Arab J Sci Eng (2016) 41:2863–2881
DOI 10.1007/s13369-016-2040-7

RESEARCH ARTICLE - COMPUTER ENGINEERING AND COMPUTER SCIENCE

Managing the Impact of UML Design Changes on Their Consistency and Quality

Dhikra Kchaou[1] · **Nadia Bouassida**[1] · **Hanene Ben-Abdallah**[2]

Received: 28 April 2015 / Accepted: 18 January 2016 / Published online: 4 February 2016

Abstract Depending on the phase in which they come up, changes induce modifications on various software models and may cause model quality deterioration—e.g., incoherence among models, increased complexity. To handle efficiently model changes, every software development project must have a means to manage the impact of every change in one model on the remaining models. Toward this end, we herein present an automated approach that analyzes the effects of changes on the different software models, while considering their impact on the models' quality. Our approach adopts a graph-based traceability technique to identify change impact in terms of the necessary updates the relevant models must undergo to remain coherent, and the effects of the change/updates on the quality of the resulting models. It uses intra and inter UML diagrams' dependencies, best-practice guidelines, and a set of quality metrics. Evaluated quantitatively on three cases, our approach showed an average precision of 0.88 and recall of 0.95 in identifying the impacts of different types of changes.

Keywords UML diagrams · Change impact · Dependency graph · Consistency rules · Quality rules · Object-oriented metrics

✉ Dhikra Kchaou
Dhikra.Kchaou@fsegs.rnu.tn

Nadia Bouassida
Nadia.Bouassida@isimsf.rnu.tn

Hanene Ben-Abdallah
HBenAbdallah@kau.edu.sa

1 Mir@cl Laboratory Sfax University, Sfax, Tunisia

2 King Abdulaziz University, Jeddah, KSA

1 Introduction

Computerized systems are inevitably subject to continuous evolution to account for the emergence of new requirements, the improvement of existing features, the reparation of errors, etc. Among the predicaments induced by the complexity of today's software are the difficulty and high expenses of their adaptation to changes imposed by their evolution. To face these predicaments, change impact analysis techniques are necessary in order to identify the potential consequences of a change in terms of efforts to accommodate it [1].

The software engineering literature offers two categories of change impact analysis (CIA) approaches. The first category of approaches, called *horizontal* or *dependency-based* CIA, tackles the problem within a particular development phase, often the design phase (e.g., [1–3]) or the code phase (e.g., [4–6]). The second category of approaches, called *vertical* or *traceability-based* CIA, focuses on changes at one level of abstraction and their corresponding impacts on another level, e.g., between the code and the design (e.g., [7]), or between the requirements and the design (e.g., [8]).

Both categories of change impact analysis approaches are necessary and complementary; obviously, it is important to ensure that models at the same level of abstraction are consistent before examining the change impact on models at other levels of abstraction. In particular, in model-driven development (MDD), both categories of CIA are imperative to the success of the software development project. Indeed, MDD relies on models as main artifacts in the software lifecycle and model transformations as a means to derive various models at different levels of abstraction (e.g., to refine a design, to produce the code). As such, any change in one model must be propagated in a coherent way to all related models at the same level of abstraction (i.e., horizontally) and at different levels of abstraction (i.e., vertically). The coherent propagation of changes can benefit from model transformation rules if these latter provide for traceability among models' elements [9]; however, not all transformations satisfy the traceability property, consequently any model change would require the re-application of the transformations on the modified model to derive the remaining models (design and/or code). Overall, face to model changes, existing works in the MDD context propose tools either to verify the consistency of evolved models (e.g., we refer to the survey in [10]), to trace meta-model change impact [9], or to identify change impact on model transformations [11]. In particular, none of the existing works offers an automated approach: (i) to detect and resolve the inconsistencies induced by the changes within the modified model and *all* inter-related models across all levels of abstraction; and (ii) to assess the impact of a change on all models' quality. These shortages are the main contribution of this paper.

More specifically, we herein propose an automated approach that analyzes the effects of changes on the design models, while considering their impact on the models' quality. Our approach deals with software designed with UML which offers multiple views of a software system through a set of diagrams. Because of the syntactic and semantic dependencies among the UML diagrams, changes in one diagram often lead to changes in other diagrams modeling the same system. Our approach for change impact analysis which is at the design level becomes of greater interest when integrated into a development process such as RUP [12] or UP [13] where changes occur from one iteration to another. In fact, change propagation in a consistent manner is vital to the success of such iterative software development process. It requires a change impact analysis method that accounts for the diagrams' syntactic and semantic dependencies. In this context, existing works rely on transformations of UML diagrams into graphs [6], grammars [14] or OCL constraints [2]. Our herein proposed method has the merit of using a dependency graph to identify inter- and intra-diagram inconsistencies between the different UML diagrams.

Besides the consistency of the various UML diagrams after a change occurs, the success of the software development also depends on the quality of the produced models. Often, changes in one UML diagram influence the quality of both the changed diagram itself as well as the related diagrams. For instance, the deletion of a class with a high coupling degree may result in a duplicate distribution of its attributes and methods, which degrades the quality of the class diagram in terms of the metric Coupling Between Objects [15]. This deletion may also affect the quality of sequence diagrams using objects of the deleted class, which may increase the software maintenance: the redistribution of the methods of the deleted class will induce several changes in all the sequence diagrams using it to distribute the methods' calls possibly with redundancies that will complicate the maintenance.

To identify change impacts on the quality of UML designs, our change impact analysis method exploits the syntactic and semantic dependencies among UML diagrams and exploits two sources of model quality analysis: design metrics ([15, 16]), and "best practices" [17]. Besides accounting for both inter-diagram consistencies and model quality, our method is the first to adopt a metric-based approach to change impact management.

Informally, the proposed change impact analysis method operates in three main steps. First, it identifies intra- and inter-diagram inconsistencies based on a set of consistency rules in order to assist the designer in correcting any inconsistency. Secondly, it measures the effects of changes based on a set of metrics in order to estimate the effort needed to accommodate each change. Finally, it produces recommendations in order to preserve the initial consistency and quality of the diagrams. To conduct these three steps, our method integrates the different UML diagrams along with their dependencies into a single graph, we call *model dependency graph*. By explicitly encoding the intra- and inter-diagram dependencies, this graph provides for the needed traceability to analyze the impact of a change on both the consistency and quality of the diagrams.

The remainder of this paper is organized as follows: Sect. 2 overviews existing approaches to change impact management in UML diagrams. Section 3 defines a set of consistency and metric-based quality rules used in our change impact analysis approach. Section 4 presents our approach for the class and sequence diagrams. Section 5 illustrates the method through an example. Section 6 discusses the results of an experimental investigation of the performance of the proposed approach. Finally, Sect. 7 concludes with a summary of the presented work and a highlight of its extensions.

2 Existing Approaches to Change Impact Analysis

Several methods were proposed to cope with change impact analysis (CIA) in UML models. They can be classified into two categories of approaches: horizontal/dependency-based CIA, and vertical/traceability-based CIA.

2.1 Horizontal CIA Methods

Horizontal CIA methods examine the impact of changes in software artefacts at the same level of abstraction, mainly at the code or the design levels.

2.1.1 CIA at the Code Level

One essential motivation behind CIA methods operating at the code level is that source code modifications during software evolution cannot be avoided. The survey of code-level, horizontal CIA methods presented by Li et al. [18] classifies them into four categories:

1. Traditional static program analysis techniques using a dependency graph [5,19,20]: The dependency graph is used essentially in a reachability analysis to determine the change impacts. For instance, [5] presents a program as a directed graph where the set of nodes represent components of the program (classes, methods and fields) and the edges represent references between these components.
2. Software repository mining techniques which apply data mining techniques on software repositories in order to identify co-change coupling information [3]. This information is then used to determine those files (or program entities) that are changed together in the software repository. Thanks to this information, when a change occurs in one file, both the list of all affected files is automatically determined and suggestions to consider further changes are inferred from the version history.
3. Coupling measurement which is based on calculating some coupling metrics such as structural coupling, conceptual coupling, dynamic function coupling and relational topical based coupling (e.g., [4,21]). The coupling impact sets are predicted by analyzing the program elements that are coupled with the changes. For instance, [21] introduce the measure of Dynamic Function Coupling (DFC) between two functions or methods to compute the impact set at the function level.
4. Execution information collection which analyses information collected during the program execution like execution traces information, coverage information and execution relation information (e.g., [22]). The collected runtime information is used to compute the impact set: the set of program entities that may be affected by the changes for a specific set of program executions.

2.1.2 CIA at the Design Level

Horizontal CIA methods in UML designs can be classified into graph-based, rule-based or grammar-based techniques. Among the works based on a graph technique, the method in [6] uses a formal approach to treat consistency analysis between the class and sequence diagrams. This method translates the class diagram into an attributed, typed graph and the sequence diagrams are converted into graph grammars. Its consistency analysis focuses only on existence, visibility and multiplicity checking. Existence checking verifies if all model elements used in the sequence diagram exist in the class diagram and if, for each link between a sender and a receiver object, there is a corresponding association in the class diagram. Visibility checking requires that the classes, attributes, operations and references are visible. Finally, multiplicity checking verifies that the multiplicities defined in the class diagram are respected when the designer changes the sequence diagram. However, this method did not check that a message (in the sequence diagram) must be defined as an operation in the receiver's class. In addition, it did not examine the changed model quality.

The second category of horizontal methods adopts a rule-based technique. Here, the dependencies among UML diagrams are expressed as a set of rules. These latter can be used both to verify the consistency of a modified diagram, and to identify how a change in one diagram impacts others. For example, the works in [2,23] propose 120 rules identified from the UML meta-model in order to verify the consistency of the changed model. In addition, they classify possible changes of the class, sequence and statechart diagrams into 97 change categories. To determine model elements that are directly or indirectly impacted by those changes, they use OCL impact analysis rules that are defined for each change category. Compared to the other techniques, the rule-based technique has the advantage of measuring the distance between impacted elements to indicate what model elements should be checked first in order to prioritize the results of the impact analysis. However, like the graph-based technique, this technique does not evaluate the changed model quality.

Also adopting a rule-based technique, the authors in [1] define four relationships among model elements from the UML meta-model: association, two relationships for composition (part and composite) and the relationship between an element and its attributes. In addition, they propose three steps to analyze the impact of seven change types: (1) extracting changes that present how to deal with composite changes; (2) checking the meta-model relationships for each change type and for each impact analysis rule; finally, (3) identifying the impacted elements based on the UML meta-model. This approach [1] has the advantage of reducing the number of rules; however, similar to the others, it lacks the quality analysis.

Among the grammar-based approaches, the authors in [14] propose a framework for automatic transformation of UML diagrams into a string and then compilation of that string for syntactic correctness and inter-diagram consistency verification. The authors propose a formal context free grammar for the class and sequence diagrams. Based on this grammar, they formalized a set of intra-diagram rules and two inter-diagram rules to identify the impact of a change.

The grammar-based work proposed in [24] is based on the model profiling techniques to identify the scope of elements affected directly or indirectly by a change. It uses the UML/Analyzer tool [25] to identify inconsistencies caused by changes and let the designer decide whether the effects of a change are desirable or not. The tool presents choices to correct the detected inconsistencies and the designer must validate one of the choices. The number of choices as well as the choices themselves may be ambiguous for the designer especially with large diagrams.

Also adopting the horizontal approach, the authors in [26] propose a change propagation process through the class, sequence and statechart diagrams. After editing one of these diagrams, the change propagation process checks firstly whether the UML consistency constraints are affected. Secondly, a library of repair plan types is used to generate plan instances and their costs for the violated constraints. The cheapest repair plan instances are presented to the designer whose selected repair plan instance is executed, updating the design model. This work is improved by Dam et al. [27] who formulated the change propagation process as a classical state space search where each state represents a snapshot of the model. A state in which all dependencies in the model are consistent is a "consistent" state, or else it is called an "inconsistent" state. The initial state represents the model after being modified by primary changes. In the goal state, all dependencies in the model are consistent. If the initial state is a consistent state, then it is also the goal state, and thus no further changes are needed to the model. However, if the initial state is inconsistent, the search continues to find a path from the initial state to a goal state, i.e., a solution path. Since there can be many solution paths, the authors try to find the shortest paths where the distance between nodes (i.e., states) represents how a version of the model (represented by a node) deviates from another version (represented by another node).

2.2 Vertical CIA Methods

Vertical or traceability-based methods are interested in managing the impact of changes between models at different levels of abstraction, produced during the different development phases. Dealing with CIA between requirements and design artefacts, Knethen et al. [8] analyze relationships based on a conceptual trace model for a specific product model in the domain of embedded systems. To manage the impact of a change, they derive analysis guidelines from the conceptual trace model.

Hammad et al. [7] proposed an approach that supports traceability from source code to the design. This work first generates differences between two files. Secondly, changes that impact the design are identified from the changed code. (This work deals with ten code changes: added/removed classes, added/removed methods, and changes in relationships: added/removed generalizations, associations, and dependencies, respectively.) Thirdly, the identified design changes are reported to the designer so that he/she updates the design to keep it consistent with the code.

Shiri et al. [28] proposed a method for CIA between requirements and test cases. This approach supports the modification analysis at the requirements level by identifying potential change impacts and retesting effort associated with a modification. Their contribution consists in collecting dynamic information from Use Case Maps in order to apply formal concept analysis [29] on these collected traces to identify the changed requirements and execution dependencies.

Overall, existing works treat the problem of CIA based on consistency rules and they propose a set of impact rules for each type of change. However, the proposed rules do not include the change impact on the quality of related diagrams. In addition, they do not make use of by "best practices" and design metrics in their change impact analysis and management approaches. Furthermore, the change impact correction and its visualization is not treated. The automatic correction and visualization allows the minimization of the designer effort. These shortages motivated our work on proposing a new approach that analyzes and manages the change impact through a hybrid technique combining graph-based, rule-based and metric-based techniques (see Sect. 3).

While CIA can be used in various development phases, we believe that an early CIA approach can provide for a means to estimate both the effort required to handle a change and its impacts on the quality of the various models; this estimate can be used to decide whether to undertake the change or to cancel it. In addition, assisted by a change impact analysis at an early development phase, the designer would have an important support in producing a good quality design which is an essential determinant in the success of the software project. Given these benefits of an early CIA, we focus in this paper in proposing an approach for change impact analysis at the design level and more specifically for software designed with UML.

3 Our Change Impact Management Method

Our method provides for the identification and measurement of potential side effects induced by changes between the different UML diagrams produced at the design level. It has two main originalities. Its first originality is that it hybridizes several techniques: graph-based, rule-based, and metric-based techniques. The graph is used to identify impacted elements by coding both syntactic and semantic dependencies among UML diagrams' elements; the rules are used to express the consistency constraints among the intra and inter UML diagram elements; and the metrics are used to estimate the change impacts in terms of quality effects and effort needed to carry out all changes necessary to preserve the consistency of the design. These techniques allow our method to collect information needed to assist the designer making a judicious decision about accepting or refusing a change based on its impact on the effort needed to accommodate it and the resulting model quality.

The second originality of our method is that, besides collecting the essential consistency rules for the UML language, it proposes a set of quality rules based on a set of metrics and best practices with UML.

We next present a set of metrics useful in measuring the impact of a change. Afterward, we present our consistency and quality rules for CIA.

3.1 Measuring the Change Impact

We propose to calculate the number of required modifications (NRM) to correct inconsistencies caused by an intra/inter-diagram change. NRM calculates the number of update/change operations needed to correct a violated consistency rule.

To calculate NRM, we propose the new intra/inter-diagram metrics shown in Table 1. For a given change, NRM is the sum of the metrics of the intra-/inter-diagrams which are concerned by the change. The proposed thresholds for these metrics will be defined using an empirical study in a future work.

3.2 Rules for Consistency-Based CIA

We distinguish between two kinds of model consistencies: Intra and Inter-model. Intra-model consistency is the consistency of a model in itself as defined by the modeling language. Once a change is introduced in one diagram, the various elements of this latter must remain consistent; that is, they must respect all syntactic and semantic consistency rules. Similarly, once a diagram undergoes a change, it must remain consistent with all other diagrams to which it is syntactically and semantically related. Inter-model consistency is the consistency of several models with respect to one another [30].

A wide classification of consistency rules for the class and sequence diagrams are presented in the literature (cf., [31–33]), e.g., syntactic vs. semantic, static vs. dynamic, intra-model vs. inter-model. Table 2 lists essential intra-diagram consistency rules for the class diagram, and inter-diagram rules between the class and sequence diagrams.

Intra-diagram consistency rules can be determined based on the information contained in the class diagram, except for the consistency rules CR2 and CR4. For these two rules, we suppose that the designer adds notes to the class diagram to indicate operation calls and to indicate also attributes used as parameters in an operation. The semantics of these UML notes obey to certain rules. As an example, they start with the word "uses" or "calls" to indicate, respectively, that a method may use one or more class attributes or to indicate that a method may contain calls to one or more class methods [34].

Table 1 Metrics used to measure the intra/inter-diagram quality

Metrics	Definitions
Intra-diagram metrics	
NAt/Op	Number of times an attribute (and operation) of a class is used (called) in an operation
NAtPr	Number of times an attribute is used as a parameter in an operation
NM2Ob	Number of messages between two objects
Inter-diagram metrics	
NCSD	Number of times a class is used as an object in SDs
NATSD	Number of times an attribute is used in SDs
NOpSD	Number of times an operation is used as a message in SDs

Table 2 Consistency rules between the class and sequence diagrams

Consistency rules CR	
Intra-class diagram rules	Class and sequence inter-diagram rules
CR1 The name of classes, attributes and methods in the same class diagram must be different	**CR8** Each object in SD must be an instantiation of a class in CD
CR2 The class operation may use one or more class attributes	**CR9** Each operation in SD must be defined in the receiver's class in CD
CR3 The descendent classes inherit all properties of the parent class	**CR10** For each message between S and R objects, there is a corresponding association in the CD
CR4 The class operation may contain calls to one or more class operations	**CR11** Classes, attributes, operations and references used in SD must have a public visibility in a CD
CR5 The class operation may use an attribute as a parameter	**CR12** The creation and the deletion of objects in SD must respect multiplicities in CD
CR6 Inheritance cycles are not allowed	**CR13** A combined fragment in SD may use an attribute
CR7 The multiplicity at composite association end is only allowed to be 1	

3.3 Rules for Quality-Based CIA

A change may affect not only the consistency of UML diagrams, but also their quality. Thus, in addition to the preservation of the consistency of UML diagrams, their quality must be evaluated and preserved after a change is introduced. For instance, the deletion of a class that has many important relationships with other classes or that participates in a design pattern [35] depreciates the quality of the class diagram.

The quality of software can be evaluated using several metrics (e.g., [15]). Choosing useful metrics is not sufficient to ensure efficient quality evaluation after a change; it is necessary to determine the threshold values, which highly influence the efficiency of the quality verification process. We should caution that, even in the software engineering field, several works tried to fix thresholds for object-oriented design metrics (e.g., [36–39]). For instance, [37] establishes three threshold ranks: (i) *good* which refers to most common values; (ii) *regular* which refers to values with low frequency, but that are not irrelevant; and (iii) *bad* which refers to values with rare occurrences. Existing works agree on some threshold values, for example they state that the lack of cohesion in methods (LCOM) metric has to be inferior to 1 to ensure a good quality. In our approach, we have adopted the thresholds proposed by Tarcísio et al. [39] because they have been adopted by many researchers. Determining appropriate thresholds empirically is our ongoing work.

Table 3 CK Metrics [14] used to measure the intra-diagram quality

Metrics [14]	Definition	Thresholds [39]
Cohesion		
LCOM "Lack Of cohesion in methods"	The number of method pairs without shared instance variables, minus the number of method pairs with shared instance variables	$T \leq 0,725$
Complexity		
WMC "Weighted methods per class"	A weighted sum of all the methods defined in a class	$T \leq 34$
Inheritance		
DIT "Depth of Inheritance"	The length of the longest path from a given class to the root class in the inheritance hierarchy	$T \leq 4$
NOC "Number Of Children"	The number of immediate child classes that inherit from a given class	$T \leq 28$
Coupling		
CBO "Coupling Between Objects"	The number of other classes to which a given class is coupled	$T \leq 16$
RFC "Response For Call"	The number of methods that can be potentially invoked in response to a message received by an object of a particular class	$T \leq 35$

Table 4 Quality rules

Quality rules (QR)	
Intra-class diagram rules	Class and sequence Inter-diagram rules
QR1 the threshold of CK metrics [15] presented in Table 3 should be respected	**QR4** Each class should be instantiated as an object in at least one of SD
QR2 A class participating in a design pattern should not be deleted	**QR5** Each association should induce an interaction in at least one SD
QR3 An isolated class should interact with related classes	**QR6** Each operation should be a message exchanged at least once in SD

For the analysis of change impact on quality, we propose the set of quality rules QR enumerated in Table 4. Our set of quality rules include metric-based rules (QR1) and best practices based rules (QR2, QR3, QR4, QR5 and QR6). Our metric-based QR use metrics from the CK metrics suite [15] with their thresholds [39] as presented in Table 3. We calculate these metrics before and after every change, and based on the thresholds, we verify the quality rules listed in Table 4 and we inform the designer about any violation.

Our best practices QR use good modeling practices, for instance, keeping elements of a design pattern after a change; in these rules, we use our design pattern identification approach [40] which indicates which classes participate in the patterns.

It is worth noting that the violation of the quality rules does not make a diagram erroneous; however, it does not guarantee the proper functioning of a system either.

4 CIA Between the Class and Sequence Diagrams

Similar to any change impact analysis technique (e.g., [1,41]), our method proceeds in three steps (Fig. 1): (1) identification of the change; (2) definition of the relationships between artifacts or definition of traceability between the elements of the different models; and (3) execution of the impact consequences (corrections and/or recommendations). We next detail these steps.

4.1 Change Identification

A change impact analysis technique needs to specify how changes are defined. Thus, in a previous work [42], we defined a MOF [43] based change meta-model that covers all change types at a high level of abstraction. Being MOF-based, our change meta-model defines all possible changes

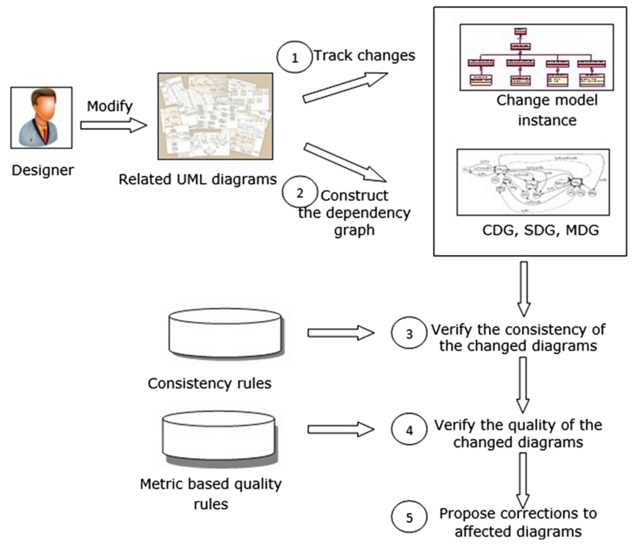

Track changes
Modify
Designer
Related UML diagrams
Change model instance
Construct the dependency graph
CDG, SDG, MDG
Consistency rules
Verify the consistency of the changed diagrams
Metric based quality rules
Verify the quality of the changed diagrams
Propose corrections to affected diagrams

Fig. 1 Steps to manage change impacts on UML diagrams

affecting the elements independently of a particular modeling language. In addition, it can be extended and adapted to present changes that affect any MOF-based model and, in particular, UML models.

4.2 Traceability Between the Class and Sequence Diagrams: MDG Construction

Based on the fact that UML diagrams can be assimilated to graphs, the dependencies between UML diagram elements could therefore be determined using a graph reachability analysis technique. Indeed, inspired from the work of Lallchandani et al. [34] for static slicing of UML models, we defined a method to construct the model dependency graph (MDG) which we use in the third step of our CIA method (Fig. 1).

More specifically, to model all dependencies, the UML class diagram is transformed into a class dependency graph (CDG) and every UML sequence diagram is transformed into a Sequence Dependency Graph (SDG). Afterward, the CDG and set of SDGs are merged in order to construct the Model Dependency Graph (MDG).

Our CDG and SDG construction method adapts the transformations initially proposed by Lallchandani and Mall [34]. Moreover, our adaptation accounts for the association relationships (not treated by Lallchandani and Mall [34]) because they have an effect on method calls in the sequence diagram, hence they should be represented in the CDG. In addition, these relationships are represented in the CDG, in order to check the consistency rules CR10, CR11 and CR12. Furthermore, multiplicities and visibilities are added to the CDG.

To transform a class diagram (CD) into a CDG, each class in CD is transformed into a class (square) node. Each method, attribute is transformed into a circle node. Each method parameter and return value in CD is transformed into a dotted circle node. A class node representing a class c is connected to every method node representing the methods of c, and to every attribute node representing the attributes of c via member dependence edges. In addition, each method node is connected by method dependence edges to parameter and return nodes representing the parameters and return values of the method. Also, a method node may be connected to an attribute node by an attribute dependence edge.

To transform a sequence diagram (SD) into a SDG, each object in SD is transformed into a class (square) node. If a combined fragment has a condition associated with it in the lifeline of an object of a class, this class is transformed into a (rounded square) predicate class node. Each reference to another sequence diagram is transformed to an interaction node. The kth message dependence edge between these nodes are represented by $I(k)$. A call dependence edge represents a transfer of a message from an object of one class to an object of another class. A return dependence edge represents a message transfer representing a value return of a method invocation by an object. Moreover, for each predicate class node, all the messages in the combined fragment form a message dependency, and they are represented as a control dependence edges. However, to connect an interaction node and other SDG nodes, inter-sequence call dependence edges and inter-sequence return dependence edges are used.

To trace the change impact across a class and sequence diagrams, the CDG and SDGs must be integrated into a single graph. The combined graph (MDG) takes all the nodes of the CDG and SDGs. In addition, the MDG includes all the member, method and data dependencies from the CDG, and the control and inter-sequence dependencies from the SDG. However, each call dependence edge between two class nodes (CNSource, CNTarget) in an SDG is altered and represented between the class node “CNSource” and the method node of the class “CNTarget” in the MDG. Furthermore, a return dependence edge between two class nodes (CNSource, CNTarget) in an SDG is altered and represented between “CNSource” and a return value node of the class node “CNTarget” in the MDG. Finally, the graph integration process modifies each relationship dependence edge between two class nodes (CNSource, CNTarget) in a CDG into a call dependence edge in the MDG.

4.3 Corrections and Recommendations

Our approach indicates, for every change type, the violated rules as well as the changes needed to correct the corresponding diagrams. In the following subsection, we show how the MDG can be used to indicate for each change type applicable to a class diagram CD, the potentially violated rules in the class diagram itself (intra-diagram analysis) as well as the impact of this change on the sequence diagrams SD (inter-diagram analysis). In addition to rules identification, we show how our method can assist the designer by presenting appropriate corrections and corresponding effort estimates.

4.3.1 Changes Applicable to Classes

Our change meta-model identifies ten types of change operations relevant to classes: addition, deletion, move and update of a class, addition, deletion and update of an attribute/operation.
Add a class:

Intra-diagram CIA:

The addition of a class may violate the rules QR3 and CR1. For QR3, the CDG is examined to check the existence of a relationship dependence edge from/to the added class: if no such edge exists, then the added class is isolated. The corrective recommendation for this inconsistency would ask the designer to add relationships between the added class and other classes in the class diagram.

The violation of the consistency rule CR1 is detected if the name of the added class already exists in the CDG. Thus, the corrective proposition for this inconsistency consists in allowing the designer to decide between updating the new name of the added class and undoing the new change.

Inter-diagram CIA:

The violation of QR4 is detected if the added class does not exist in the MDG (i.e., the added class does not participate in an SD) and association relationships exist in CDG between the added class and other classes which participate in an SDG. Thus, to improve the design, the designer is advised that the added class should be used as an object in at least one SD.

Estimation of CIA Effort:

To estimate the effort needed for change impact management when a class is added and CR1 is violated, the NRM is calculated as follows:

$$\mathrm{NRM} = 1 + \mathrm{NCSD}$$

That is, NRM equals one update operation for the added class plus the number of updates for this class in all the SDs. Note that, the corrections needed to respect CR1 consist in renaming which can be done automatically. The NCSD metric gives an idea about the impacts on SDs: The larger NCSD is, the higher the number of impacts of the class addition on SDs is, and therefore we indicate to the designer that maintenance becomes more difficult and testing becomes more complex.
Delete a class:

Intra-diagram CIA:

The deletion of a class can cause the violation of the consistency rules CR2, CR3 and the quality rules QR1 and QR2.

The violation of CR2 is detected when MDG contains a data dependence edge between an operation and an attribute or a method of the deleted class. To correct the inconsistency caused by the violation of CR2, we propose to the designer to update the operation which uses/calls the attribute/operation of the deleted class or to undo the change. The violation of CR3 is detected if there exists a generalization dependence edge between the deleted class and inheriting classes. Its correction needs an update of the descendant classes.

The violation of QR1 is detected when the CBO (coupling between objects) of the deleted class is high. In this case, we indicate to the designer that the deletion of a class with a large coupling increases the sensitivity to changes in other parts of the design and therefore maintenance could become more difficult and testing would need to be more rigorous [15]. The designer has to decide to undo the change or to ignore our recommendation.

Violation of the rule QR2 is detected when the deleted class participates in a pattern, therefore, we indicate to the designer that this class should not be deleted. For this purpose, we use our design pattern identification approach [40], which indicates which classes participate in the patterns.

Inter-diagram CIA:

The deletion of a class which is instantiated as an object in SD presents a violation of the CR8. This inconsistency is detected if the deleted class exists in the MDG. The correction of this inconsistency consists in deleting objects in SDs corresponding to the deleted class as well as deleting all messages from-to this object.

Estimation of CIA Effort:

When CR2 is violated, the estimation of correction efforts (NRM) is calculated as follows:

$$\mathrm{NRM} = \mathrm{NAt/Op}$$

That is, NRM equals to the number of updates needed for the operations which use/call the attributes or operations of the deleted class.

The correction of CR2 cannot be done at the design level and needs the designer intervention. However, we can inform the designer about the number of required updates.

When CR3 is violated, the NRM metric is equal to the number of updates needed for the descendants of the deleted class. That is,

$$\mathrm{NRM} = \mathrm{NOC}$$

When CR8 is violated, NRM is equal to the number of times the deleted class is used as an object in SDs:

$$\mathrm{NRM} = \mathrm{NCSD}$$

A big NCSD value implies difficulty of change impact identification and their corrections, and therefore we indicate to the designer that maintenance could become more difficult and testing would be more complex.
Move a class:

Intra-diagram CIA:

Moving a class must respect the consistency rule CR3.

A CR3 violation is detected if the moved class is a super class and a generalization dependence edge exists from child classes to the moved class in CDG. The corrections consist in transferring the old/new descendants under the moved class.

Inter-diagram CIA:

Moving a class may cause the violation of CR10. In fact, moving a class causes the loss of association relationships of this class.

The CR10 violation is detected if a call dependence edge exists from Class Node source to the method of the moved class and an association dependence edge did not exist between these classes. The proposed correction consists in deleting messages from-to the object corresponding to the moved class and unrelated object.

Estimation of CIA Effort:

When rule CR3 is violated, the NRM metric equals one update for each old/new descendent class of the moved class:

$$\text{NRM} = \text{NOC}.$$

This modification can be done automatically.

When rule CR10 is violated, NRM equals the number of deleted messages from-to the object corresponding to the moved class. It equals the number of messages between the objects corresponding to the moved class and unrelated objects (corresponding to unrelated classes with the moved class):

$$\text{NRM} = \text{NM2Ob}$$

This correction can be done automatically. However, the metric NM2Ob gives an idea about the amount of information that must be deleted in an SD.

Add attribute:

Intra-diagram CIA:

The addition of an attribute must respect the consistency rule CR1. The analysis of this type of change is the same as for the "add class" change.

Estimation of CIA Effort:

To assist the designer in his choice, we use the NRM. In this case, NRM is equal to one update operation for the added attribute plus the number of updates needed for this attribute in all the SDs:

$$\text{NRM} = 1 + \text{NATSD}$$

Note that the corrections needed to respect CR1 consists of renaming which can be done automatically. However, a large NATSD value increases the number of corrections needed.

Delete attribute:

Intra-diagram CIA:

An attribute deletion may violate the consistency rules CR2 and CR5.

The violation of rule CR2 is analyzed in a similar way to "delete class". It is detected when a data dependence edge exists from an operation to the deleted attribute. As for the rule CR5, it can be detected if the deleted attribute is used as a parameter node in CDG. To correct these inconsistencies, the designer should choose to update the operation that uses the deleted attribute or to undo the change.

Inter-diagram CIA:

An attribute deletion may affect a SD if the deleted attribute is used in a combined fragment in MDG in which case the rule CR13 is detected. The designer has to update or delete the combined fragment that uses the deleted attribute or undo the change.

Estimation of CIA Effort:

When rule CR2 is violated, the NRM is equal to the update of the operations that use the deleted attribute.

$$\text{NRM} = \text{NAt/Op}$$

After the violation of rule CR5, NRM is equal to one update of the operation that uses the deleted attribute as a parameter:

$$\text{NRM} = \text{NAtPr}$$

The correction of rules CR2 and CR5 cannot be done at the design level and requires the designer's intervention. However, the NRM gives an idea about the number of times the designer must update.

To correct rule CR13, the NRM is calculated as follows: an update or a deletion of the combined fragment that uses the deleted attribute:

$$\text{NRM} = \text{NATSD}$$

The choice between the update and the deletion of the combined fragment using the deleted attribute needs the intervention of the designer. This choice can be guided by the NATSD since the larger this number, the higher the impacts on SDs and therefore a large number of update needs more effort than a large number of delete.

Delete operation:

Intra-diagram CIA:

The deletion of an operation may violate CR4 if a data dependence edge exists from an operation to the deleted operation. To correct this inconsistency, the designer must update the operation that calls the deleted operation or to undo the change.

Moreover, the deletion of an operation in a class may lead to a lack of cohesion in methods (LCOM) of that class which increases complexity (QR1).

Inter-diagram CIA:

The deletion of an operation may affect the SD and violate the CR9 if a call dependence edge exists from a class node to the deleted operation. To correct this inconsistency, we must delete the messages in SD corresponding to the deleted operation in CD.

Estimation of CIA Effort:

The consistency rule CR4 cannot be corrected at the design level and needs the designer intervention. The NRM metric is equal to the number of updates of the operations that call the deleted operation:

$$\text{NRM} = \text{NAt/Op}$$

The NRM gives an idea about the number of times the designer has to update. However, the consistency rule CR9 can be determined and corrected automatically. The NRM metric is equal to the number of deletion of the messages corresponding to the deleted operation:

$$\text{NRM} = \text{NOpSD}$$

The NOpSD metric gives an idea about the number of impacted elements after an operation is deleted from SDs: the designer is advised that a high NOpSD, means a high impact on SDs, which also increases the maintenance difficulty and testing complexity.

4.3.2 Changes Applicable to Relations

Add a relation:

Intra-diagram CIA:

The addition of a relationship may violate the consistency rule CR6 if the new relationship is a generalization that forms a cycle in the MDG. In addition, adding a relationship may violate the consistency rule CR7 if the new relationship is a composition dependence edge and the multiplicity is different from 1.

To correct the violation of rule CR6 or CR7, the added relationship must be either deleted or its type must be updated: in the case of addition of a generalization that forms a cycle, we recommend that the designer changes the type of the relationship or to delete another generalization to eliminate the cycle.

Besides the consistency of the class diagram, the addition of a generalization relationship may affect its quality: it may lead to deeper inheritance trees which increase the design complexity. To detect this risk, our method measures the resulting DIT metric [15] and, depending on its value, an appropriate recommendation (QR1) is given to the designer: a high DIT resulting from the added generalization increases the complexity of the design (in terms of comprehension, testing, and maintenance, etc.) because it increases the interaction possibilities between the classes. Moreover, the addition of an association may increase the coupling of related classes. When the number of couplings is large, maintenance becomes more difficult.

Inter-diagram CIA:

An added association relationship should induce additional interactions in the SDs (QR5).

The correction for rules CR6 and CR7 requires the intervention of the designer to choose between a deletion or an update of the added relationship.

Delete a relationship:

Inter-diagram CIA:

The deletion of an association relationship may lead to an inconsistency in the sequence diagram (violation of rule CR10) if a call dependence edge exists from a class node source, which represents the end 1 of the deleted association, to the method of the target class Node which represents the method of the deleted association (end 2). We inform the designer about this violation and let him decide between undoing of this change and deleting the corresponding messages in the SD.

In addition, the deletion of an association relationship may lead to an isolated class, thus, a violation of the recommendation rule QR3 can be detected.

Estimation of CIA Effort:

The detection and correction of CR10 violation can be done automatically without the designer intervention. The designer can be guided in his correction choice through the NRM which gives an idea about the number of deletions: NRM is equal to the number of deleted messages between objects corresponding to classes related by the deleted association; in other words, it is equal to the number of messages between the two objects corresponding to the classes related by the deleted association. Hence,

$$\text{NRM} = \text{NM2Ob}$$

4.3.3 Changes Applicable to Objects

Add object (lifeline):

The addition of an object in a sequence diagram may violate the rules CR8, CR11 and CR12. The violation of CR8 means that the designer added an object which is not an instantiation of a class in the class diagram and is detected if the added object does not exist as a class node in CDGs.

In addition, the violation of CR11 and CR12 indicates that the multiplicity and the visibility of the added object prevent the addition of this object in the sequence diagram.

The rule CR11 is detected if the visibility of the class (in the CDGs) corresponding to the added object is not public.

The rule CR12 is detected if the multiplicity of the class corresponding to the added object prevents its use in the sequence diagram. In these cases, we inform the designer about the violated consistency rules and we propose to delete or to update the added object.

Estimation of CIA Effort:

The correction of inconsistencies caused by the violation of CR8, CR11 and CR12 needs the designer decision to choose between the deletion and update of the added object.

Delete object (lifeline):

The deletion of an object may violate the CR12 if the multiplicity of the class corresponding to the deleted object in CDGs is not respected. The designer must undo this change.

Estimation of CIA Effort:

NRM = undo the deletion of the object = 1.

This correction which is caused by the violation of CR12 is automatic.

Add message:

The addition of a message in a sequence diagram must respect the rules CR9 and CR10.

The violation of CR9 means that the added message does not have a corresponding operation in the class diagram and is detected if the added message does not exist in the methods of the target class node. This addition is not allowed and the designer had to undo this change or to add an operation corresponding to the added message in the CD.

Moreover, the addition of a message between two objects which are not related by a corresponding association in CD leads to an inconsistency if an association relationship does not exist from the class node source of the deleted message to the class node target of the deleted message (rule CR10). So, the designer is advised to either undo this change or add an association between these two objects.

Estimation of CIA Effort:

The NRM after the violation of CR9 is equal to an addition of an operation in CD corresponding to the added message.

After the violation of CR10, NRM is equal to an addition of an association in CD. The addition of an operation in CD corresponding to the added message can be done automatically as well as the violation of rule CR10 can be corrected automatically.

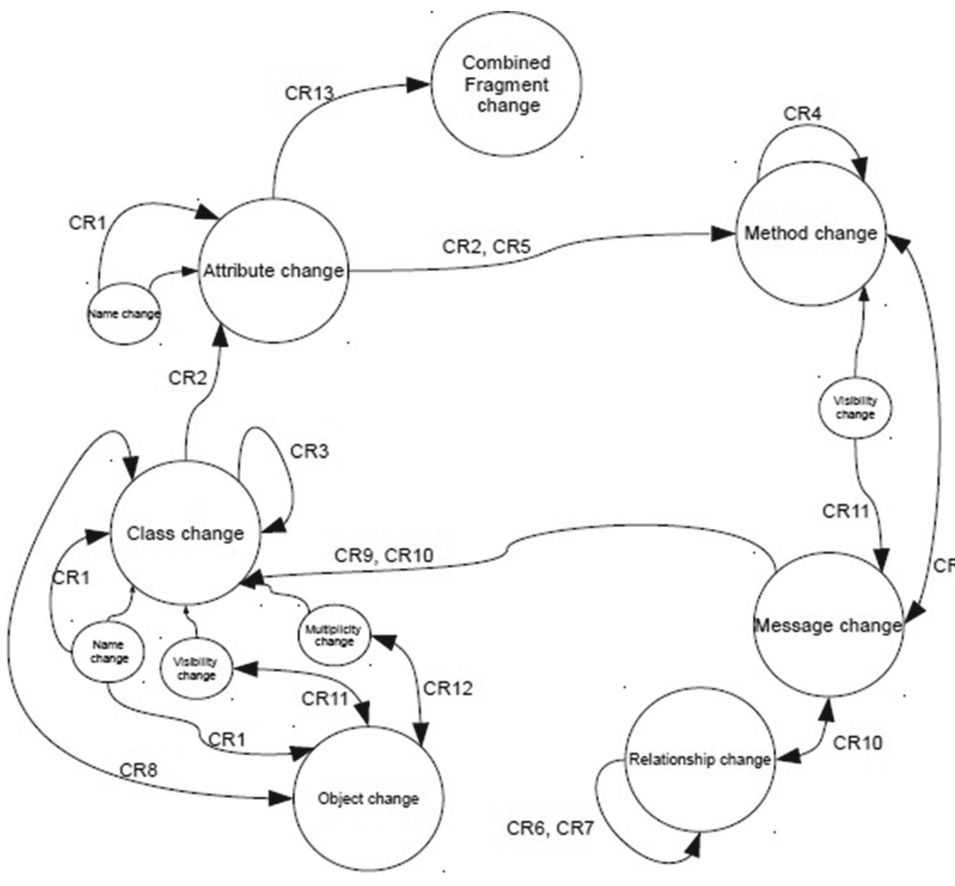


Fig. 2 CFG for our change impact analysis approach between class and sequence diagrams

Note that, the changes can be transitive (i.e., an element change can lead to another element change after the violation of a consistency rule). To summarize, we present the control flow graph (CFG) as illustrated in Fig. 2. That is, the CFG is used to detect inconsistencies that may occur after a change correction.

5 Example

In this section, we illustrate the use of our approach by showing how to browse the MDG when applying our change impact analysis rules. In addition, we illustrate the calculus of the NRM metric to help the designer in making the appropriate decision about the change induced corrections.

To illustrate how to construct the MDG, let us consider the automatic teller machine (ATM) system example [44]. We work on the CD shown in Fig. 3. The CDG corresponding to our class diagram example is presented in Fig. 4. It shows that the classes (ATM, Networktobank, customerconsole, log, Transaction, session, etc.), attributes, methods and parameters are transformed into nodes. Member dependence edges connect a class with its attributes. Method dependence edges connect a method with its parameters. Data dependence edges are used to show that the "Performsession()" method uses the "atm" attribute (this information is presented as a note in the CD). Association relationships are also presented between class nodes. Finally, multiplicities and visibilities are added to the CDG.

The SDGs corresponding to the sequence diagrams of Fig. 5 (SD1) and (SD2) are presented in Fig. 6. The SDG1 shows that objects (ATM, Card reader, Customerconsole) are transformed to square nodes. "Session" and "Transaction" objects are transformed into rounded square predicate class nodes since they participate to a combined fragment. Messages between these objects (I1(6), I1(7)) are transformed into control dependence edges.

The MDG corresponding to the integration of the CDG of Fig. 4 and the SDGs of Fig. 6 is presented in Fig. 7. The class nodes in the CDG, the SDG 1 and the SDG 2 (ATM, Session,Transaction, Cardreader, NetworktoBank, ReceiptPrinter and Customerconsole) are included in the MDG. The dependence edges presented in the CDG between these class nodes are also included in the MDG. The call dependence edges between two class nodes (CNSource, CNTarget) in an SDG are transferred between the class node "CNSource" and the method node of the class "CNTarget". For instance, the message I1(1) between the "Cardreader" and "ATM" class nodes is transformed in the MDG between the "Cardreader" class node and the "cardinserted()" method node of the "ATM" class.

Let us suppose that the designer wants to make the following changes to the class diagram presented in Fig. 3: (1) delete the attribute "atm" that belongs to the "Cardreader" class, (2) delete the operation "ejectcard" belonging to the class "Card reader", (3) delete the association relationship between the classes "ATM" and "Session" (4) add the operations "OpenConnection()" and "CloseConnection()" to the "NetworkToBank" class.

The deletion of the attribute "atm" makes the CD erroneous and triggers intra-diagram inconsistencies indicating the non respect of the consistency rules CR2 and CR5. On the one hand, the data dependence edge between the "Performsession()" method and the "atm" attribute shows that this method uses the attribute "atm" and consequently the deletion of this attribute affects the CD and violates the CR2. To correct this inconsistency, the designer has to decide whether to undo the change or to update the methods which use the "atm" attribute. In our case, he has to update the "Performsession()" method which uses the deleted attribute.

$$\begin{aligned} \text{NRM} &= \text{NAt/Op} \\ &= \text{number of data dependence edge related to the "atm"} \\ &\quad \text{attribute node} = 1 \end{aligned}$$

On the other hand, we can see that the deleted attribute "atm" is used as a parameter (PR node) in the "CreateTransaction()", "PerformTransaction()" and "CompleteTransaction()" methods of the "Transaction" class. Thus, the consistency rule CR5 is not respected and must be analyzed and corrected. In this case, the designer also should choose between undoing the change and updating the "CreateTransaction()", "PerformTransaction()" and "CompleteTransaction()" methods. As a consequence, he has to update the methods which use the "atm" attribute as a parameter. For this change, we have:

$$\text{NRM} = \text{NAtPr} = \text{number of "atm" parameter node} = 3$$

The "atm" attribute is not used in a combined fragment in SDs; thus, this change does not trigger the CR13 and have no inter-diagram impact. Finally, the total NRM needed to correct the deletion of the "atm" is the sum of NRM after the violation of CR2 and NRM after the violation of CR5, as a result, NRM equals 4.

Concerning the second change, it consists in deleting the "ejectcard" operation of the "Cardreader" class. This operation is called in the SD1 since there exists in the MDG a call dependence edge I1(8) from the "Session" class to this deleted method. The deletion of the operation violates CR9 and the correction consists in deleting the messages in SD1 corresponding to the deleted operation in CD. Note

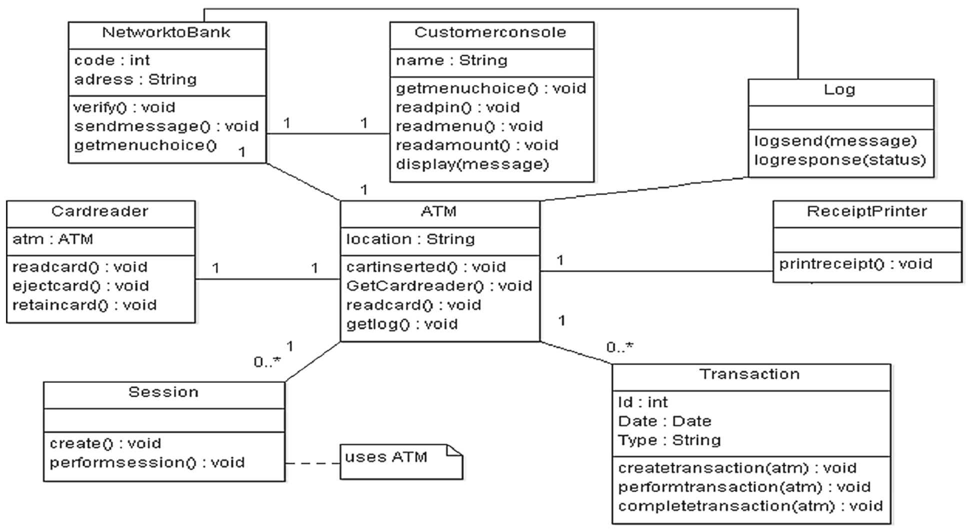


Fig. 3 The ATM system class diagram (CD) [44]

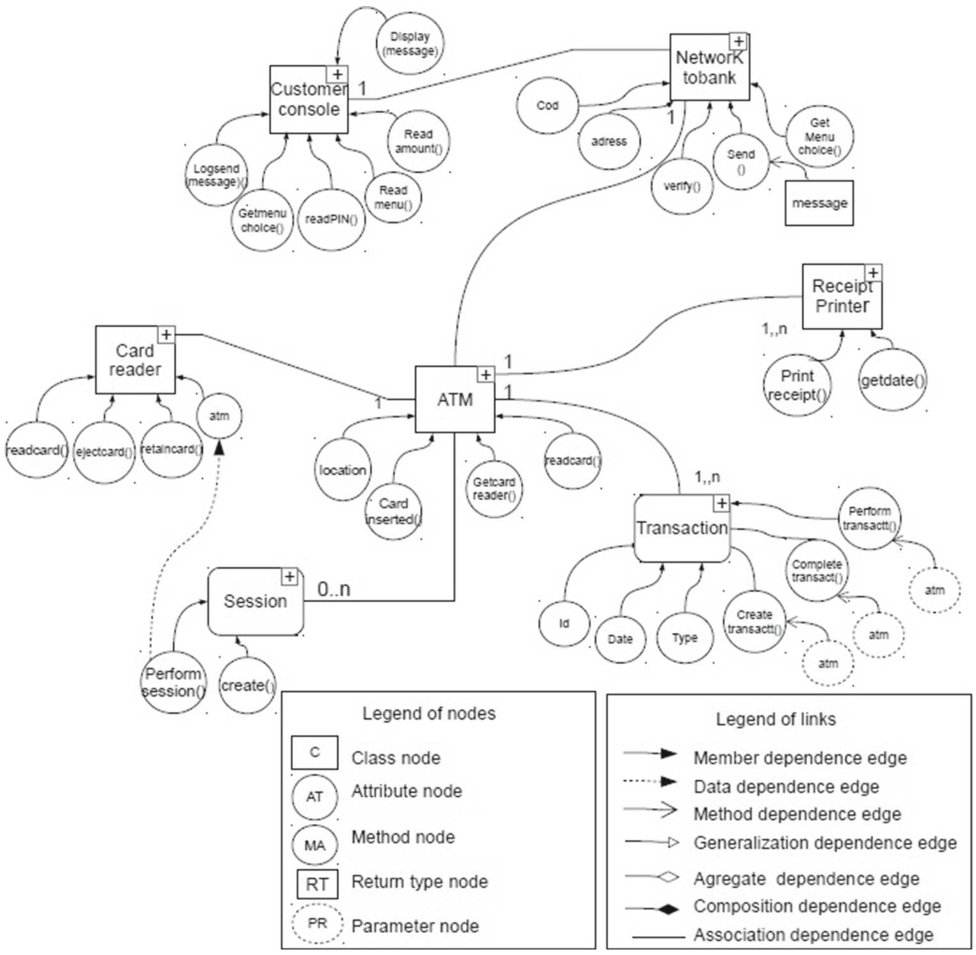


Fig. 4 The CDG corresponding to the ATM system CD

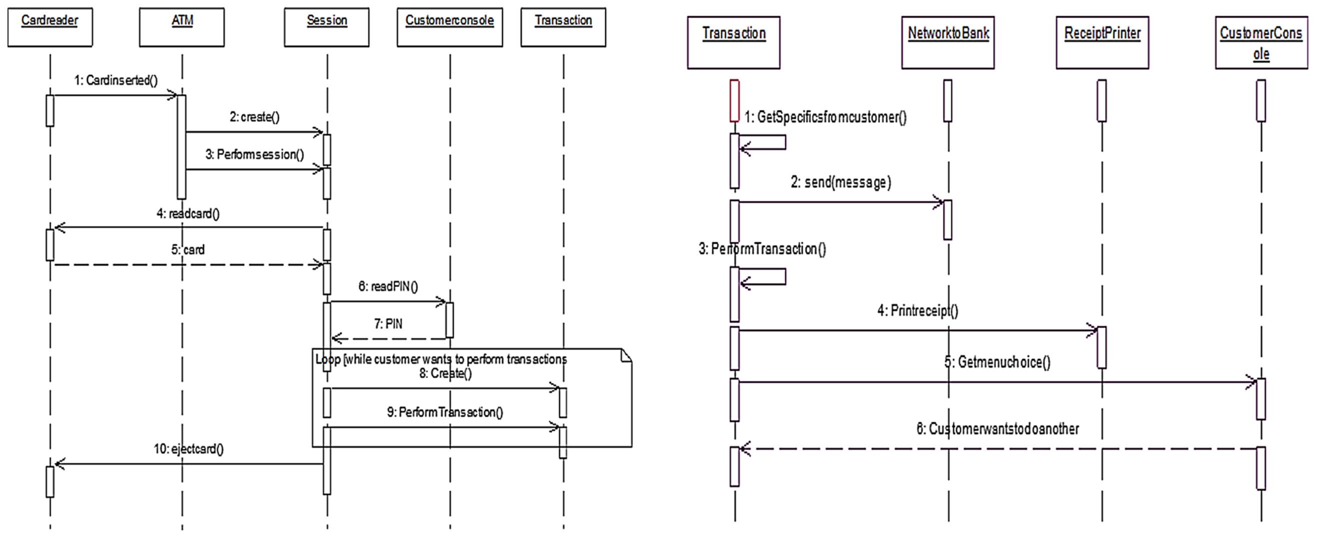
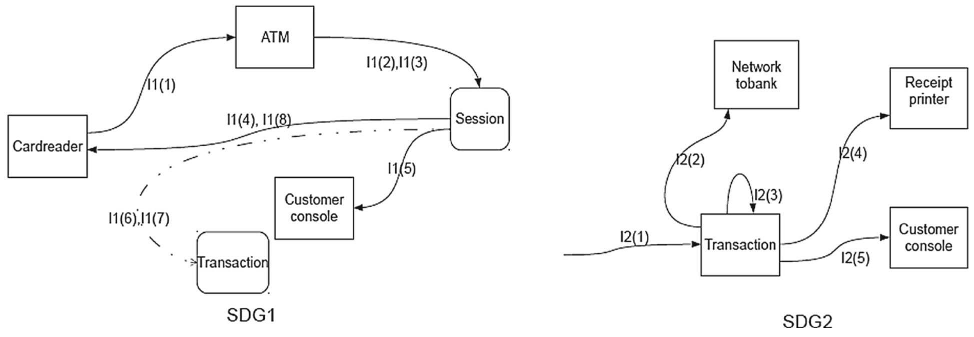


Fig. 5 Session and Transaction sequence diagrams of the ATM system [44]

Fig. 6 Sequence dependency graphs for the SD1 and the SD2



that the deletion of the “ejectcard” operation does not violate the intra-diagram consistency rule CR4. For this change, we have:

$$\begin{aligned} \text{NRM} &= \text{NOpSD} \\ &= \text{number of call dependence edges} \\ &\quad \text{to the “ejectcard” method node} = 1. \end{aligned}$$

The third change, delete the association relation between the “ATM” and “Session” classes, triggers the consistency rule CR10. The call dependence edge from the “ATM” class to the methods of the “Session” class require the existence of an association relationship between these classes. To correct this inconsistency, the designer should choose to undo the change or to delete the corresponding messages in SD: the messages between “ATM” and “Session” objects (I1(2), I1(3)). For this change, we have:

$$\begin{aligned} \text{NRM} &= \text{NM2Ob} \\ &= \text{number of call and return dependence} \\ &\quad \text{edges between “ATM” and “Session”} \\ &\quad \text{classes (I1(2), I1(3))} = 2. \end{aligned}$$

Finally, the fourth change consists in adding the “Openconnection” and “Closeconnection” operations to the “NetworkToBank” class. After calculating the WMC metric, which equals the number of methods in the “NetworkToBank” class and equals 5, we inform the designer that the addition of this operation increases the WMC and consequently the time and effort to develop and maintain the class as indicated by rule QR1.

The total number of NRM for the entire set of changes is equal to 7. This number helps the designer to decide not only about single changes, but also about the global change.

To support our change impact management approach, we developed a tool named CQV-UML Tool: a Consistency and Quality Verification tool for UML diagrams. This tool automates the change impact management on UML diagrams while taking into account the quality of the UML models and the effort needed to handle this change.

The principal activities performed by CQV-UML Tool are: the change detection, the consistency verification and quality verification. The tool takes as input a set of UML models versions corresponding to the original and changed diagrams in order to record and display the list of changes to the designer. Afterward, our graph-based technique (MDG)

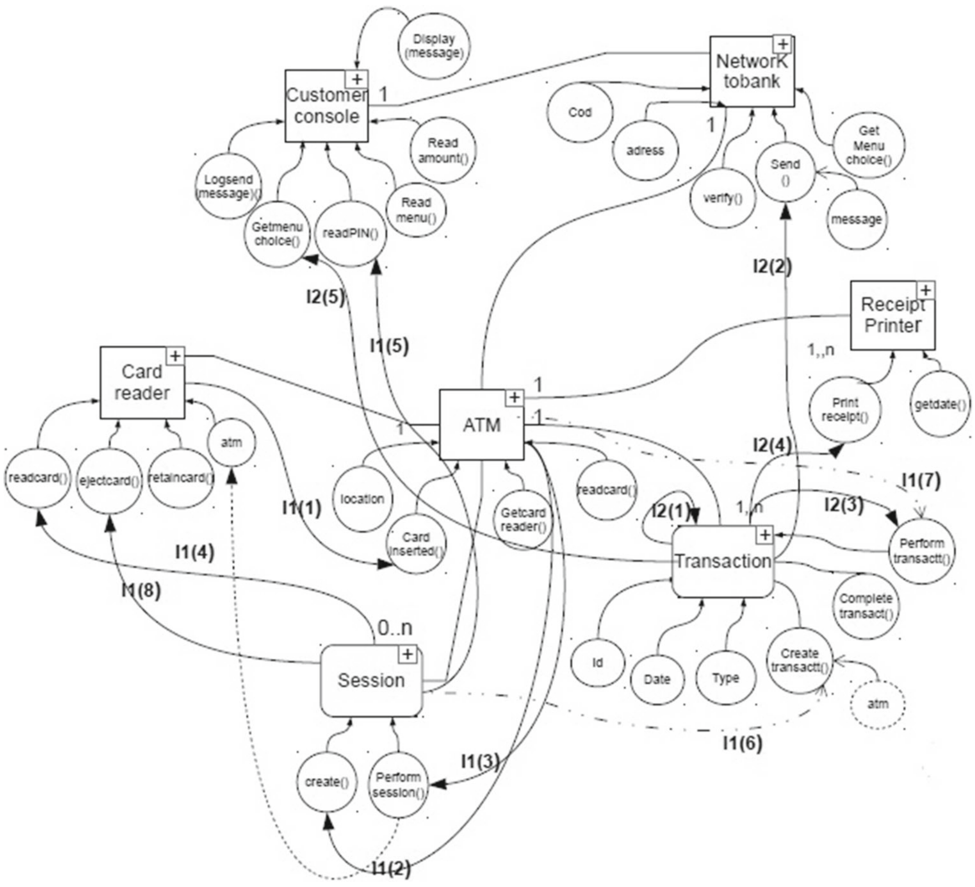
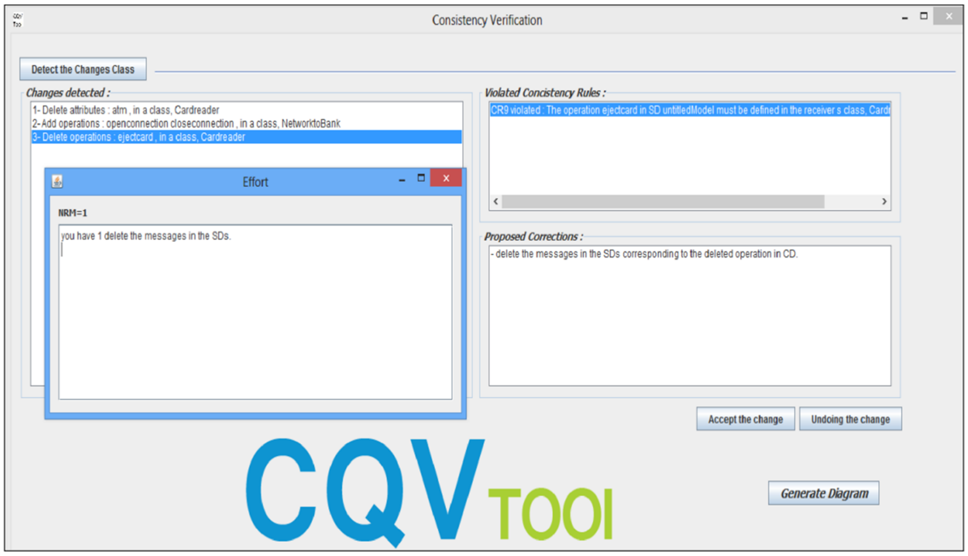


Fig. 7 MDG for the example CDG and SDGs of Figs. 4 and 6



Consistency Verification

Detect the Changes Class

Changes detected :

1- Delete attributes : atm , in a class, Cardreader
2- Add operations : openconnection closeconnection , in a class, NetworktoBank
3- Delete operations : ejectcard , in a class, Cardreader

Effort

NRM=1

you have 1 delete the messages in the SDs.

Violated Concistency Rules :

CR9 violated : The operation ejectcard in SD untitledModel must be defined in the receiver s class, Card

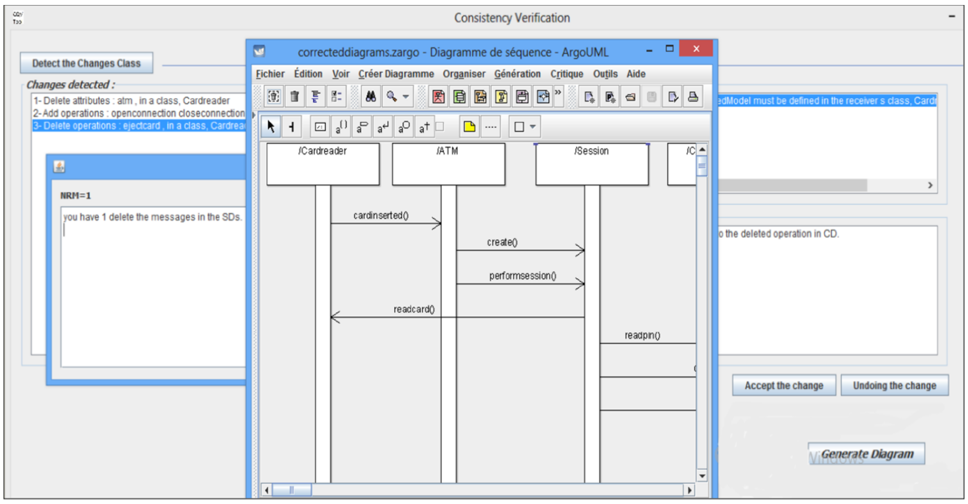
Proposed Corrections :

- delete the messages in the SDs corresponding to the deleted operation in CD.

Fig. 8 The violated consistency rule and proposed correction for the delete ejectcard() operation change

is implemented in order to establish the traceability between the interdependent diagrams. Based on the achieved traceability, the set of violated consistency rules corresponding to each change type is displayed to the designer. Figure 8 depicts a screen snapshot of the consistency verification after the deletion of the "ejectcard()" operation. The violated consistency rule CR9 is detected and its corresponding corrections are presented. Based on the NRM and on the proposed corrections, the designer chooses between the undo of the change and the acceptation.

When the designer accepts the proposed corrections, CQV-UML Tool parses the CFG to detect if these proposed corrections lead to inconsistencies. Otherwise, the affected diagrams are modified in order to cope with the changes and



Changes detected :
1- Delete attributes : atm , in a class, Cardreader
2- Add operations : openconnection closeconnection
3- Delete operations : ejectcard , in a class, Cardrea

NRM=1

you have 1 delete the messages in the SDs.

edModel must be defined in the receiver s class, Card

o the deleted operation in CD.

Fig. 9 Generation of the corrected sequence diagram

they are displayed to the designer. Figure 9 shows the generation of the corrected sequence diagram.

6 Empirical Investigation

The overall objective of this section is to show the ability of our CIA approach in preserving the consistency and the quality of the UML class and sequence diagrams after a change. For this purpose, we evaluated our method based on a comparison between results (impacted elements) built by applying our method and results constructed by experts. Experts are specialists in UML modeling and they have several years of experience studying and developing projects designed with UML. In addition, they are project managers who have dealt with change management in their projects. The experts were recruited through a convenience sampling, we already knew them before and we had their contact details. Note that they participated in the experiment for free.

Following the Wohlin et al. [45] empirical investigation guidelines, we next present the planning, operation (data collection) and finally the analysis and interpretation of the results.

6.1 Planning and Data Collection

For evaluation purposes, we presented three cases to three UML experts: the Automatic Teller Machine (ATM) and EU-Rent system adapted from [1], and the Cruise control system adapted from [2]. The ATM model consists of one class diagram containing 18 classes and 15 sequence diagrams. The EU-Rent system consists of 4 class diagrams containing about 75 classes and 52 sequence diagrams containing around 164 lifelines. The Cruise Control System consists of one class diagram containing 37 classes, and 8 sequence diagrams.

Note that, we proposed four change types per case, meaning 12 changes in total. In each case, we applied different change operations: an addition, a modification and two deletions. Each expert made an independent solution in a first step, and then a consensus discussion was launched among all experts to obtain one consensual solution; this latter is compared with our solution (constructed by our tool).

6.2 Evaluation Results and Interpretation

For evaluation purposes, we adapt the measures of precision and recall introduced in the domain of information retrieval [46]. In our experiment, precision represents the number of correct impacted elements detected by our tool among the found impacted elements; recall represents the number of correct impacted elements detected by our tool among all the existing real impacted elements (detected by the experts).

When calculating the above measures, we interpret the results by counting the number of true positives (TP), false positives (FP), and false negatives (FN). False positives are impacted elements wrongly identified. False negatives are actual impacted elements that have not been detected by our approach. The sum of true positives and false negatives is equal to the total number of actual impacted elements in the analyzed UML class and sequence diagrams. Moreover, precision is TP/(TP+FP) and Recall is TP/(TP+FN).

In the ATM system evaluation (illustrated in Table 5), the precision is 0.82 which is explained by the fact that we found some false positive impacted elements (i.e., incorrect detected impacted elements). These false positives are due to some incoherencies between the terms used in the class and sequence diagrams. For instance, the name of a method in the class diagram is very different from the messages in the sequence diagrams, consequently no traceability is established between them and the deletion of the class does not lead to the deletion of the corresponding erroneous objects.

Table 5 A comparative study by measurement

Cases	TP	FP	FN	Precision	Recall
ATM					
C1	18	5	2	0.82	0.94
C2	12	3	1		
C3	15	2	1		
C4	5	1	0		
Cruise control					
C1	19	2	2	0.95	0.97
C2	18	0	0		
C3	2	0	0		
C4	10	1	0		
EU-Rent					
C1	8	0	1	0.88	0.94
C2	10	2	0		
C3	25	3	2		
C4	18	4	1		
Average	0.883	0.95			

The average recall for the ATM system is 0.94 indicating that we have also some false negative impacted elements (i.e., true impacted elements that are not detected). These false negatives can be explained by the fact that our MDG represents every class using a unique class node. This is irrespective with the number of objects of a class existing across various sequence diagrams. For example, in our case, the "Transaction" class is instantiated more than once in the ATM sequence diagrams. However, the deletion of this class does not lead to the deletion of all object instances. This increases the necessity to create additional nodes in case of repeated instantiation of any classes, and their objects. Moreover, the concept of abstract classes and interfaces are not yet included in the MDG.

To position our work with existing works, we compare our results to the evaluation results of Keller et al. [1] for the ATM system (precision = 0.97, recall = 0.93) and the EU-Rent system (precision = 0.77, recall = 0.95). Keller et al. [1] have slightly better precision than our method in the ATM system; however, it has a lower recall. In our context, we consider that a higher recall is better since it implies that we found more correct impacted elements. In the EU-Rent system, our precision and recall values are better than those of Keller et al. The results of this comparison should however be taken with caution. Indeed, even though similar to Keller et al. we worked on the ATM system, we do not have the same changes. In fact, to have a fair comparison with existing works and to be able to evaluate thoroughly the efficiency of the different existing approaches, we must evaluate all the approaches on the same corpus. Unfortunately, there is no standard or historical datasets that could be used. We believe that it is necessary to have a benchmark for change impact analysis and this would be an interesting research axis.

6.3 Threats to Validity

Considering the validity of the results of an experiment is a fundamental point in each research. The results are said to have adequate validity if they are valid for the population to which we would like to generalize [45]. Different types of validity can be categorized depending on the goal of the experiment. In our case, the analyzed threats to validity are: *external validity* and *internal validity*.

Threats to external validity are conditions that limit our ability to generalize the results of our experiment to industrial practice [45]. This threat concerns the selection of people who participate in the experiment. On the one hand, when evaluating our approach, the experts had to be chosen carefully since if they are inexperienced, this produces wrong evaluations. We tried to reduce this threat to external validity by making the experimental environment as realistic as possible through the selection of experienced experts who have really made and managed changes. On the other hand, if the UML designers produce models that are not coherent and the terminology used in the different diagrams differs from one diagram to another, this makes the traceability very difficult and consequently the change impact cannot be determined correctly. (This was highlighted in the ATM case.)

Besides the above external threat, we identify two internal threats. The first internal threat concerns the construction of the model dependency graph "MDG" used as a traceability model. More specifically, if the MDG lacks some relations between model elements due to the fact that we do not treat interfaces or abstract classes, then our approach would fail in detecting all impacted elements. Consequently, if the MDG is not well constructed and misses information, then the consistency and the quality of the derived corrected models is not guaranteed in terms of consistency rules, quality rules and metrics. Moreover, the second internal threat concerns the correctness of the implementation [45]. A major concern is the difficulty in manipulating the data structure of a large MDG; indeed, it is hard to combine all the UML diagrams in one file when we have overloaded diagrams because of the huge number of relations (dependence edges).

7 Conclusion

This paper proposed an approach for the management of change impact on UML class and sequence diagrams based on three techniques: graph-based, rule-based, and metric-based techniques. The main originality of our approach is its capacity to analyze the change impact on both the consistency within each diagram and among the related diagrams, as well as the quality of the changed diagrams. To do so, our method uses: a dependency graph to identify impacted elements by coding both syntactic and semantic dependencies among the elements of the class and sequence elements; a set of rules to express the consistency constraints among the intra and inter UML diagram elements; and a set of metrics and quality rules (derived from best practices) to estimate the change impacts in terms of quality effects and effort needed to carry out all changes necessary to preserve the consistency of the design. Combined, these techniques allow our method to collect information needed to assist the designer making a judicious decision about accepting or refusing a change based on its impact on the effort needed to accommodate it and the resulting model quality.

To assess the efficiency of our approach, we implemented a tool support (CQV-UML Tool) that we used to conduct a quantitative experimental evaluation. Evaluated on three cases with the help of three UML experts, our method has on average a precision of 88.33 % and recall of 95 %. These rates are encouraging, nonetheless should be generalized with caution due to two essential threats to the validity of our experiment: an external threat pertinent to the choice of the participating experts, and an internal threat pertinent to the correctness of the implementation. As such, a wider evaluation should be undertaken in real software development settings where the expertise of the designers would validate both the performance of the method and the correctness of its tool support.

Furthermore, as noted in our comparative evaluation attempt, it is necessary to establish a benchmark of designs that can be used to compare the various change impact analysis approaches.

Besides elaborating the benchmark and extending our approach to the remaining development phases and activities (e.g., testing), another research axis we are currently pursuing is how to integrate our method within a project management approach for a better accounting of change impacts.

References

1. Keller, A.; Demeyer, S.: Change impact analysis for UML model maintenance. Book chapter: Emerging Technologies for the Evolution and Maintenance of Software Models, pp. 32–56 (2012)
2. Briand, L.C.; Labiche Y.; O'Sullivan, L.: Impact Analysis and Change Management of UML Models. In: Proceedings of the International Conference on Software Maintenance, pp. 276–280 (2003)
3. Zimmermann, T.; Weigerber, P.; Diehl, S.; Zeller, A.: Mining version histories to guide software changes. In: Proceedings of the 26th International Conference on Software Engineering, Edinburgh, Scotland, UK, pp. 563–572 (2004)
4. Kagdi, H.; Gethers, M.; Poshyvanyk, D.; Collard, M.L.: Blending conceptual and evolutionary couplings to support change impact analysis in source code. In: Proceedings of the IEEE Working Conference on Reverse Engineering, Beverly, MA, USA, pp. 119–128 (2010)
5. Petrenko, M.; Rajlich, V.: Variable granularity for improving precision of impact analysis. In: Proceedings of the International Conference on Program Comprehension, Vancouver, BC, Canada, pp. 10–19 (2009)
6. Tsiolakis, A.: Consistency analysis of UML class and sequence diagrams based on attributed typed graphs and their transformations. ETAPS Workshop on Graph Transformation Systems (2000)
7. Hammad, M.; Collard, M.L.; Maletic, J.: Automatically identifying changes that impact code-to-design traceability during evolution. J. Softw. Qual. **19**(1), 35–64 (2011)
8. Knethen, A.; Grund, M.: QuaTrace: a tool environment for (semi-) automatic impact analysis based on traces. In: Proceedings of the International Conference on Software Maintenance, Amsterdam, Netherlands, pp. 246–255 (2003)
9. Rocco, J.D.; Ruscio, D.D.; Iovino, L.; Pierantonio, A.: Traceability visualization in metamodel change impact detection. In: 2nd Workshop on Graphical Modeling Language Development, pp. 51–62 (2013)
10. Khalil, A.; Dingel, J.: Supporting the Evolution of UML Models in Model Driven Software Development: A Survey, Technical Report 2013-602, School of Computing, Queen's University Kingston, Canada (2013)
11. Kchaou, D.; Ben-Abdallah, H.; Bouassida, N.: Change impact management on model transformations. In: Proceedings of the International Conference on Computational Intelligence and Software Engineering (CiSE2012), Wuhan, China, December (2012)
12. Rational Software White Paper, Rational Unified Process Best Practices for Software Development Teams, TP026B (2001)
13. Jacobson, I.; Booch, G.; Rumbaugh, J.: The Unified Software Development Process. Addison-Wesley Longman, Reading (1999)
14. Chanda, J.; Kanjilal, A.; Sengupta S.: UML-Compiler: a framework for syntactic and semantic verification of UML diagrams. In: International Conference Distributed Computing and Internet Technology, LNCS, vol. 5966, pp. 194–205 (2010)
15. Chidamber, S.R.; Kemerer, C.F.: Towards a metrics suite for object oriented design. In: Conference proceedings of Object-oriented programming systems, languages, and applications, pp. 197–211 (1991)
16. Mohagheghi, P.; Dehlen, V.: Existing model metrics and relations to model quality Software Quality. In: ICSE Workshop, pp. 39–45 (2009)
17. OMG Unified Modeling Language TM OMG UML, Version 2.4.1. http://www.omg.org/spec/UML/2.4.1 (2011)
18. Li, B.; Sun, X.; Leung, H.; Zhang, S.: A survey of code-based change impact analysis techniques. J. Softw. Test. Verif. Reliab. **23**, 613–646 (2012)
19. Badri, L.; Badri, M.; St-Yves, D.: Supporting predictive change impact analysis: a control call graph based technique. In: Proceedings of the Asia-Pacific Software Engineering Conference, Taipei, Taiwan, China, pp. 167–175 (2005)
20. Sun, X.; Li, B.; Tao, C.; Wen, W.; Zhang, S.: Change impact analysis based on a taxonomy of change types. In: International Conference on Computer Software and Applications, Seoul, Korea, pp. 373–382 (2010)
21. Beszedes, A.; Gergely, T.; Farao, S.; Gyimothy, T.; Fischer, F.: The dynamic function coupling metric and its use in software evolution. In: Proceedings of the 11th European Conference on Software Maintenance and Reengineering, Amsterdam, Netherlands, pp. 103–112 (2007)
22. Apiwattanapong, T.; Orso, A.; Harrold, J.M.: Efficient and precise dynamic impact analysis using execute-after sequences. In: Proceedings of the International Conference on Software Engineering, St. Louis, MO, USA, pp. 432–441 (2005)
23. Briand, L.C.; Labiche, Y.; O'Sullivan, L.; Sówka, M.: Automated impact analysis of UML models. J. Syst. Softw. **79**, 339–352 (2006)

24. Egyed, A.: Fixing Inconsistencies in UML Design Models. In: Proceedings of the 29th International Conference on Software Engineering, pp. 292–301 (2007)
25. Egyed, A.: UML/Analyzer: a tool for the instant consistency checking of UML models. In: Proceedings of the 29th International Conference on Software Engineering, pp. 793–796 (2007)
26. Dam, H.K.; Winiko, M.: Supporting change propagation in UML models. In: Proceedings of the 26th IEEE International Conference on Software Maintenance, IEEE Computer Society, Washington (2010)
27. Dam, H.K.; Ghose, A.: Towards rational and minimal change propagation in model evolution. CoRR. abs/1402.6046 (2014)
28. Shiri, M.; Hassine, J.; Rilling, J.: A requirement level modification analysis support framework. In; Proceedings of the International Workshop on Software Evolvability, Maison Internationale, Paris, France, pp. 67–74 (2007)
29. Ganter, B.; Wille, R.: Formal Concept Analysis: Mathematical Foundations. Springer, New York (1997)
30. Trollmann, F.; Blumendorf, M.; Schwartze, V.; Albayrak, S.: Formalizing model consistency based on the abstract syntax. In: Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems, pp. 79–84 (2011)
31. Elaasar, M.; Briand, L.: An Overview of UML Consistency Management, Technical Report SCE-04-18, 1125 Colonel-By Drive, Ottawa, ON, August 24 (2004)
32. Liu W.; Easterbrook, S.; Mylopoulos, J.: Rule-based detection of inconsistency in UML models. In: Proceedings of UML Workshop on Consistency Problems in UML-based Software Development, Blekinge Institute of Technology, pp. 106–123 (2002)
33. Spanoudakis, G.; Zisman, A.: Inconsistency management in software engineering: survey and open research issues. Handb. Softw. Eng. Knowl. Eng. **1**, 329–380 (2001)
34. Lallchandani, J.T.; Mall, R.: Static slicing of UML architectural models. J. Object Technol. **8**(1), 159–188 (2009)
35. Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: Design Patterns: Elements of Reusable Object Oriented Software. Addisson-Wesley, Reading (1995)
36. Chandra, E.; Linda, P.: Class break point determination using CK metrics thresholds. Glob. J. Comput. Sci. Technol. **10**, 73–77 (2010)
37. Ferreira, K.A.M.; Bigonha, M.A.S.; Bigonha, R.S.; Mendes, L.F.O.; Almeida, H.C.: Identifying thresholds for object-oriented software metrics. J. Syst. Softw. **85**, 244–257 (2012)
38. Bakar, A.D.; Sultan, A.B.M.; Zulzalil, H.; Din, J.: Review on software metrics thresholds for object-oriented software. Int. Rev. Comput. Softw. **8**(11), 2593–2600 (2013)
39. Tarcísio, G.S.F.; Bigonha, M.A.S: A catalogue of thresholds for object-oriented software metrics. In: SOFTENG 2015: The First International Conference on Advances and Trends in Software Engineering (2015)
40. Bouassida, N.; Ben-Abdallah, H.; Issaoui, I.: Evaluation of an automated multi-phase approach for pattern discovery. Int. J. Softw. Eng. Knowl. Eng. **23**(10), 1367–1398 (2013)
41. Arnold, R.; Bohner, S.: Change Impact Analysis, 1st edn. Wiley-IEEE Computer Society Press, New York (1996)
42. Kchaou, D.; Bouassida, N.; Ben-Abdallah, H.: A MOF-based change meta-model. Proceedings of the International Arab Conference on Information Technology, CCIS, Zarqa, Jordon (2012)
43. OMG Meta Object Facility (MOF) Core Specification, Version 2.4.1, OMG Document Number: formal/2011-08-07. http://www.omg.org/spec/MOF/2.4.1/PDF (2011)
44. Russell, C.B.: An Example of Object-Oriented Design: An ATM Simulation, Gordon College, Copyright2004. http://www.math-cs.gordon.edu/courses/cs211/ATMExample/
45. Wohlin, C.; Runeson, P.; Höst, M.; Ohlsson, M.C.; Regnell, B.; Wesslén, A.: Experimentation in Software Engineering. Kluwer, Norwell (2000)
46. Frakes, W.B.; Baeza-Yates, R.: Information Retrieval: Data Structures and Algorithms. Prentice Hall, Englewood Cliffs (1992)