how is it determined?) thus preventing us from comparing our approach with theirs in terms of impact analysis rules.

3. Problem definition, objectives, and overview of the approach

Automating impact analysis of UML design models can be decomposed into several sub-problems: Verify the consistency of changed diagrams; Automatically detect and classify changes across different versions of UML models; Perform an impact analysis to determine the potential side effects of changes in the design; Prioritize the results of impact analysis according to the likelihood of occurrence of predicted impacted elements.

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 Briand et al. (2003). The concepts presented in this section, and their relationships, are illustrated by the conceptual model in Fig. 1 (using a UML class diagram). Note that Fig. 1 is also an excerpt of the class diagram of our prototype tool (Briand et al., 2003), referred to as iACMTool.

3.1. Verify the consistency of changed diagrams

The original and modified models must be self-consistent for any impact analysis algorithm to provide correct results. 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) between two model versions, but structural inconsistencies between UML diagrams of a single model, e.g., a class instance (classifier role²) in a sequence diagram whose class is not in the class diagram.

Since consistency in complex UML models is not always easy to achieve, verifying consistency must be supported by tools. Inconsistencies may be automatically modeled and detected by a set of *consistency rules*, also called *well-formedness rules* in OMG (2001). Each rule corresponds to one type of inconsistency and must be implemented in any tool supporting impact analysis on UML diagrams. We have used 120 of the rules defined in OMG (2001). For example, one simple rule we use can be described informally³ as:

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

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).

³ It can also be expressed using OCL on the UML meta-model as shown in OMG (2001).

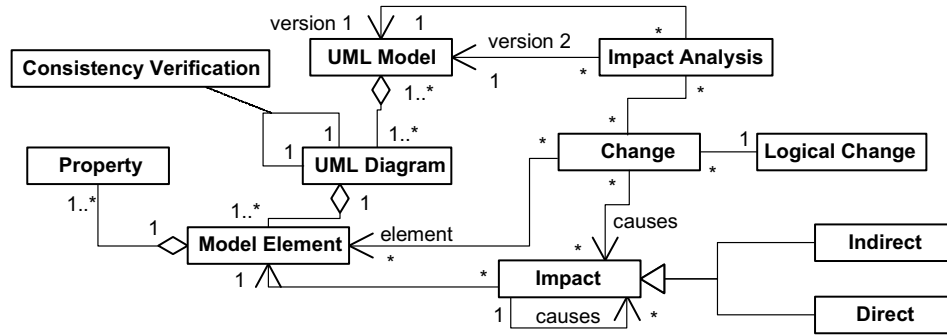


Fig. 1. Conceptual model.

3.2. Automatically detect and classify changes

Ideally, one modifies a UML model and then uses the impact analysis tool to automatically identify all the changes performed since the last version (once the tool has verified that each version of UML diagrams is consistent). 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.

In UML, a model element is a constituent of a model (OMG, 2001): An attribute, a classifier (e.g., a class) are model elements. Each model element is defined by a set of *properties*, which is a named value denoting a characteristic of an element (Briand et al., 2002a; OMG, 2001): An attribute is a property of a classifier (e.g., a class), a name is a property of an attribute.

Thus, the identification of a change to a model element requires checking if any of its properties has changed. Each model element change (class *Change* in Fig. 1) is classified according to a *change taxonomy* in order to associate impact analysis rules with each type of change. The change taxonomy we defined reflects changes to class diagrams, sequence diagrams, and statecharts. Some more details are provided in Section 4. The complete change taxonomy contains 97 change categories.⁴

3.3. Perform an impact analysis

Once we have verified that the diagrams of a UML 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 5): class *Impact* and its subclasses in Fig. 1. 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 most cases, 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).⁵ 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 (in a library system, a requirement change could be to allow customers to check out borrowed books themselves without asking a librarian). 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. For instance, if logical changes *A* and *B* result in *x* and *y* changed model elements respectively, impact analysis can be performed for any one of the changed model elements separately, for either logical change *A* or *B* (thus only considering *x* or *y* changed model elements), or for both logical changes all together (thus considering $x + y$ changed model elements).

3.4. Prioritize the results of impact analysis

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. In order for impact analysis to

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 Even when no model property changes, the model element implementation may require change. For example, a change in an operation's algorithm will not necessarily be visible as a change of the operation's contract or any other change in the model.

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. For example, Briand et al. (1999) have explored the use of coupling measures and predictive statistical model for that purpose. To do so, we define a measure of distance between the changed elements and potentially impacted elements (Section 6) where the assumption is that the larger the distance, the less likely is the model element to be impacted.

4. 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 meta-model (OMG, 2001). 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 (class Classifier in the meta-model) to which it belongs is considered impacted. (In the meta-model, there is a composition between classes Classifier and Feature, parent class of class Attribute.) 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, considering an excerpt of the meta-model in Fig. 2, 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 instances of Attribute) and others as attributes (e.g., isNavigable to model whether an association end is navigable).

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 article. We first provide some preliminary definitions. Then we define model element changes (changes to properties that do not uniquely identify model elements) and impact related elements (changes to elements impact other elements). Last we show how changes to properties that do uniquely identify model elements can have an impact on other elements.

Definitions:

Let E be the set of all model elements (i.e., instances of classes in the UML meta-model) in the UML model under study. Let Pe be the set of all the properties of $e \in E$, and $Pe'' \subset Pe$ be the set of properties that uniquely identify $e \in E$.

Definition 1: Model element changes

If any one $p \in (Pe - Pe'')$ is changed, then e is changed.

Definition 2: Impact related elements

$\forall e_1, e_2 \in E \bullet e_1 \neq e_2, e_2$ is said to be an impact related element of e_1 if when e_2 is changed then e_1 is considered changed.

Definition 3: Added/deleted model elements

$\forall e_1, e_2 \in E \bullet e_2$ is an impact related element of e_1 ,
 e_2 is added/deleted $\Rightarrow e_1$ is a changed element.

Using definitions above, a change taxonomy is provided in Briand et al. (2003). The UML class diagram notation is used to describe the taxonomy, as illustrated in Fig. 4. Each non-terminal node in the taxonomy represents an abstract change category of a model element. The leaf nodes are concrete change categories and correspond to one changed element property. The taxonomy (the class diagram) can be seen as a meta-model for the model element changes: Any change in a UML model is an instance of exactly one change category in the taxonomy.

For example, let us look at a simple change category example: Adding a message in a sequence diagram. We provide in Fig. 3 a description of the change category. Each change category has an acronym, a short textual description, and an OCL expression that shows how,

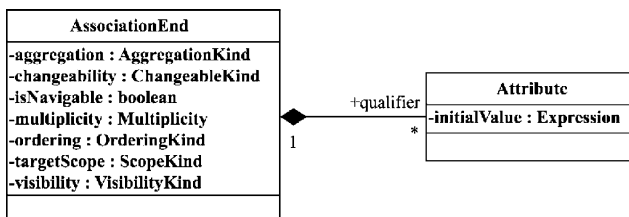


Fig. 2. Example of impact related element from the meta-model.

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( mNew:Message |
    not self.model.application.originalModel.
        sequenceDiagramView.message->exists( mOld:Message |
            mNew.getIDStr() = mOld.getIDStr()
        )
)

```

Fig. 3. Example change type.

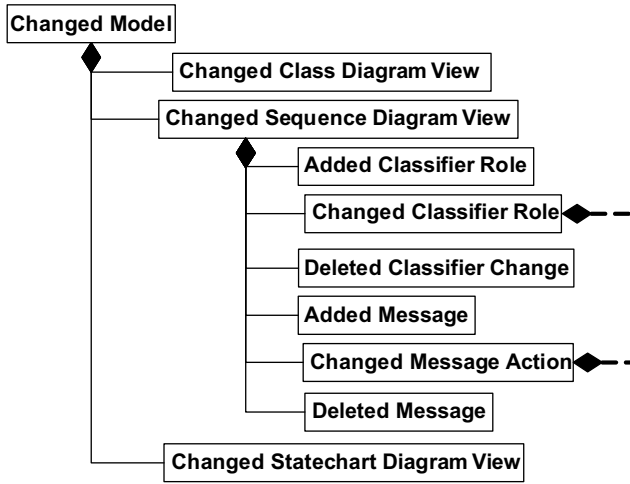


Fig. 4. Excerpt of change taxonomy.

based on the UML meta-model, the correspond-subsystem class diagram (which is instantiated by the parser subsystem), the corresponding 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 excerpt of the UML metamodel (in subsystem model in the iACMTool) and the modelChange class diagram (another iACMTool subsystem) are sufficient to implement a workable change retrieval algorithm.

Fig. 5 shows an excerpt of the model subsystem class diagram (with a link to the Change class in model-Change) that is navigated by the OCL expression of our example in Fig. 3. 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.

Fig. 4 shows an excerpt of the change taxonomy where our example change category (added message) is located. We see it is in the changed Sequence Diagram View, which may itself be composed (note the composi-

tion) 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 Briand et al. (2003). The taxonomy has been designed so that we could define precise impact analysis rules for every leaf change category.

5. Impact analysis rules

Each impact analysis rule is a specification (using OCL) of how to derive several collections 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 OCL bags, i.e., collections with possibly several occurrences of an element (Warmer and Kleppe, 1999), 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 the change (this cannot always be decided with certainty). There is one impact analysis rule for each change category in the taxonomy. As further described below, these rules are recursively applied in order to obtain model elements that are indirectly impacted.

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 $\forall i \in I \subseteq E', \exists e \in E \cap E'$ such that $e \neq i$ and, when recursively applying impact analysis rules from e , we obtain a bag of model elements containing i .

Though this is rare, note that the bag of impacted elements I may be empty, i.e., it is certain that no resulting

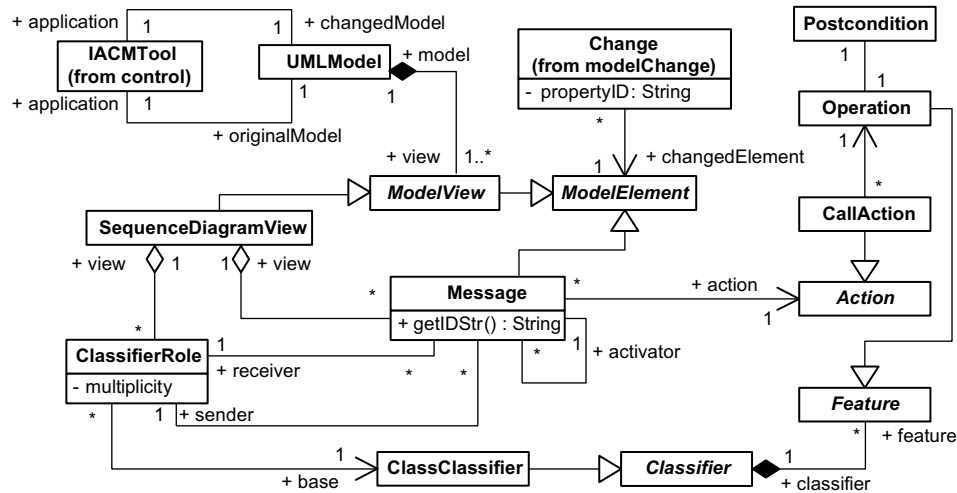


Fig. 5. Excerpt from the iACMTool class diagram (Briand et al., 2003).

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 Fig. 6 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 Fig. 3), 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 as a result of a changed property: 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 (Bohner, 2002). 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 Fig. 6, see meta-model excerpt in Fig. 5). 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 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 diagram.⁶ This identification involves selecting (the *select* operator) the method declaration in the class that corresponds to the invoked method in the sequence diagram, and is realized by the *equals()* operation⁷ in the OCL expression of Fig. 6. 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.

⁶ 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.

⁷ This operation is not trivial because of overloaded methods and we have to resort to heuristics since UML sequence diagrams do not show formal parameter and return types but only arguments (actual parameters) (Briand et al., 2002a,b).

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. Additionally, the operation postcondition may no longer accurately represent the effect (what is true on completion) of the operation: the postcondition is thus impacted.

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.sequenceDiagramView.message->select(m:Message |
    m.getIDStr()=self.propertyID)
  let sendingOperation:Operation = (
    if addedMessage.activator.action.ocliIsTypeOf(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
```

Fig. 6. Impact analysis rule example.

6. Distance measure

When impacts between model elements are indirect, following the general guidelines in Bohner (2002), we suggest using a distance measure between the changed model elements and the impacted elements. In Bohner (2002), it is stated that a common assumption⁸ 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.

Definition: Distance between a changed element and an impacted element

Let $\text{ImpactRule} \in E \times E$ be the relation that associates a changed model element to its direct impact related elements when applying once any one of the impact analysis rules: $e_1 \in E, e_2 \in E, e_1 \text{ ImpactRule } e_2 \iff e_1$ has a direct impact on e_2 .
 $e_1 \in E, e_2 \in E, e_1$ has a (direct or indirect) impact on $e_2 \iff \exists n \in \mathbb{N} \bullet e_1 \text{ ImpactRule}^n e_2$ where n is the distance between changed model element e_1 and impacted element e_2 . If the impact is direct then $n = 1$, otherwise, $n > 1$.

If we use Fig. 7 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 added message example of Fig. 6. This

⁸ This fundamental assumption seems reasonable, but empirical investigations are warranted to validate it.

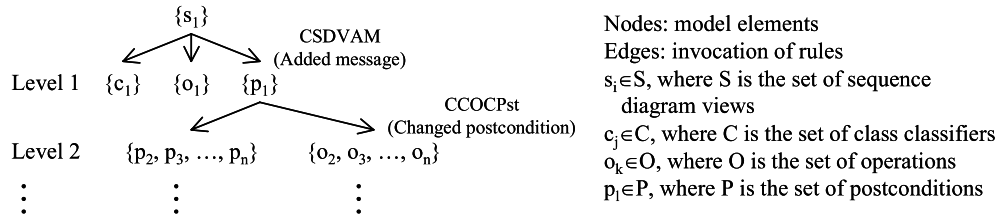


Fig. 7. Example distance between a changed element and an impacted element.

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 specified 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).

In our definition of distance, all the impact analysis rules have the same weight. However, one could argue that some rules may correspond to impacts occurring with higher probabilities than others and this will be investigated by future work. But it seems a priori difficult to establish a relationship between the rules and impact likelihood as it likely depends on the system under study (e.g., the semantics of the associations between classes). Moreover, results discussed next suggest that our definition of distance is sufficient.

7. Case studies

We have selected an Automated Teller Machine (ATM) and a Cruise Control (CC) system as case studies. Results are presented below in a consistent form for the two case studies and general conclusions are drawn in the next section. We first describe the procedure we followed to perform the case studies.

7.1. Procedure followed during case studies

The procedure is illustrated by the activity diagram in Fig. 8. For each system under study (SUS), we first defined a number of realistic logical changes to the original system and then determined their effect to produce updated UML diagrams. These logical changes were selected to be of three types: requirements changes, design improvements, and error corrections. We then used our prototype tool (iACMTool) to identify the changes between the two UML model versions and performed impact analysis (steps 1 and 2 in Fig. 8). This

resulted is a set of impacted elements in UML models, at various distances.

Concurrently, an investigation was done to examine the accuracy of the strategy with respect to the implementation (source code) of the model: Do direct and indirect impacts at various distances, as identified from the UML models, actually correspond to changes in the source code? By answering this question, we determine a set of false-positives that correspond to cases where there is an impacted model element (e.g., operation, class) but no change to the corresponding code. This task corresponds to steps 3–6 in Fig. 8. First, the impacts resulting from the logical changes made to the SUS's UML model were implemented in the SUS's source code. The modified source code was then tested (functional test cases) to ensure changes were correctly implemented, i.e., that added or changed functionalities were correctly implemented and that unchanged functionalities were not affected. The source code files of the two versions were then compared to identify changes, resulting in a set of impacted elements in the source code. Last (step 6), we checked the consistency of the two sets of impacted elements: the first set is produced by our tool, and the second by our analysis of the source code. Note that some logical changes do not result into any impacted element in the UML diagrams. Those changes are therefore not considered in the accuracy analysis.

The procedure in Fig. 8 is an attempt to perform an objective evaluation of the accuracy of the results produced by our approach and prototype tool based on UML diagrams. Though part of the analysis is manual and faults can be overlooked, the procedure leaves no room for subjective human interpretation.

For each of the two case studies discussed below, we first report on the results generated by the iACMTool, and then on the accuracy analysis.

7.2. Automated teller machine (ATM)

7.2.1. ATM description

The first version of UML documents contains 19 classes (e.g., ATM, Bank, Withdrawal) and 15 use cases (e.g., Transaction, Withdrawal, GetPIN, CardNotReadable), each use case being associated with a sequence diagram. Furthermore, 29 attributes

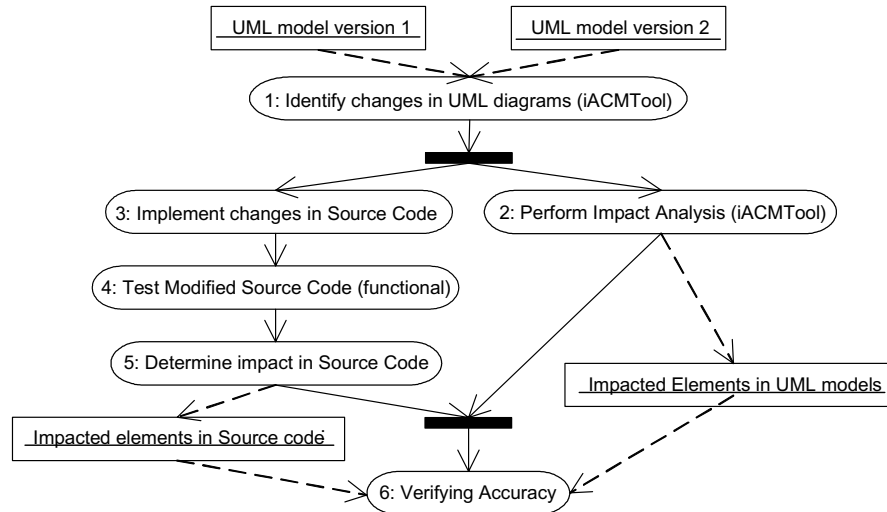


Fig. 8. Procedure followed during case studies.

and 72 operations appear in the class diagram, and classes are related by inheritance (4), association (11) and dependency (3) relationships. The system functionality can be summarized as follows: The customer inserts his/her card, enters a PIN and then performs a number of transactions such as withdrawal and deposit before a receipt is issued by the ATM at the end of all the transactions.

7.2.2. Impact analysis

We made eleven realistic, logical changes to the original version of the UML diagrams: requirements changes (5), design improvements (2), and error corrections (4). They result in 43 model element changes. Let us take a few examples of logical changes and describe them. One logical change (requirement change) stems from the need to be able to keep track of how many times per session a user attempts to enter the PIN—after three invalid PIN's the card will be retained. This logical change translates into 11 model element changes. Another logical change (design improvement) is to change class ATM state's representation from an integer to an enumeration class, and results into 23 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). The last logical change example consists of making class Account abstract since only its subclasses are instantiated (e.g., Saving).

Of the 43 model element changes resulting from the eleven logical changes, 29 resulted in impacts (at a distance above or equal to 1) among which only seven

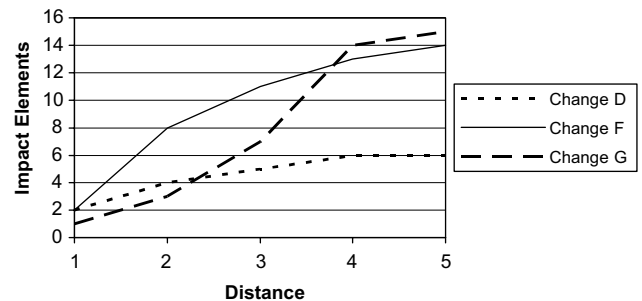


Fig. 9. Cumulative number of impacted elements vs. distance (ATM).

result in impacts at a distance (strictly) greater than one.⁹ This is due to a combination of: (1) The corresponding rules not identifying any impacts since it is certain that no further change is necessary to accomplish the original change; and (2) The necessary conditions to identify impacted elements have not been met. For example, the CCAO change (Changed ClassClassifier—Added Operation) does not result in any impacts since the model is assumed to be consistent (i.e., the added operation is likely used in a sequence diagram in an added message and this results in impacts to the caller operation). Another example is that a CAECM change (Changed AssociationEnd—ChangedMultiplicity) does not impact any other element when the changed association end is not navigable (a necessary condition for that change to impact other elements).

Let us consider all the impacted elements, when accounting for all changed model elements taken together, and their distance to the changed model

⁹ Twenty-nine model element changes result in impacts at a distance of 1 or above. Seven model element changes result in impacts at a distance of 2 or above. Three model element changes result in impacts at a distance of 3 or above.

elements. Fig. 9 plots, for each of the three model element changes that propagate to a distance (strictly) greater than two (namely changes D, F and G), the cumulative number of impacted elements for each distance value to a maximum of five. It shows curves representing the cumulative number of impacted elements (y -axis) for each distance value (x -axis) of the three changes. For instance, change model element F (plain line curve) results in impacted elements at various distances from one to five: two impacted elements at distance one, six at distance two, three at distance three (thus a cumulative number of 11 at distance 3), two at distance four and one at distance five. Note that we do not count impacts but impacted elements: each one is counted only once (i.e., the first time it is impacted), regardless of how many times it has been impacted.

Fig. 9 clearly shows that, when there is propagation of impacts, the curves are not exponential, as suggested in Bohner (2002), but rather linear. This is important as it suggests that our impact analysis rules are rather precise. 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,¹⁰ instead of connectivity graphs (see Bohner, 2002), 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 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 all model elements impacted together but the same graphs could be plotted for classes or operations separately, each showing the same trends as in Fig. 9 (though with significantly different distances and number of impacted elements at various distance values).

7.2.3. Results accuracy

In this investigation, a subset (six) of the eleven logical changes made to the ATM's UML model was checked against the ATM's source code. Those six logical changes were selected as they are the only ones that lead to impacted elements at a distance greater than one.

The results of the investigation are shown in Fig. 10. Results for only three of the six logical changes are reported as logical changes number 2, 6 and 11 are representative enough. Note that logical changes 6 and 11 include model element changes F and G (Fig. 9), respectively, and that logical change 2 does not include D, F, or G. For each of those three logical changes, Fig. 10 shows the number of impacted elements (as reported

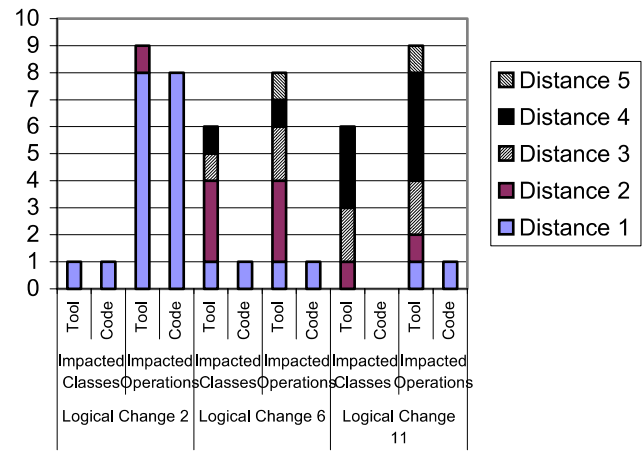


Fig. 10. Code impacts vs. iACMTool impact (ATM).

by our iACMTool tool) as well as code impacted classes and operations, i.e., impacts that affect the implementation (as determined by manual investigation), at different distances for classes and operations. For instance, the iACMTool reports, for logical change number 6, one class impacted at distance one, three at distance two, one at distance three and one at distance four (“Tool” bar), whereas our manual investigation reports that only the class impacted at distance one results in a change to the implementation (“Code” bar). This indicates that, for logical change number 6, classes impacted at a distance greater than one, as reported by the iACMTool, are false-positives. Note that in Fig. 10 impacted elements are counted as opposed to impacts—the set of impacts may include several impacts to a given element and when an element is impacted at various distances, only the minimum distance is considered in the analysis. Fig. 10 shows that there are no required code changes above distance one, and that all the impacted elements at distance one lead to code changes.¹¹ If this result were to be confirmed on other case studies and generalized, this would entail important practical consequences. That would imply that, in practice, one does not need to consider impacted elements above distance one to identify changes in the code. However, it is likely that code changes resulting from higher distance impacted elements are still possible, though rare. It is then a matter of cost-benefit analysis to decide about the maximum distance that should be considered for code impact analysis, i.e., checking whether the code needs to be changed.

¹¹ In the case of logical change 11, the iACMTool indicates that no class is impacted at distance 1, but an impacted operation at distance 1 propagates into an impacted class at distance 2. When looking at the source code, the impacted operation at distance one does correspond to a code change, but the impacted class at distance two does not.

¹⁰ We use the semantics of the UML notation to define our rules.

7.3. Cruise control system

7.3.1. System description

The original version of the Cruise Control System is made up of 11 use cases, 37 classes, 86 attributes, and 142 operations. The main functionality of this system is to emulate the cruise control feature of an automobile. The user is able to turn on and off the car, press the gas and break pedals, set a desired cruising speed, accelerate the car to a new speed, turn off cruise control, resume to the desired speed and view the current speed, average trip speed and average trip fuel consumption.

7.3.2. Impact analysis

We performed four, realistic logical changes. Three of the four logical changes are requirement changes and one of the logical changes results from refactorization of a sequence diagram. The first logical change to the system is the addition of an odometer interface. To implement the odometer functionality in the system, the distance travelled is read at timed intervals and displayed on the odometer interface. The second change consists of a more realistic implementation of a gas tank interface. The new gas tank interface has a limited 45L capacity, is able to gauge the level of fuel, emits a warning signal when the fuel is low, and has facilities for fuel refilling. The third change is a reorganization of the Resume sequence diagram. The Cruise use case is removed from the Resume sequence diagram and implemented in a new Cruise sequence diagram. The last logical change implements a speed limiter to the cruise control system. The speed limiter dictates a maximum speed, and if the current speed equals to or exceeds the maximum speed the throttle interface is appropriately adjusted.

Of the 84 model element changes, 40 resulted in detected impacts. Thirty of them show impacts at a distance greater than two. Fig. 11 represents the impacted elements at a given distance, and shows only four curves because these curves are identical for a number of model element changes. The number of actual model element changes that result in the shown distance vs. impacts curves is noted in the legend of Fig. 11. Similar to the

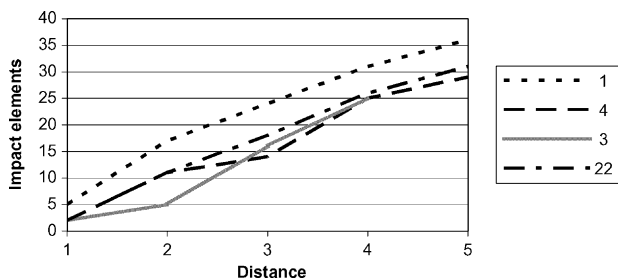


Fig. 11. Cumulative number of impacted elements vs. distance (cruise control).

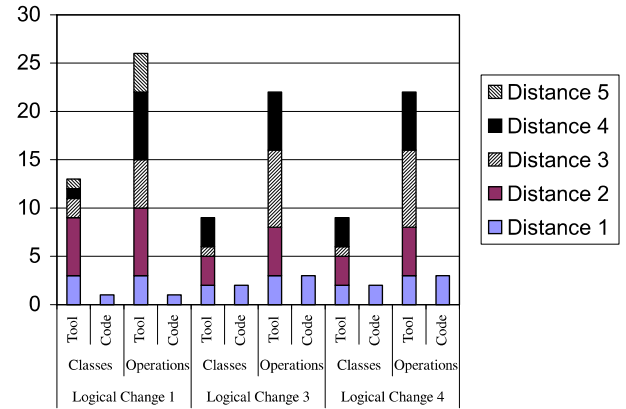


Fig. 12. Code impacts vs. iACMTool impact (cruise control).

ATM system, the most important result to note is that distance/impacts relationships do not look exponential and exhibit a linear trend.

7.3.3. Results accuracy

Using the same format as for the ATM system, Fig. 12 shows, for each logical change, the number of impacted elements at each distance level, for the model and code. We see, once again, that only changes at distance one result in code changes. Our most plausible explanation for this recurring result is that both systems used in our case studies are well-designed. Good object-oriented designs promote encapsulation so as to prevent change propagation. Our results in both case studies show that change propagation is indeed rare.

However, it is very much possible that we would obtain dramatically different results with a poorly designed system. In such cases, changes would propagate through code and greater distances would have to be considered during impact analysis. Then, to avoid combinatorial explosion, the above results showing a linear increase of impacted elements with distance would be very important to ensure and confirm.

However, one difference when compared to the ATM case study can be observed for logical change 1: One of the identified impacted elements at distance one does not result in code change. This is because a change in post-condition does not necessitate a change to the calling operation or class.

8. Conclusions

We present in this article 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 two case studies, 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 in the code, based on the distance between changed model elements and impacted elements. Whether this measure is a good heuristic will have to be further empirically validated but our initial results are encouraging. In our case studies, we investigated whether the impacted elements, as reported by our tool, actually necessitate changes to the implementation. It appears that all the impacted elements at distance one (i.e., impact is a direct result of change), with one exception in the Cruise Control system, necessitate code changes, whereas all the impacts at a distance greater than one do not, thus indicating false-positives. Though this has to be confirmed by additional studies, this suggests that one does not need to consider all the impacted model elements (at various distances) to assess the impact of a logical change on the implementation. We believe that this is due to the good object-oriented design of our case studies, that prevents changes from propagating, but may not be a general trend when the design does not address encapsulation well. Therefore, the focus when considering change impacts in a well-designed system, should mostly be on impacted model elements at distance one.

Though we made a conscious effort to be as exhaustive as possible in terms of rules to identify 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. Both the above types of rules have been defined using OCL on the UML meta-model and we expect that such precise and formal definitions will help refine and evolve our methodology.

Future work includes performing additional case studies. This will also be used in an attempt to associate probabilities with impact analysis rules based on empirical data. This would allow us to further refine our ranking of impacted elements according to their likelihood of actually requiring change.

Acknowledgement

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 on testing object-oriented systems with the UML (<http://www.sce.carleton.ca/Squall/>). 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 also would like to thank Karin Buist for her help in defining and implementing (in the UML diagrams) the logical changes.

References

- Bohner, S.A., 2002. Software change impacts—an evolving perspective. In: *Proceedings of the IEEE International Conference on Software Maintenance*, 3–6 October, pp. 263–272.
- Bohner, S.A., Arnold, R.S., 1996a. An Introduction to Software Change Impact Analysis. In: Bohner, S.A., Arnold, R.S. (Eds.), *Software Change Impact Analysis*. IEEE Computer Society.
- Bohner, S.A., Arnold, R.S., 1996b. *Software Change Impact Analysis*. IEEE Computer Society Press, 0-8186-7384-2.
- Borland, 2003. Together. <<http://www.borland.com/together>>.
- Briand, L.C., Labiche, Y., O'Sullivan, L., 2003. Impact Analysis and Change Management of UML Models. SCE-03-01, Carleton University, <<http://www.sce.carleton.ca/Squall/>>.
- Briand, L.C., Labiche, Y., Soccar, G., 2002a. Automating impact analysis and regression test selection base on UML designs. In: *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, October 3–6, pp. 252–261.
- Briand, L.C., Labiche, Y., Soccar, G., 2002b. Automating impact analysis and regression test selection based on UML designs. Technical Report SCE-02-04, Carleton University, <<http://www.sce.carleton.ca/Squall/>>.
- Briand, L.C., Wust, J., Lounis, H., 1999. Using coupling measurement for impact analysis in object-oriented systems. In: *Proceedings of the IEEE International Conference on Software Maintenance*, September, pp. 475–482.
- Bruegge, B., Dutoit, A.H., 2000. *Object-Oriented Software Engineering-Conquering Complex and Changing Systems*. Prentice Hall, 0-13-489725-0.
- Kung, D., Gao, J., Hsia, P., Wen, F., Toyoshima, Y., Chen, C., 1994. Change impact identification in object oriented software maintenance. In: *Proceedings of the IEEE International Conference on Software Maintenance*, pp. 202–211.
- OMG, 2001. Unified Modeling Language (UML), Object Management Group, <<http://www.omg.org/technology/uml/>>.
- OMG, 2003. OCL 2.0 Specification. Final Adopted Specification ptc/03-10-14, Object Management Group.
- von Kethen, A., Grund, M., 2003. QuaTrace: A tool environment for (semi-)automatic impact analysis based on traces. In: *Proceedings of the International Conference on Software Maintenance*, pp. 246–255.
- Von Mayrhauser, A., Zhang, N., 1999. Automated regression testing using DBT and Sleuth. *Journal of Software Maintenance* 11 (2), 93–116.
- Warmer, J., Kleppe, A., 1999. *The Object Constraint Language*. Addison-Wesley, 0-201-37940-6.

Lionel C. Briand is full professor with the Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, where he founded and leads the Software Quality Engineering Laboratory. He has been granted the Canada Research Chair in Software Quality Engineering and is also a visiting professor at the Simula laboratories, University of Oslo, Norway. Before that he was the software quality engineering department head at the Fraunhofer Institute for Experimental Software Engineering, Germany. Dr. Lionel also worked as a research scientist for the Software Engineering Laboratory, a consortium of the NASA Goddard Space Flight Center, CSC, and the University of Maryland. He has been on the program, steering, or organization committees of many international, IEEE/ACM conferences and is editor-in-chief of *Empirical Software Engineering* (Springer), a member of the editorial board of *Systems and Software Modeling* (Springer) and was on the board of *IEEE Transactions on Software Engineering* from 2000 to 2004. His research interests include: object-oriented analysis and design, inspections and testing in the context of object-oriented development, quality assurance and control, project planning and risk analysis, and technology evaluation.

Yvan Labiche received the BSc in Computer System Engineering, from the graduate school of engineering: CUST (Centre Universitaire des Science et Techniques, Clermont-Ferrand), France. He completed a Master of fundamental computer science and production systems in 1995 (Université Blaise Pascal, Clermont Ferrand, France). While

doing his Ph.D. in Software Engineering, completed in 2000 at LAAS/CNRS in Toulouse, France, Yvan worked with Aerospatiale Matra Airbus (now EADS Airbus) on the definition of testing strategies for safety-critical, on-board software, developed using object-oriented technologies. In January 2001, Dr. Yvan Labiche joined the Department of Systems and Computer Engineering at Carleton University, as an Assistant Professor. His research interests include: object-oriented analysis and design, software testing in the context of object-oriented development, and technology evaluation. He is a member of the IEEE and the ACM.

Leeshawn O'Sullivan received his B.Sc. (Honours) in Electrical and Computer Engineering from the University of the West Indies, 1997. He received his Master of Applied Science in Electrical and Computer Engineering (Software Engineering Major) from Carleton University, 2003. He is currently employed to Direct Energy Business Services (formerly BASE Controls Ltd.) as a Project Engineer. Leeshawn is a licensed Engineer (P.Eng.) in the province of Ontario.

Michael M. Sówka is a M.A.Sc in Software Engineering student at the Software Quality Engineering Laboratory (SQUALL) in Carleton University, Ottawa, Canada. He received his B.Eng. in Systems and Computer Engineering from Carleton in 2003, and has since been studying in the SQUALL lab at Carleton University. His eclectic interests in software systems include software modeling, distributed systems, and pervasive computing.