Proposition and evaluation of a software Change Impact Analysis model

Julien Delplanque

Faculté
des Sciences
Department of Computer Sciences

UMONS
Université de Mons

Proposition and evaluation of a software Change Impact Analysis model

Master's Thesis submitted by Julien DELPLANQUE
With a view of getting the degree of Master in Computer Sciences

Academic Year 2016–2017

Supervisors: Dr. Anne ETIEN & Pr. Tom MENS

Team: Software Engineering

Faculty of Sciences • University of Mons • Place du Parc 20 •
B-7000 Mons

Remerciements

Ce mémoire représente l'aboutissement de ma formation. Sa réalisation n'aurait pas été possible sans l'aide de plusieurs personnes à qui je tiens à exprimer ma gratitude et ma reconnaissance.

Tout d'abord, je remercie Anne Etien, directrice de ce mémoire, pour ses précieux conseils, sa disponibilité et son encadrement tout au long de la réalisation de ce travail.

Je remercie également Tom Mens pour l'encadrement de ce mémoire et, plus généralement, pour m'avoir fait découvrir le domaine de l'ingénierie logiciel durant mon cursus. De façon plus générale, je remercie tout le personnel de l'Université de Mons pour avoir contribué à ma formation et pour la découverte de la diversité des domaines de l'informatique.

Je remercie Nicolas Anquetil, Stéphane Ducasse, Vincent Blondeau et Olivier Auverlot pour les diverses discussions que j'ai pu avoir avec eux dans le cadre de ce travail et pour leurs différents conseils et relectures.

Finalement, je remercie toute personne qui a contribué de près ou de loin à la réalisation de ce mémoire.

Contents

Introduction

Software often needs to change. These changes are motivated by the need to introduce new features, to fix bugs, to improve the non-functional quality of the software, etc... To apply these changes, adaptations of the software source code may be needed.

In RMod team[1], searchers are working in different fields related to software changes. Gustavo worked on improving refactoring techniques in Pharo. Vincent works on the computation of the smallest unit test set to re-execute after a change has been performed on a software that ensure that tests concerning features impacted by the modification are run. Véronica worked on techniques to integrate features from a software into another software. Each of these works uses Change Impact Analysis to compute the impact a change have on the software concerned. Indeed, Gustavo needs to know how his refactoring technique will impact a software to adapt it accordingly, Vincent needs to compute the impact of a change to find the unit tests to re-execute and Véronica needs to assess what will have to be modified to integrate the new features. Nevertheless, they are all using a custom technique to evaluate impact. From this observation comes the following question: *Can a unified technique, that could be adapted to the needs of anyone, be implemented?*

Furthermore, RMod team often works on the analysis of huge software that can be written in any language and in any paradigm. To analyse these software systems, different modelling technologies can be used. These facts induce that *an ideal tool to perform Change Impact Analysis would be independent of the language, the paradigm and the model used.*

This document aims to answer this question and requirements. The first step to do that was to realised a state of the art of Change Impact Analysis research field. This state of the art aims to answer the two following questions. First, *What approaches already exist in the literature?* and second, *What are the essential features that such approach should provide?*

Using the knowledge acquired during the state of the art realisation, a generic approach to compute the impact of a change on a software has been created as well as an extensible change model. Then, an implementation of the approach in the Pharo[2] environment has been proposed. This implementation is generic and

[1] https://rmod.inria.fr/web
[2] http://pharo.org

can potentially be used to analyse the impact of a change on a software of any language using any paradigm and being modelled by any modelling technology. The only thing required to do that is to specialise the framework by overriding specific classes and methods.

The framework has then been specialised to analyse the impact of a change on Pharo software. This specialisation is provided with a graphical user interface as well as an integration the development environment of Pharo to make its use more user-friendly.

The end of this document is dedicated to the evaluation of the approach. A complicated part of the evaluation of a tool performing Change Impact Analysis is the analysis of its performance in terms of precision and recall. The details of the problems caused by the assessment of the performances are detailed in this part of the work. Because of this complexity and because of a lack of time to perform the experiment, no experimental results could be get. Nevertheless, a protocol to evaluate the performance of the tool is presented and its threats to validity are discussed.

The remainder of this document is organized as follow. Chapter I presents Change Impact Analysis research field at a high level of abstraction by explaining and discussing Lehnert taxonomy and its evaluation [15, 16]. The Chapter II explores approaches using *Call Graphs* and *Dependency Analysis* in order to create a state of the art of the approaches using these techniques which are used by the tool proposed in this document. From this state of the art, the objectives that guided the approach conception are developed. Chapter III introduces the approach proposed to fulfil use cases and achieve objectives defined in the preceding chapters. Then, Chapter IV evaluates the approach complexity and classifies it using Lehnert Taxonomy. The problems associated to the measurement of Change Impact Analysis performance are also discussed and an experimental protocol is proposed. Finally, this work is concluded and some perspectives are discussed.

Chapter I

Change Impact Analysis

Change Impact Analysis is the research field that focuses on understanding how a modification on a software (a change) will impact it. That is to say, analysing what parts of the software, that are not directly involved in the change, are potentially subject to be impacted because of their relations with the entity changed. Many tools and methods have been created over the years to approximate the impact of a change on a software. This chapter aims to present the research field at a high level of abstraction in order to have a view on existing approaches. For this purpose, the motivation behind Change Impact Analysis are developed. Next, the important concepts of the research field are presented. Then, a taxonomy for Change Impact Analysis approaches is presented. Finally, the taxonomy is discussed as well as the new research topics it raises.

I.1 Motivation

Change Impact Analysis is mainly motivated by helping developers to apply changes on a software and adapt it to this change. This is done by, giving a change on a software, showing *what* will be impacted and eventually *how*. In the context of this work, three use cases were identified where Change Impact Analysis can help. These use cases are explained in the first sub-section of this section.

The automation of Change Impact Analysis is motivated by studies suggesting that developers are bad at predicting the impact of a change on a software [18, 24]. These studies, explained in the second sub-section of this section, provide a good motivation for creating a strong tool support to help developer in change integration.

I.1.1 Use Cases

To strengthen the need for change impact analysis tools, three use cases where the use of Change Impact Analysis is required are described in this sections. These use cases are inspired from the process normally followed by developers when they want to change a software. That is to say, lets say a developer want to modify a software. He will first *estimate the cost of the change*, second *apply*

the change and third, *re-execute the unit tests impacted by the modification.* These three use cases are detailed below.

Change Cost Estimation

The implementation of a new feature in a software or the modifications of a software has a cost. This cost can be expressed as time and therefore as money if the person implementing this change has to be paid.

The cost of this change will depend on the amount of modification to perform. This could be evaluated by performing Change Impact Analysis. Indeed, such analysis can compute what has to be changed in the software to integrate the change and how it has to be changed. Thus, it is possible to compute an estimation of cost of this change as time and with more information as money.

Modifying the Software

Once the cost of the modification has been estimated, a developer can start working on its implementation. An impact analysis of the change to perform in order to modify the software can be done before applying it. This analysis will return the set of entities that will be indirectly impacted by the change performed by the developer. Eventually, the change impact analysis can also return how the entities will be impacted. With this set of entities potentially impacted, the developer can integrate the change more easily. Indeed, the developer knows what parts of the software have to be verified in order to integrate the change correctly.

Unit Tests to Re-Execute Selection

After the implementation of the modification, it is needed to check that the modification did not break the behaviour of the software. Indeed, even if the change impact analysis performed in the precedent use case try to ensure that it is not the case, it is still possible that some impact undetected during the analysis modified the software behaviour. Unit tests are pieces of source code that check that the behaviour of a software matches its requirements. To do that, the developer write code that manipulate the software to test in order to check its results.

In large software systems, the number of unit tests can be consequent. Therefore, the time to run them all to ensure that a modification does not modify the software behaviour may be consequent and not acceptable in the work flow of a developer. Nevertheless, unit tests should be run each time a modification is performed.

To solve this problem, a possible solution is to compute the subset of unit tests that are impacted by the developer's changes. By doing that, it is possible to reduce the number of unit tests to run after a modification while still ensuring that the software behaviour did not change. This process uses Change Impact Analysis in order to determine the tests to re-run after a change was performed.

I.1.2 Motivation for Automated Change Impact Analysis

The motivation behind Change Impact Analysis automation comes from the fact that programmers have difficulties to evaluate the impact of a modification on a software. Indeed, multiple studies interested in comparing the results of manual change impact analysis approaches with automated (or semi-automated) change impact analysis approaches concluded that there is a need for a strong tools support [18, 24].

Lindvall [18] conducted an experiment to quantify *how good is a team of professional C++ developers to predict impact in a real project.* From this experiment, he observed that the prediction of developers:

- was correct in about half of the cases of impact analysis proposed;
- identifies only one third of the changed classes of the system; and
- identifies only classes that are actually changed (no false positives).

Lindvall's study concludes that developers are bad at performing the prediction of the impact of a change and that, consequently, there is a need for tools to help developers in this activity.

More recently, Tóth et al. [24] did a similar study. The procedure of the experiment was the following: first, a group of developers has to individually evaluate the impact of different changes using a tool that computes the ripple effect of a change. Their evaluations are recorded. Second, multiple tools are used to evaluate the changes earlier evaluated by developers. The results of these tools are stored. Third, developers are allowed to see the tools' results and to modify their initial evaluation. Finally, the union of the tools' results is compared with developers' results. From their experiment, Tóth et al. observed that:

- the algorithms used during the experiment produce quite different results;
- the overall opinion of developers showed a large deviation;
- the decision of developers after seeing the results of other tools than the one they use did not change much but when it does it is mainly to add a new impacted entity.

As Lindvall's study, Tóth et al. study suggests that better tools to help in the computation of the impact of a change are needed.

There are not much papers that compare the results of human performing impact analysis and the results of automated tools. No other studies were found during the researches made in the context of this work. Nevertheless, these two studies are providing the same analysis on the difficulty for developers to evaluate the impact of a change on a software.

I.2 Definitions

In order to describe Change Impact Analysis research field, some concepts need to be defined. These concepts are really important for the rest of the document since they will often be used.

Software Change: A software change (or simple *a change* in this document) is defined as a modification applied on a software. An example of software change is *rename*. It describes the fact that an entity of the software has its name changed.

Ripple Effect: The ripple effect is defined as a spreading effect or series of consequences caused by a single action or event. In the context of software change impact analysis, it designates the propagation of the effect of a change across the entities of the system. In case of renaming an entity, the ripple effect is to also change the name of the entity each time it is referenced and not only when defined.

Change Impact Analysis: Arnold and Bohner [1] proposed a definition of Impact Analysis which is generally accepted. The definition is the following: "Impact analysis (IA) is the activity of identifying what to modify to accomplish a change, or of identifying the potential consequences of a change". Below, some examples of IA cited from their paper [1]:

- using cross reference listings to see what other parts of a program contain references to a given variable or procedure;
- using program slicing to determine the program subset that can affect the value of a given variable;
- browsing a program by opening and closing related files (to find out the impact of change "manually");
- using traceability relationships to identify changing artefacts;
- using configuration management systems to track and find changes; and
- consulting designs and specifications to determine the scope of a change.

Even if Arnold and Bohner do not use the term "change impact analysis", they note that "IA precedes, or is used in conjunction with, change. It provides input to perform the change". Nevertheless, more recent authors sometime use the term "change impact analysis" to designate "impact analysis". In this report we consider that these two terms have the same meaning.

I.3 Taxonomy

In the literature, there are essentially three taxonomies on Change Impact Analysis that have been proposed since the beginning of the researches in this field: Arnold et al taxonomy [1], Kilpinen's taxonomy [12] and Lehnert's taxonomy [15]. The taxonomy chosen in the context of this work is Lehnert's taxonomy.

This choice was motivated by two main reasons. First, as highlighted by Lehnert, the two others taxonomies are too coarse grained and they do not provide strict criteria to classify research approaches or they lack of important criteria to be used in practice. Second, Lehnert's taxonomy has been evaluated empirically and showed good results. Indeed, on 150 Change Impact Analysis approaches reviewed by Lehnert [16], the coverage of the criteria defined in its taxonomy is of 85% on average. That is to say: on average, for a criterion it was possible to evaluate it on 85% of the 150 approaches.

With his taxonomy, Lehnert provides a way to classify the tool presented later in this document and to compare it with others existing approaches. Lehnert's taxonomy is composed of height main criterion: the *scope analysis*, the *granularities of entities*, the *utilized technique*, the *style of analysis*, the *tool support*, the *supported languages*, the *scalability* and the *experimental results*. Some of these criterion are divided in sub-criterion allowing a finer understanding of the Change Impact Analysis approach being classified.

I.3.1 Scopes of Analysis

The first criterion assesses the scope of analysis in which the approach works (code, models, etc. . .). Scopes of Analysis criterion is subdivided in 2 categories: *code* and *models*.

Code category refers to all the approaches concerned with the source code. This category is itself subdivided in three sub-categories: *static* which is a change impact analysis performed on a representation of the source code without executing it, *dynamic* which uses data collected from a program execution and *online* which is performed at the same time as the program analysed is executed.

Models category refers to all approaches operating at the model level. It is sub-divided in two categories: *architecture* which is a change impact analysis performed on the architecture and abstract design of the software. The abstraction level may vary depending on the case. The second subdivision is named *requirements* and refers approaches performing change impact analysis on formalized requirements specification and software models. Figure I.1 represents the criterion and its categories.

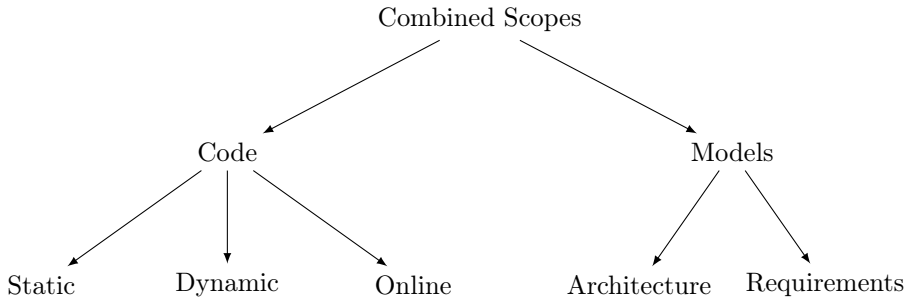


Figure I.1: Scopes of Change Impact Analysis.

Even if the "code" category is dedicated to analysis concerning the code, it is important to realise that, often, the approaches falling in this category build a model from the source code to be able to process it. The difference between these two categories is that for the first category, the model is computed from the code but in the second, there is no code at all. For the second category, the software is directly stored as a model and it is this model that is analysed.

I.3.2 Granularity of Entities

The second criterion evaluates the granularities supported by the approach. That is to say, the precision of artefacts, changes and results in their description of the data they model. The meaning of "granularity" for each entity type is detailed below.

1. *Artefacts granularity*: What is the smallest entity considered? For example, in OO software an approach can perform the analysis at the package level which would be a coarse granularity. A finer approach would be to work at the class/method/instance variable level. An even finer approach consists in working at the statement level. The choice of the granularity to use depends on the information available in the artefacts;

2. *Change granularity*: How detailed is the description of a change? The simplest change description imaginable is simply describing "this entity changed". In this case the granularity of the changes provided to the approach is coarse. On the other hand, a finer granularity of change types can be imagined if there is enough information available. For example it is possible to have change types like renaming of an entity, move of an entity or even more complex types of changes like (for OO software): pull up instance variable, push down instance variable, extract method, etc...

3. *Results granularity*: What kind of results are provided and how precise are their? The coarsest grain result possible is the set of entities that are impacted. A finer grained result would be to have the entities that are impacted and *how* they are impacted.

Depending on the developer expectations on these three granularities, a tool may be preferred over another.

I.3.3 Utilized Technique

The third criterion assesses the techniques and algorithms used by the approach. In his taxonomy [15], Lehnert identified 10 techniques appearing recurrently. This list of techniques shortly presented below may not be complete (since a new technique that has never been used could appear) but it provides a basis to classify a tool according to the technique(s) it uses.

1. *Program Slicing* computes the statements of a program that are affected by a change.

2. *Call Graphs* are graphs built from method and function calls extracted from code.

3. *Execution Traces* technique analyses the traces of methods and functions that have been called during a past execution.

4. *Program Dependency Graphs* are graphs built from the dependencies between program entities extracted from code. On possible dependency to use is call between functions. Nevertheless, for OO software, there are other kinds of dependency like references to entities or accesses to attributes.

5. *Message Dependency Graphs* is designed to model communication between distributed systems and to propagate changes between them.

6. *Traceability links* can be used to compute the propagation of the impact of a change across different level of abstraction.

7. *Explicit Rules* are rules defining explicitly what entities are impacted when an entity is changed. These rules should be defined by an expert of the system.

8. *Information Retrieval* exploit the similarities between attributes of classes, methods, etc... to identify impacted elements.

9. *Probabilistic Models* uses probability to compute the likelihood of a change affecting entities in the program.

10. *History Mining* consists in mining the versions of a software in a repository in order to find which entity are impact when certain entities are changed in a empirical way. Then, from the model built, this technique predicts the impact of a new change on the software.

Additionally to these techniques, it is possible to encounter combinations of them.

I.3.4 Style of Analysis

The fourth criterion evaluates the style used by the analysis. That is to say, the procedure and strategy it supports. The 3 following strategies have been found by Lehnert:

- *Global Analysis* analyses the whole system independently of the task being done. That is to say, for a set of changes, the impact will be computed over the whole system. Independently of what the developer wants to do.

- *Search Based* operates on demand for specific change requests. That is to say, the impact is computed for a specific change and the result only contains entities impacted by this change.

- *Exploratory* operates in a step-by-step fashion, guiding the developer in the possible impacted elements of the system. That is to say, the impact is computed iteratively according to the choices made by the developer in the orientation of the exploration.

I.3.5 Tool Support

The fifth criterion evaluates the approach on whatever it has been implemented or not. It allows to judge its direct usability. Indeed, if a developer needs a tool to perform impact analysis, he needs it implemented. Otherwise he can not use it. The approaches available in the literature are generally not easy to apply by hand on a software for the developer.

I.3.6 Supported Languages

The sixth criterion assesses the languages understood by the approach. It is complementary to the "Scope of Analysis" criterion by giving explicitly the programming language(s) and the modelling language(s) supported by the tools. Two aspect are distinguished:

- *Programming language* which is the language in which is written the software analysed (*e.g.*, Java).
- *Modelling language* which is the language used to model the system (*e.g.*, Unified Modelling Language).

I.3.7 Scalability

This seventh criterion evaluates the scalability of the approach. In its taxonomy, Lehnert decides to use the worst-case time and space Bachmann-Landau-notation (*i.e.*, Big-O-notation). The scalability allows to assess what system size can be treated by the approach. Indeed, for an approach that has a high time or space complexity, the computation of the impact of a change on a big system maybe infeasible in a reasonable time.

I.3.8 Experimental Results

To introduce the last criterion, some additional definitions are needed. These definitions have been proposed by Arnold and Bohner [1] and provide metrics to judge of the precision and recall of a change impact analysis tool on OO software:

- *Starting Impact Set (SIS)*: The set of objects that are thought to be initially affected by a change. In the context of this work, we'll consider that the SIS contains the objects that are actually impacted.
- *Estimated Impact Set (EIS)*: The set of objects estimated to be affected by the IA approach.
- *Actual Impact Set (AIS)*: The set of objects actually modified as the result of performing the change.

Ideally EIS and AIS should hold the same elements. Nevertheless, a tool can overestimate the impact (*i.e.*, $|EIS| > |AIS|$) or miss impacted elements (*i.e.*, $\exists e \in$ AIS such that $e \notin$ EIS). In the context of this document, we extend these definitions to "entities" which are not necessarily objects if the software analysed is not using the object-oriented paradigm.

With these definitions, the criterion assesses the experimental results that have been eventually gathered during the tool's experiments. Inside this criterion, it is possible to distinguish 4 sub-criterion:

1. *Size of Studied System* which can be expressed in different ways such as the number of lines of code (LOCs), the number of revisions, the number of changes or the number of changed files.

2. *Precision* which is defined as $\frac{|EIS|+|AIS|}{|EIS|}$ and determines how many identified elements are really impacted.

3. *Recall* which is defined as $\frac{|EIS|+|AIS|}{|AIS|}$ and determines how many impacted elements were detected by the approach.

4. *Time* which describes how much time it took to compute the results on average.

I.4 Discussion

This chapter motivates the need for Change Impact Analysis, defines the fundamental concepts associated and provides an overview the research field using Lehnert's taxonomy [15]. Using this taxonomy, it will be possible to position the approach proposed later in the document along the others existing approaches.

In its conclusion, Lehnert's taxonomy opened three research topics [16] listed and discussed below. These questions provide directions to explore in Change Impact Analysis research field and can be used as a basis for reasoning a proposition.

- *The lack of approaches covering the whole development process* (*i.e.*, requirements, model, code, etc...). This may be due to the fact that different techniques have to be applied depending on the part of the development process to be analysed. Therefore, to have a tool covering all the development process, combined techniques probably need to be used. Nevertheless, this topic is out of the scope of this work and will not be more developed.

- *The lack of a unified classification scheme for change types and dependency types.* Indeed, authors currently propose a new classification scheme for their approach which influences the impact analysis approach proposed. This observation may be due to the fact that approaches have specific needs of information about the changes. These specific needs are therefore translated as a specific change model for an approach. A similar reason can explain the lack of unified classification for dependency types. In Chapter III, a change model specific to the approach presented is defined and the use of dependency common to all software are discussed.

- *The lack of empirical validation of proposed ideas* (33% of them were not empirically evaluated). It probably comes from the fact that the Actual Impact Set for a change is unknown. The best thing we are able to do with Change Impact Analysis is to find an approximation of the Actual Impact

Set named the Estimated Impact Set. That being said, it is complicated to evaluate the the performance of a tool in terms of precision and recall. This research topic will be deeper explored in Chapter IV.

The next chapter is dedicated to the establishment of a state of the art concerning the two techniques used by the tool proposed in Chapter III. That is to say *Call Graphs* and *Program Dependency Graphs* techniques. Then, objectives for the tool are listed and discussed.

Chapter II

Call Graphs and Program Dependency Graphs

In this chapter, a review of techniques using *Call Graphs* and *Program Dependency Graphs* listed in Lehnert's taxonomy validation [16] is done in order to evaluate what has already been done and to inspire the tool developed later in this document.

Call Graphs and *Program Dependency Graphs* have been chosen to inspire the tool for multiple reasons. These reasons mainly come from the use cases presented in the Introduction of this document. First, the tool should be usable on a project that does not have preceding versions. Therefore technique requiring large data set such as *Probabilistic Models* and *History Mining* can not be used. Next, the tool should be usable without having to run the software analysed. Therefore, *Execution Traces* is not a practicable technique. The approach should be usable by someone that have low knowledge on the software so *Explicit Rules* is not suitable technique. Managing distributed systems and managing impact between elements at different levels of abstraction are not required features for the tool, thus *Message Dependency Graphs* and *Traceability links* will not be used. The results returned by the tool should just contain entity potentially impacted, not with a probability of being impacted. Therefore, *Information Retrieval* will not be used. Finally, the tool does not only targets the statement level of a program but more generally entities of the program thus, *Program Slicing* will not be used.

The remainder of this chapter is organised as follow. First, approaches using *Call Graphs* and *Program Dependency Graphs* are reviewed. Then the approaches are analysed to better understand what already exist. Finally, the objectives for the approach developed in the next chapter are defined, inspired by the analysis results.

II.1 Call Graphs

As described in the preceding chapter, *Call Graphs* (CG) are graphs built from methods and functions calls extracted from the code. These graphs allow to

perform impact analysis and some authors in the literature explored this data structure. This section review proposition of Call Graphs approaches listed in Lehnert's taxonomy.

II.1.1 Chianti

Ryder and Tip [23] created a method of change impact analysis implemented by Chianti [22], an impact analysis tool for Java systems developed by Ren et al. It works as follows. First, given two versions of a program and a set of tests covering a part of the program, it determines which tests have to be re-executed in after the change performed on the software. Ren et al. claim that their method generate an estimated impact set that at least contains the actual impact set (*i.e.*, $AIS \subseteq EIS$).

Second, from the impacted tests, the set of affecting changes is computed. These changes are those that may have given rise to a test's changed behaviour. The authors say their method is conservative because the set of affecting changes computed is guaranteed to contain at least every change that may have caused a change to a test behaviour.

Chianti works with model entities at the class, method, variable and test case granularity and works with the following changes: add/delete class, add/delete/change method and add/delete variable. The results provided by the tool is a set of test cases to be re-executed after a change has been applied.

II.1.2 Change Impact Dependency

Xia and Srikanth [25] proposed a theoretical method for change impact analysis based on a specific metric named *the change impact dependency*. One of the goals of this metric is to count the direct change impact as well as the indirect change impact (which, according to the authors, is not done by others metrics).

The metric assume that there is no information available about the type of change that was performed on the software. The granularity of this technique is at the statement level. The metric allows to select a set of lines of code that are potentially subject to be indirectly impacted by the change.

II.1.3 CCGImpact

Badri et al. [2] proposed an approach that mixes *Call Graph* (CG) and *Control Flow Graph* (CFG) approaches. The data structure they proposed is named a *Control Call Graph* (CCG) and contains more information than a CG. More formally, below are presented the definitions of oriented graph, call graph, control flow graph and control call graphs as defined in their article:

- *Oriented Graph*: An oriented graph $G = (S, A)$ is composed of a finite set S of vertex and a set of pairs $A \subseteq S \times S$ of arcs. Each arc $(x, y) \in A$ represents an oriented relation between two vertices between x as origin and y as target.

- *Call Graph*: A call graph is an oriented graph for which the set S contains vertices representing the methods of the software and the set A contains arcs representing the calls between methods. $(x, y) \in A$ represents the fact that x is the caller and y is the callee.
- *Control Flow Graph*: A control flow graph is an oriented graph for which the set S contains vertices representing decision points (if-then-else, while, case, etc...), an instruction or a sequential block of instructions (*i.e.*, a suite of instruction for which if the first instruction is executed, all the others instruction of the suite will be executed and always in the same order). An arc between two vertices $(x, y) \in A$ is present if it is possible to have an execution that leads to have y is executed after x has been executed.
- *Control Call Graph*: A control call graph is a control flow graph for which the vertices representing instructions not leading to method calls are eliminated.

The method takes the method that has been changed and propagates the change using the CCG independently of the type of change.

II.2 Program Dependency Graphs

Program Dependency Graphs (PDG) are graphs built from the dependencies between program entities. These graphs are built by analysing the source code of a software. This section review proposition of Program Dependency Graphs approaches listed in Lehnert's taxonomy.

II.2.1 Briand et al.

Briand et al. did an empirical study [7] that investigates the relation between coupling metrics and the ripple effect in object-oriented software. They identified that for two classes C and D the following metrics are relevant to predict the ripple effect:

- *PIM*: The number of method invocations in C of methods in D.
- *CBO'*: Coupling between object classes when there is no inheritance relations between the two classes. C and D are coupled together, if methods of one class use methods or attributes of the other, or vice versa. CBO is then a binary indicator, yielding 1 if C is coupled to D, else 0.
- *INAG*: Indirect aggregation coupling. Takes the transitive closure of the "C has an attribute of type D" relation into account.

To predict the impact of a change on a class C, the authors rank the classes of the system accord using the average of PIM, CBO' and INAG values with an equal weight for each metric. The more a class is high in the ranking, the more it is probable that it is impacted by the change.

From their impact model, Briand et al. conclude that using the metrics cited before indeed indicates classes with a higher chance to be impacted by a change

but also that some aspects of the ripple effect are not taken into account by the model. To have better results, the authors say that future research should try to provide new metrics that are not only based on the source code but also from all kind or requirement and design documentation.

II.2.2 Kung et al.

Kung et al. provide a tool [13] to evaluate the impact of changes in OO libraries. This tool is based on a formal model made for capturing changes and predicting classes affected by these changes.

The authors define a specific change model for their tool. This change model is based on the types of entities considered: *data change* that concerns global variables, local variables and class data members, *method change* that concerns methods, *class change* that concerns direct modification on classes and *class library change* that concerns modification on classes of the package or modification of the relationships between the classes of the package. Table II.2.2 lists all the changes defined by Kung et al.

Three types of graph are used by Kung et al. formal model: *Object Relation Diagram* which describes objects of the software and their relations, *Block Branch Diagram* (BBD) which model methods source code allowing to find which methods were modified by comparing the initial BBD to the BBD of the modified source code and *Object State Diagram* describing the state behaviour of a class. These are graphs generated from the source code and used to evaluate the impact of a change.

In the article, the authors talk about an experience with their tool indicating that it is extremely time consuming and tedious to test and maintain an OO software system. They also claim that they get good feedbacks from the developers that used their tools in the industry. Nevertheless, there is no description of the experimental protocol and there is no details about the experiment's results.

II.2.3 Ripples 2

The work of Rajlich [21] presents two approaches to compute change propagation implemented in a tool named *Ripples 2*. These approaches are based on a *dependency graph* that is built from the source code being analysed. This graph holds the entities of the software analysed and the dependencies between them.

To model a change on the software, the approach uses *graph rewriting*. That is to say, a change is represented as an algorithmic transformation of the initial graph to a target graph. Using this technique, the evolution of dependencies between entities of the software can be observed by comparing the graph transformations corresponding to the interested versions. Using this model, the two following approaches are proposed:

1. *Change-and-fix* approach which consists in two steps. First, the developer modify the software (apply a change on it) and second the change triggers

Components	Changes
data changes	• change data definition/ declaration/uses • change data access scope/mode • add/delete data
method interface changes	• add/delete external data usage • add/delete external data updates • add/delete/change a method call/a message • change its signature
method structure changes	• add/delete a sequential segment • add/delete/change a branch/loop
method component changes	• change a predicate • add/delete/change local data • change a sequential segment
class component changes	• change a defined/redefined method • add/delete a defined/redefined method • add/delete/change a defined data attribute • add/delete a virtual abstract method • change an attribute access mode/scope
class relationship changes	• add/delete a superclass • add/delete a subclass • add/delete an object pointer • add/delete an aggregated object • add/delete an object message
class library component changes	• change a class (defined attributes)
class library relationship changes	• add/delete a relation between classes • add/delete a class and its relations • add/delete an independent class

Table II.1: Change model proposed by Kung et al. tool [13].

a dependency analysis to evaluate the impact of the modification. This process is repeated until a feature has been implemented and the developer stop its modification.

2. *Top-down* approach starts by visiting top entities, that is to say entities that do not have dependencies with other entities. If no change is required in top elements, the approach starts examining entities that have dependencies between them.

Rajlich applies its tool on a object-oriented software with 19 classes and 2k lines of code. No information about the precision, recall nor the time taken by its tool is available in the article.

II.2.4 Pirklbauer et al.

Pirklbauer et al. [20] implemented a tool to support a change impact analysis process they defined, to visualize software artefacts, dependencies between them and metrics. This tool build a graph representing the software entities and the relations between them that the user can browse it (using a graphical user interface) to mark the entities correctly identified as impacted. Once the entities are marked, the developer has the set of entities to modify in order to integrate the change.

Even if an experiment to evaluate the tool has been set up, no information about its performance in terms of precision and recall is provided.

II.2.5 Zalewski and Schupp

Zalewski and Schupp [27] propose the concept of conceptual change impact analysis. This analysis interested in C++ standard template libraries (STL) modifications to assess the impact of conceptual specification changes on generic libraries.

The approach uses a pipes and filters mechanism. That is to say, initially in the analysis, all parts of the conceptual specification impacted by a change are detected. Then, the output of this detection is provided to different filters in order to refine the output and detect specific change impact. Two filters are presented. One detecting the impact of changes on the compatibility of different versions of concept specifications. The other detect the impact of a change on the degree of generality of an algorithm.

A case study is provided in the article but no experimental data as the size of the system analysed, the precision and recall of the tool nor the time it took to be run is provided.

II.2.6 Petrenko and Rajlich

Petrenko and Rajlich work [19] aims to extend change impact analysis to be able to work at variable granularity. To do that, they use an exploratory technique that visits a dependency graph built on the software for which nodes have been

marked changed, propagates, next, inspected or blank.

The developer explore this graph and when an impacted node is met, he/she has the possibility to let the impact be propagated at a finer or coarser grain level. The tool has been implemented as an extension for JRipple's integration in Eclipse[1] development environment.

Petrenko and Rajlich conducted a case study with the hypothesis that *Change Impact Analysis done on variable granularity is more precise than Change Impact Analysis done on the granularity of classes.* To test this hypothesis, they analysis the commit history of two open-source Java software. For each commit, they performed impact analysis at different granularities on the changes extracted. From these analyses, they extracted the precision and the recall of the results. The authors consider that the result of JRipple contains all the entities potentially impacted by the changes. Therefore, since their tool works on the results of JRipple, the recall of their tool is always 100%. The precision of the tool is thus evaluated as the number of entities that have actually changed on the number of entities detected as impacted by the tool. The precision of the tool were close but smaller than 19% on average.

The case study results suggest that using a finer granularity for the analysis tends to provide more precise results than change impact analysis using a coarser granularity.

II.2.7 Ripple Effect and Stability Tool

Black [6] is interested in reformulating Yau et al. ripple effect algorithm [26] and implemented an optimized version named *Ripple Effect and Stability Tool.* This algorithm uses a dependency matrix to analyse C code. On this dependency matrix, it uses software metrics to assess the impact of a change.

In the article, an evaluation is provided to show empirically that the algorithm proposed gives better results than the original algorithm proposed by Yau et al. Nevertheless, the authors do not provide any experimental result about the precision and recall of the tool.

II.2.8 Bilal et al.

Bilal et al. [4] are interested in the amelioration of Yau et al. ripple effect algorithm [26] as well. To do that, they propose new metrics based on *intramodule change propagation* which is the propagation from one variable to another within a module and *intermodule change propagation* which is propagation from one module to another. The authors propose 4 different algorithm implementing these change propagation and did an evaluation of them. Nevertheless, the authors do not provide informations about the results of these algorithms.

[1] `https://www.eclipse.org/`

II.2.9 ChAT

Lee et al. [14] use three kinds of graph describing relations between entities of object-oriented software. These graphs are used to compute the impact of a change on a software by performing transitive closure on them from the starting impact set. The authors also provide a set of metrics for object-oriented software that allow to evaluate the impact of a change.

The proposition of Lee et al. is implemented in ChAT, a tool that compute the impacts of changes on C++ programs. According to the authors, ChAT is flexible enough to be adapted to other programming languages. Nevertheless, such adaptations are not provided.

In the article, a case study is provided but no experimental results concerning the precision and recall of the tool.

II.2.10 Li and Offut

Li and Offut [17] analyse possible changes on object-oriented software and how these changes affect the classes. To compute this, they propose a set of algorithm that determine what classes will be affected by a change.

To do that, the authors use a Control Flow Graph. A nodes of this data structure represent a statement of the program or a sequence of statements and an edge represent a possible flow of control between two statements (or sequences of statements). Once the graph is built, a transitive closure is applied on it and its result combined with the types of relation between entities are used to assess the impact of a change on the software.

In the article, an example of the execution of the algorithm is provided but without any experimental result.

II.2.11 Static Execute After

Beszédes et al. [3] propose a new type of relation named *Static Execute After* that aims to determine the explicit and hidden dependencies (not expressed as explicit references in the source code) of entities in a software and to be less expensive than traditional dependency analysis techniques in computation time. This relation is used to find dependence between classes using their method. For the authors, a class B depends on a class A if and only if B may be executed after A in any possible execution of the program.

The graph formed by the entities of the program and the SEA relation allows to perform Change Impact Analysis. An experiment consisting in comparing the results of their tool with tools performing program slicing in order to compare their results. They concluded that their proposition is more efficient than the other tools evaluated. Furthermore, their experiment allows to assess the precision and recall of their tool by comparing its results with the results of the other tools. The precision varied between 65.5% and 96.5% where the recall was always of 100% (it was in fact $< 100\%$ for one of the tool but is was due to an

error made by the tool).

Additionally, the authors assess if the Coupling Between Object Classes (CBO) metric is suitable to evaluate whether a couple of classes have hidden dependencies. After the experiment, they observed that it was not always the case. Indeed, they found examples of classes with a low CBO metric but for which hidden dependencies were found by their tool. Thus, they concluded that this metric was insufficient to assess if there are or not hidden dependencies between two classes.

II.2.12 Static Execute Before

Jász et al. [11] extend the work of Beszédes et al. [3] and propose the concept of *Static Execute Before* (SEB) which is the dual concept of *Static Execute After*. Their article proposes an empirical comparison of SEA/SEB techniques combined performances and traditional methods to do change impact analysis performances.

Their experiment shows that using SEA/SEB techniques results in a better scalability. Indeed, the computation time decrease. Nevertheless, this better scalability comes with a cost of 4% less precision on average. The experimental results in terms of precision and recall are the following. The precision range from 67% to 98.77% where the recall is always 100%.

II.2.13 RIPPLES

Chen and Rajlich [8] introduce a Change Impact Analysis tool (RIPPLES) that uses the Abstract System Dependence Graph (ASDG) (*i.e.*, a graph which represents dependencies among components of the software) of the program to which are added conceptual dependencies (*i.e.*, which are dependencies that are not detectable using static analysis of the software's source code).

RIPPLES extract the ASDG from C source code. Nodes of this graph can be program variables, methods, and data types. Next, the ASGD is reduced by removing unnecessary nodes (*e.g.*, nodes containing statement with no function call) and the user can inspect the graph. Then, the nodes of this graph are marked with a tag depending of the state in which they are unmarked (default), candidate, visited, located or unrelated (initially, all nodes are marked as "unmarked"). Edges are also marked with unmarked, backward or forward depending on the search direction that should be taken.

A developer that visits the ASDG updates it by changing the nodes/edges tags in order to find the set of entities impacted by the change. RIPPLES allows the developer not to get lost during the impact estimation by providing it a picture of the change impact analysis being done.

A case study were done by the authors and experimental results are provided. To introduce their results, Chen and Rajlich present the three following metrics:

- *Recall* defined as:
$$\frac{\text{The number of relevant edges suggested by RIPPLES}}{\text{The number of relevant edges}}$$
- *Theoretical Precision* defined as:
$$\frac{\text{The number of relevant edges suggested by RIPPLES}}{\text{The number of edges suggested by RIPPLES}}$$
- *Realistic Precision* defined as:
$$\frac{\text{The number of relevant edges suggested by RIPPLES}}{\text{The number of edges suggested by RIPPLES and investigated}}$$

During their case analysis, the authors found the following results:

- *Recall* = 100%
- *Theoretical Precision* = 7.69%
- *Realistic Precision* = 93.10%

which suggests that the tool provides more edges than necessary for the exploration but does not miss any relevant edge.

II.2.14 JTracker

Gwizdala et al. [10] propose a tool that helps developers to integrate changes in Java software using change impact analysis. This tool, JTracker, scans the source code of Java program in order to find dependencies between entities and stores them. It is integrated in JBuilder Java environment as an extension. JBuilder assists the user during the development by, whenever an entity change, marking the neighbours entities in terms of dependencies to be visited. Indeed, when an entity is modified in the system, the entities having a dependency relation with them should be checked by the developer.

In their article, the authors did a case study on a Java software but did not provide any results on the precision and recall of the tool.

II.2.15 Bishop

In his master thesis [5], Bishop proposes an incremental software change impact model. Its proposition is motivated by a need of a compromise between precision and computation time required to analyse the impact of a change on a software. To achieve that, his model reuse the preceding results of analysis when performing a new analysis.

A program dependency graph is built from the initial change impact analysis and is stored. Then, each time an analysis is performed, the preceding version of the graph is updated according to the new software version and the impact is computed incrementally from the preceding version.

The author evaluated the performance of the algorithm in terms of computation time by benchmarking it but no indication on its precision and recall is given.

II.3 Analysis on Articles Reviewed

This section analyses and discusses the articles summarized previously. The Table II.4 at the end of this chapter lists the approaches analysed and their criteria in Lehnert's taxonomy. In this table, a dash means that it was impossible to assess the criterion for the approach (for example because the information was not in the article). The columns of the table concerning experimental results contains the metrics in average (for example, some articles provide multiple measures for precision, in the table the average of these measures is shown).

The same type of entity is observed in different approaches for *entities granularity* criterion. The Figure II.1 shows the support of entities granularity of the approaches (for approaches for which the granularity of entities can be evaluated). The *entities granularity criterion* could be assessed for 17/18 approaches. The most commonly supported entities are method, class and variable. Less approaches support statement, specification and test case.

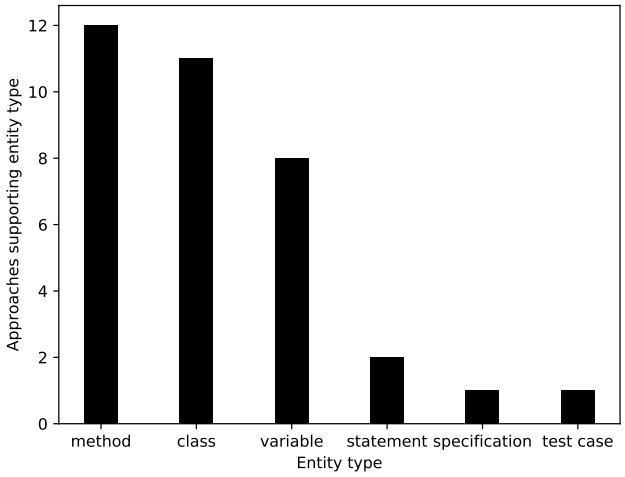


Figure II.1: Types of entities supported by approaches and the number of approach(es) supporting them.

Some types of changes are also used recurrently by the approaches analysed. The Figure II.2 shows the support of change types by the approaches. Method, class and variable addition/deletion are the most common change types. Then come the others change types with 1 or 2 tools supporting them. The *changes granularity* criterion could be assessed for 7/18 approaches.

The results of the approaches reviewed previously contain the entities impacted by the change on the software and for one tool, the specification impacted by the change on specification proposed. Figure II.3 shows the types of results supported by the approaches. The most supported result type is class, then method and variable. Statement is provided as result by two approaches and test case by a single approach. The *results granularity* criterion could be assessed for 17/18 approaches. All approaches for which the criteria could be

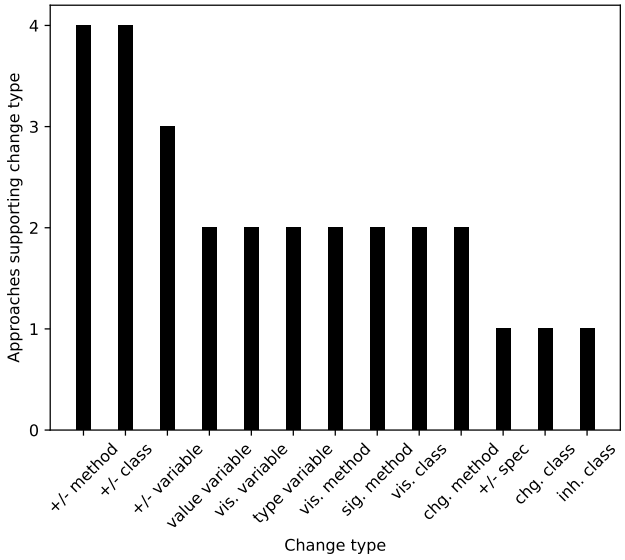


Figure II.2: Types of changes supported by approaches and the number of approach(es) supporting them.

assessed only provide the entities impacted as result and not how they are impacted.

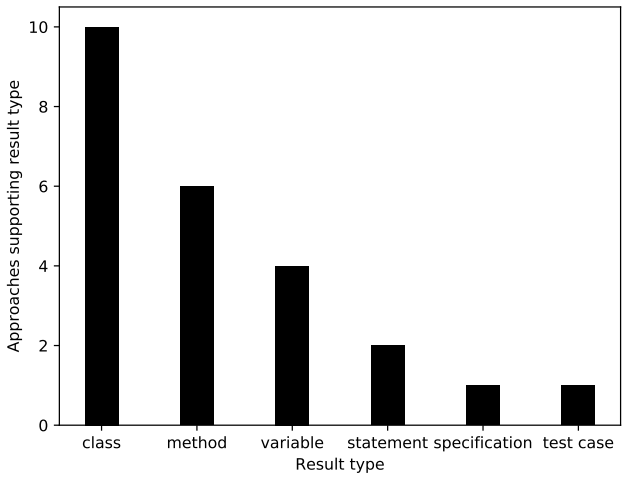


Figure II.3: Types of results supported by approaches and the number of approach(es) supporting them.

Figure II.4 shows the languages supported by approaches. This criterion could be assessed for 15/18 approaches. The most supported languages are C++ and Java which is not surprising since they are widely spread technologies. The two paradigm supported by these approaches are the object-oriented

paradigm and the procedural paradigm.

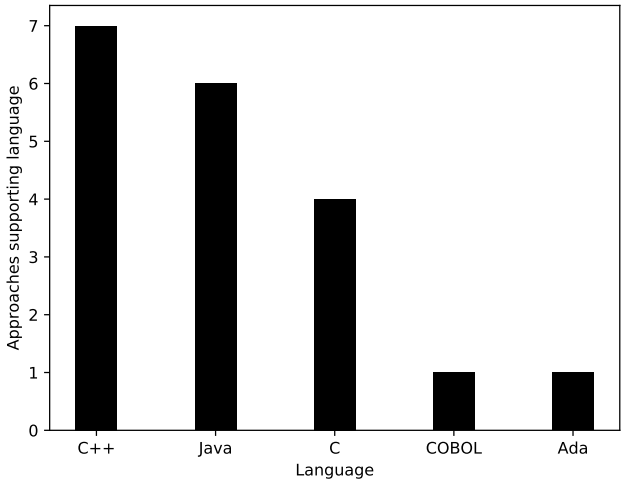


Figure II.4: Languages supported by approaches and the number of approach(es) supporting them.

Most of the approaches got implemented. Indeed, 14/18 approaches have at least one tool implementing them. Nevertheless, the presence of this tool does not ensure the presence of experimental results. Indeed, on the 18 approaches, only one has all the experimental results required by the taxonomy, 12 of them give a part of the results required (where 7 only provide the size of the system) and 5 approaches do not provide experimental results at all. Only 3 approaches provide an estimation of their worst-case complexity and only 4 approaches allow to assess the style of analyse they use (and it is always *exploratory*).

II.4 Objectives for the Implementation

The analysis made in the previous section allows to define objectives for the approach presented later in this document. Indeed, the review of these approaches highlights some of their weakness that are appearing regularly in their design.

1. *Software Paradigm Independence*: Ideally, the tool should be as independent of the software paradigm as possible. This objective could be achieved using relations between entities that are common to all software or by providing a way to specialize the framework to any software paradigm. With such an objective fulfilled, the approach could be used in more kinds of software (*e.g.*, software using functional paradigm).

2. *Software Language Independence*: Since the software paradigm independence should be assured, the software language independence should be supported as well. Indeed, as observed in the previous section, most of the approaches are working on C++ and Java software. The development of Change Impact Analysis tools for other languages is required as well. This

could be done using a modelling framework and to work on an abstract model of the software.

3. *Model Independence*: The type of model used to represent the software the approach should not constrain the its extensibility. In order to achieve this, the possibility to adapt the approach to different model should be provided.

4. *Extensible Change Types*: As observed during the review, there are change types that are common to all software (*e.g.*, *add*, *remove*, etc...). These change types should be implemented by default in the tool. Nevertheless, depending on the software paradigm and language used, the possible change types may vary. For that reason, the change types should be extensible.

5. *Use Case Independence*: The approach should let the possibility to adapt it for multiple use cases. Indeed, as discussed in the preceding chapter, an impact analysis approach could be used for cost estimation, refactoring software or selecting unit tests to re-run after a change has been integrated. The review of articles confirm the existence of multiple use-cases for Change Impact Analysis since some of the approaches are used to help the developers to integrate a change and others to find the unit-tests to re-execute.

6. *Richer Results*: All the approaches analysed only provide the entities impact by the change given as parameter. In their results, no more information is provided. Nevertheless, it should be possible to provide more information in the results such as for example *how* the entities are impacted by the original change.

Approach	Technique	Entities Granularity	Changes Granularity	Results Granularity	Tool Support	Supported Language	Scalability	Style of Analysis	Size	Precision	Recall	Time
Ryder, Tip and Ren et al [23, 22]	CG	class, method, variable, test case	+/- class, +/-/chg. method, +/- variable	test case	Chianti	Java	-	-	123 kLOC, 11k changes	-	1.0	10 min
Xia and Srikanth [25]	CG	statements	-	statements	no	-	-	Expl.	-	-	-	-
Badri et al. [2]	CG	method	chg. method	method	PCIA Tool	Java	-	-	77 classes	-	-	-
Briand et al. [7]	PDG	class	-	class	Concerto 2/AUDIT	C++	-	-	40 kLOC	-	-	-
Kung et al. [13]	PDG	class, method, variable	+/-/inh./vis. class, +/-/sig./vis. method, +/-/type/vis. variable	class, method, variable	OOTME	C++	-	-	$>$ 140 classes	-	-	-
Li and Offut [17]	PDG	class, method, variable	+/- class, +/-/sig./vis. method, +/-/type/val./vis. variable	class	no	-	$O(m^3n^2)$	-	-	-	-	-
Rajlich [21]	PDG	class	+/-/chg. class	class	Ripples 2	C, C++	-	Expl.	2 kLOC	-	-	-
Pirklbauer et al. [20]	PDG	-	-	-	CIAMSS	COBOL	-	-	-	-	-	-
Zalewski and Schupp [27]	PDG	STL spec.	+/- STL spec.	STL spec.	no	C++	-	-	-	-	-	-
Petrenko and Rajlich [19]	PDG	class, method, statement, variable	-	class, method, statement, variable	plug-in for for JRipples	Java	-	Expl	550 kLOC	$<$ 0.19	-	-
Black [6]	PDG	variable	-	variable	REST	C	-	-	725 kLOC	-	-	-
Lee et al. [14]	PDG	class, method	-	class	ChAT	-	-	-	30 kLOC	-	-	-
Beszédes et al. [3]	PDG	class, method	-	class	-	C++, Java	$O(n*e+n*k*m)$	-	400 classes	0.85	1.0	-
Bilal et al. [4]	PDG	class, method	-	class, method	REST, CodeSurfer	C++	-	-	-	-	-	-
Jász et al. [11]	PDG	method	-	method	CodeSurfer	C, C++, Ada	$O(n+e)$	-	1.4 mLOC, 83k methods	0.87	1.0	3h
Chen and Rajlich [8]	PDG	method, variable	-	method, variable	RIPPLES	C	-	Expl.	-	0.09	1.0	-
Gwizdala et al. [10]	PDG	class, method, variable	-	class	JTracker	Java	-	Expl	400 classes	-	-	-
Bishop [5]	PDG	class, method, variable	+/- method	class	Incremental Impact Analyzer	Java	-	-	9 kLOC	-	-	350ms

Table II.2: Summary of the review of approaches (inspired from Lehnert table [18]).

Chapter III

Proposition

This chapter introduces an approach that can be specialized to perform change impact analysis on any kind of software. Because of this, the approach is said to be generic. It has been implemented in the Pharo[1] environment and in the context of this work, it has been specialised to analyse Pharo software.

The remainder of this chapter is organised as follow. First, important concepts used by the approach are defined. Next, the change model and the results model are presented. Then, the approach and its implementation in Pharo are described. Finally, the approach and the implementation are discussed.

III.1 Entities and Relations

In the context of this work, an entity is defined as a part of the software structure. The meaning of entity depends on the model extracted from the software analysed. For example, if a Java program is modelled, the result of this modelling can be an UML diagram representing its structure. In this case, an entity would be an element of this diagram. If the result of the modelling is an abstract syntactic tree, therefore an entity would designate a node of this tree.

Between these entities, there are relations. These relations define how the software is organised. The approach proposed later in this chapter makes the assumption that the software model analysed has at least the two following relations. The first one is the *composition* which allows to compose entities. That is to say the possibility to combine entities into more complex entities. The second is the *reference* between entities. That is to say, an entity can point to another entity eventually without being composed of it. Two of the objectives defined in the Chapter II were the *Software Paradigm Independence* and *Software language Independence*. The use of these two special relation allows to achieve these objectives.

Without loss of generality, the *inheritance* relation is supported by the approach. This support does not modify the generic nature of the approach since if a paradigm or a software analysed does not support inheritance, the relation

[1] http://pharo.org

will simply not be used. It is important to support this relation because it is not only available in object oriented software. Indeed, for example PostgreSQL (a database management system) defines an inheritance relation between tables[2].

To wrap these entities and relations, we define the concept of *software model*. Thus, in the rest of this document, this term is used to designate this wrapper.

III.2 Change Model

The approach presented uses a change model made of five kinds of change:

- *Add* change represents the fact that an entity has been added to the software.
- *Remove* change represents the fact that an entity has been removed from the software.
- *Move* change represents the fact that an entity has been moved in the model. A movement in the model is defined as a modification of the container of the entity (*e.g.*, for object-oriented software a class is moved if the package holding it has changed).
- *Rename* change represents the fact that an entity has been renamed in the model. This change can only be applied on entities that are named.
- *Body* change represents the fact that an entity had its body modified. This change is only applicable on entity that have a body (*e.g.*, in object-oriented software, a method has a body but an instance variable does not have a body). This change type allows to avoid that the analysis works at an finer granularity. That is to say, if the work is done at the package/class/method/instance variable level, it may not be wanted to have impact at the abstract syntactic tree level.

More specific changes could be used. For example, in the OO paradigm, the notion of inheritance is defined. Because this notion exists, additional changes can be used such as for example some of those identified by Flower [9]:

- *Pull up entity* change represents the fact that an entity for which the container is subject to inheritance is moved to the mother entity in terms of inheritance. For example, an instance variable or a method that belongs to a class can be moved to the mother class of the class they belong to.
- *Push down entity* change represents the opposite of *pull up entity*. Indeed, an entity belonging to a container entity subject to inheritance is moved to one or multiple of the children entities in terms of inheritance. For example, an instance variable or a method that belongs to a class can be moved to one or multiple of its children class of the class they belong to.
- *Modify parent entity* change represents the fact that one of the super entity in the inheritance chain of an entity subject to inheritance is replaced by another one.

[2] https://www.postgresql.org/docs/current/static/tutorial-inheritance.html

Indeed, these additional changes can be expressed as one or a sequence of *add*, *remove*, *move*, *rename* and *body* change. Nevertheless, by expressing these additional changes as new change types allows to know the intention of the author performing the change in order to compute its impact.

In the context of this work, the five first change types were implemented. That is to say: *add*, *remove*, *rename*, *move* and *body*. Since one of the objectives defined in Chapter II were the *extensibility of change types*, the implementation proposed below let the possibility to extend these five changes in order to obtain a finer analysis for specific change types.

III.3 Results Model

In the approach presented below, each element of the resulting set stores the following information:

1. The entity computed as impacted by the original change;
2. The type of change that should be performed on the impacted entity in order to to integrate the original change;
3. Whatever the change has been induced by another change of the resulting set or not; and
4. The changes that have been induced by the changed entity.

III.4 Approach Description

This section describes the approach at a high level of abstraction. The pseudo code of the algorithm is provided and described. This section presents the approach naively in order to make it easier to understand. Nevertheless, if the approach was implemented as described it would consume unnecessary memory. For this reason, the next section present an optimized version of the approach.

III.4.1 Description

The general process used by the proposition is composed of the two following steps:

1. From a change concerning an entity and a software model, a directed graph is built. The nodes of this graph contain the entities and the edges represent the relations between these nodes.
2. The graph is used to compute the impact of a change on one of its entities by exploring it according to a strategy depending on what the user wants to achieve.

The first step is highly dependent of the software analysed and the modelling language/framework used. Indeed, the approach used to build the graph will be influenced by the paradigm used by the software and its language. On the other hand, the framework used to model the software in memory in order to

be manipulable algorithmically can also influence the building of the graph. For these reasons, it will not be detailed here.

The pseudo code describing the exploration of the graph (step 2.) can be written as in Algorithm 1. This algorithm roughly describes a breadth-first exploration of the graph starting from the initially changed entity. Lines 2 to 4 initialize the first-in-first-out (FIFO) queue which holds the next changes that the while loop at line 5 will treat. The while loop from line 5 to line 18 contains the core of the algorithm. The next change to process c' is extracted from the FIFO and depending on the change type of c', one of the function to process the change is called. Theses functions: *processAdd*, *processRemove*, *processMove*, *processRename* and *processBody* depend on the goals of the user (their implementation may vary depending on the use-case). Finally, when there are no more changes to process in *toProcess* FIFO queue, the *impacted* set is returned (line 19).

ALGORITHM 1: Algorithm that computes the Estimated Impact Set of a change on a software.

Input: A graph built from a software model s and a change c on it.
Output: The set of changes induced by the change c.

```
1:  procedure impact(s, c)
2:      toProcess := new FIFO
3:      toProcess.add(c)
4:      impacted := {c}
5:      while s.toProcess ≠ ∅ do
6:          c' := toProcess.next()
7:          switch c do
8:              case Add:
9:                  processAdd(c', s, toProcess, impacted)
10:             case Remove:
11:                 processRemove(c', s, toProcess, impacted)
12:             case Move:
13:                 processMove(c', s, toProcess, impacted)
14:             case Rename:
15:                 processRename(c', s, toProcess, impacted)
16:             case Body:
17:                 processBody(c', s, toProcess, impacted)
18:     end while
19:     return impacted
20: end procedure
```

To define the functions called in the switching statement of Algorithm 1, the following informations are needed:

1. The *goals* the developer wants to achieve with the impact analysis. A change impact analysis can be done in different context as shown by the use cases presented in Chapter I. As a remainder, the three examples of use cases given previously were: estimating the cost of a change, helping in the modification a software and selecting tests to re-execute after a change

has been applied. The goals to achieve will influence the exploration of the graph.

2. The *type* of change that affects the entity. Depending on the type of change, the impact will spread (or not) to the entities connected to the entity via composition and references.

The goal of the implementation proposed in the context of this work is to help developers to evaluate the impact of a change they need to perform. To do this, the approach does not propagate the impact of *body* change. Indeed, the idea behind this is that when there is a body change, the developer should manually analyse the entity to eventually adapt it to the change. For such a strategy, the functions to implement in Algorithm 1 have the following behaviours:

- *processAdd*: An *add* change concerning an entity will have no detectable impact on the software except if the entity added is subject to inheritance and override the behaviour of its super-entity. In this case, all the references to the super-entity must be checked by the developer in order to ensure that the behaviour of the parts of the software using it keep the behaviour expected. This behaviour is implemented in Algorithm 2.

ALGORITHM 2: Algorithm to update *toProcess* queue and *impacted* set according to an add change.

Input: A change c on a software model s, a fifo queue $toProcess$ and the estimated impact set $impacted$.
Output: Nothing, updated $toProcess$ and $impacted$.

```
1:  procedure processAdd(c, s, toProcess, impacted)
2:      if hasInheritanceSuperEntity(c.e) then
3:          for r ∈ s.referencesTo(c.e.super) do
4:              c' := new Body
5:              c'.entity := r
6:              toProcess.add(c')
7:              impacted.add(c')
8:          end for
9:      end if
10: end procedure
```

- *processRemove*: A *remove* change concerning an entity e will have the 2 following effects. First, all the references to the removed entity will have to be checked in order to remove the reference to the entity e. Second the entities contained are marked as removed and put in the queue of entities to treat at the next iteration of the algorithm. In Algorithm 3, the first step is implemented from line 1 to 7 and the second step is implemented from line 8 to line 13.

- *processMove*: A *move* change concerning an entity e to put in the new container k has the effect to impact all the body of all entities referencing e. Indeed, these bodies needs to be updated in order to adapt them to the move of the entity. The Algorithm 4 implements this behaviour.

ALGORITHM 3: Algorithm to update *toProcess* queue and *impacted* set according to a remove change.

Input: A change c on a software model s, a fifo queue *toProcess* and the estimated impact set *impacted*.
Output: Nothing, updated *toProcess* and *impacted*.

```
1:  procedure processRemove(c, s, toProcess, impacted)
2:      for r ∈ s.referencesTo(c.e) do
3:          c' := new Body
4:          c'.entity := r
5:          toProcess.add(c')
6:          impacted.add(c')
7:      end for
8:      for e ∈ s.containedEntities(c.e) do
9:          c' := new Remove
10:         c'.entity := e
11:         toProcess.add(c')
12:         impacted.add(c')
13:     end for
14: end procedure
```

ALGORITHM 4: Algorithm to update *toProcess* queue and *impacted* set according to a move change.

Input: A change c on a software model s, a fifo queue *toProcess* and the estimated impact set *impacted*.
Output: Nothing, updated *toProcess* and *impacted*.

```
1:  procedure processMove(c, s, toProcess, impacted)
2:      for r ∈ s.referencesTo(c.e) do
3:          c' := new Body
4:          c'.entity := r
5:          toProcess.add(c')
6:          impacted.add(c')
7:      end for
8:  end procedure
```

- *processRename*: A *rename* change concerning an entity e will impact all the body referencing e. In all these bodies, the reference(s) to e need(s) to be renamed according to the new name. The Algorithm 5 implements this functionality.

ALGORITHM 5: Algorithm to update *toProcess* queue and *impacted* set according to a rename change.

Input: A change c on a software model s, a fifo queue *toProcess* and the estimated impact set *impacted*.
Output: Nothing, updated *toProcess* and *impacted*.

```
1: procedure processRename(c, s, toProcess, impacted)
2:     for r ∈ s.referencesTo(c.e) do
3:         c' := new Body
4:         c'.entity := r
5:         toProcess.add(c')
6:         impacted.add(c')
7:     end for
8: end procedure
```

- *processBody*: A *body* change concerning an entity requires the developer to check it. Therefore, in this context, it does nothing. The pseudo-code of this algorithm is therefore not presented since it is pointless.

An important remark about the use of *add* on *toProcess* and *impacted* is that this function add a change of type t concerning an entity e to respectively the FIFO queue and the set if and only if the couple (t, e) is not already present in *impacted* or in *toProcess*. This behaviour implies that a change of a certain type concerning an entity is treated only once.

III.4.2 Example

As an example, let's take the software illustrated in Figure III.1. Additionally to the information available in this UML diagram, we precise the following things: $A.m1$ is invoked by $A.m2$, A is referenced by the method $C.m6$ and B references A since it is its superclass. Let's say we want to remove the class A to another package. The graph built from this software model is illustrated in Figure III.2.

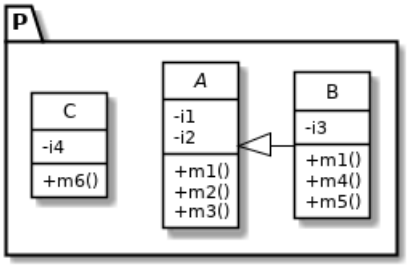


Figure III.1: UML diagram of a software created for illustration purposes.

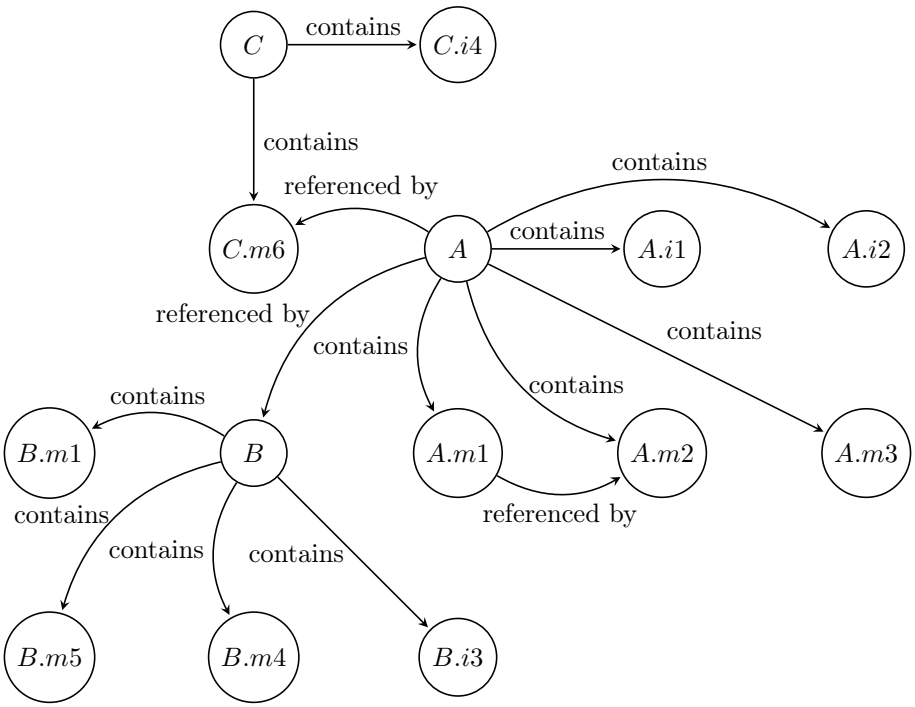


Figure III.2: Graph built from the software model illustrated in Figure III.1.

The result of the algorithm on this graph with the change $Remove(A)$ would provide the following result: $\{Remove_d(A)$, $Remove_i(A.i1)$, $Remove_i(A.i2)$, $Remove_i(A.m1)$, $Remove_i(A.m2)$, $Remove_i(A.m3)$, $Body_i(A.m2)$, $Body_i(B)$, $Body_i(C.m6)\}$. A small d after the change name means that the change is due to the developer action where a small i means that the change has been induced by another change. The Figure III.3 provides a graphical representation of the result set. An arrow from a change to another change means that a change induced the other change.

III.4.3 Memory-Optimized Approach

As illustrated in the precedent example, some parts of the graph may not be visited during the impact computation. In this example, the nodes containing the entities C, $C.i4$, $B.i3$, $B.m1$, $B.m4$ and $B.m5$ are not visited by the algorithm. Therefore, for the computation of the impact of removing the entity A the information is stored in the graph while being useless.

Thus, building directly the graph used by the algorithm is a bad idea in term of memory consumption. Instead, it is possible to simply explore the model entities and its relations without explicitly building the graph. To perform that, the way to explore the model has to be defined algorithmically. This is what is done by the tool proposed in this report.

Therefore, in the optimized approach, instead of defining a graph builder taking a software model as input and producing a graph as output, an abstract

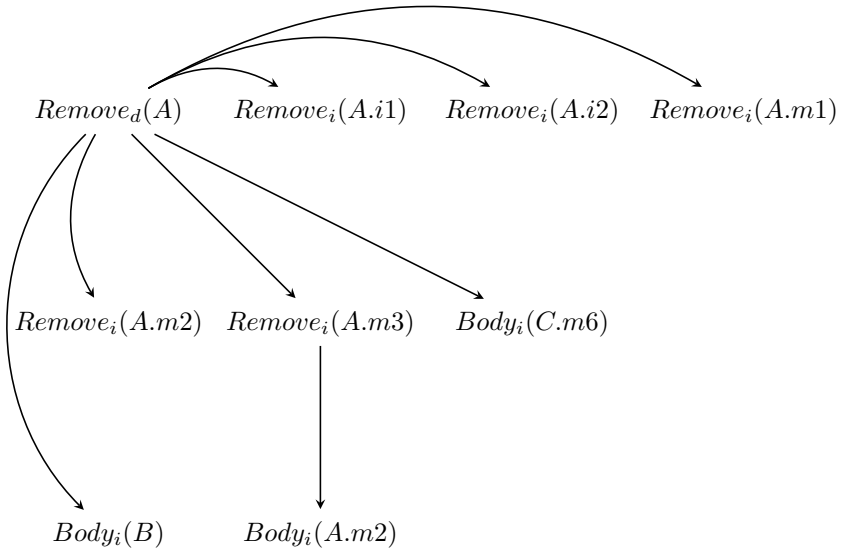


Figure III.3: Impact objects resulting from the computation of the impact of *Remove(A)* from the software associated to the graph illustrated in Figure III.2.

algorithm to explore the model is designed. In order to browse a specific model (and thus being able to compute the impact of a change on it), methods helping to browse it have to be implemented. That is to say, in the Algorithms 2, 3, 4 and 5 where $referencesTo$ and *containedEntities* were simply accessors of the graph's node, these methods are re-implemented to compute $referencesTo$ and *containedEntities* directly from the entities of the model.

III.5 Implementation

This section describes the implementation of the tool in the Pharo environment. Pharo has been chosen as a platform for the implementation of the tool because there are frameworks available for it that make software modelling and analysis easier. Additionally to the fact that the tool is implemented in Pharo, it has been specialized to compute the impact of a change on a Pharo entity while being implemented in a way that let the possibility to specialize it for another language if needed.

In this section, the implementation of the change and the result models are presented. Next, the concept of filter is presented as well as its implementation. A filter allows to add rules concerning the impact propagation. Finally, the implementation of the strategies to browse the model and to achieve the goals a developer wants to achieve are presented.

III.5.1 Change Model

The Figure III.4 is the UML diagram of the change model. *Change* is the abstract object representing a change. It simply defines the fact that a change

holds an entity.

The subclass of *Change* are the concrete possible changes for impact analysis. They are similar except for *RenameChange* that stores the new name additionally to the entity concerned and *MoveChange* that stores the new container of the entity additionally to the entity concerned.

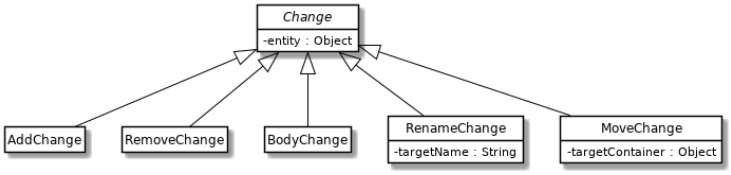


Figure III.4: UML diagram of the changes model.

III.5.2 Results Model

As explained before, the results model allows to hold information about an element in the set resulting of the algorithm: whatever the element is caused by another element or by the developer, the entity impacted, the type of change to apply on the entity in order to keep the software behaviour in the same state as before the change and the elements induced in the resulting set.

The Figure III.5 is the UML diagram of the results model. The concept of *Impact* is defined as what holds the information an element of the set resulting of the algorithm execution hold. Therefore, an *Impact* object holds a change and a list containing the *Impact* objects they induce. The fact that the impact is induced by an other *Impact* object or not is encoded by its class (*DirectImpact* or *IndirectImpact*).

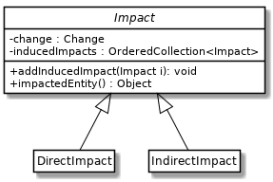


Figure III.5: UML diagram of results model.

III.5.3 Filters

Filters have been introduced in the framework in order to gain control on the software model exploration without having to modify the strategy used (see next sub-section). A filter is an object implementing methods that allows the strategy to decide if it has to process a change or not by implementing a method named *shouldChangeBeFiltered* taking a change as parameter and returning true if the change has to be filtered, else false. It is used at the moment a change has to be put in *toProcess* FIFO queue. Instead of directly putting the change in the queue, the filter is used to test if the change has to be filtered or not.

Depending on what *shouldChangeBeFiltered* returns, the change is either added to the queue or not.

Figure III.6 shows the UML diagram of the filters implementation. *Filter* is an abstract superclass that has to be subclasses by all filters. *NullFilter* is the implementation of the null-object design pattern. It allows to initialize strategies with a default filter that rejects no change. Therefore, *shouldChangeBeFiltered* always return false.

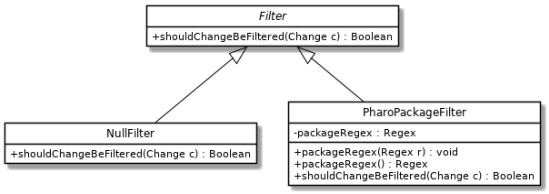


Figure III.6: UML diagram of filters.

A specific filter for Pharo software analysis named *PharoPackagesFilter* has been implemented. It allows one to limit the impact analysis to a set of packages by providing a regular expression that selects the packages to analyse only if their names match the regular expression. This filter is really useful in the case of Pharo because of the fact it is dynamically typed and the whole system source code is available. The first fact implies that when there is a method call, it is not possible to know from which class implementing a method with the same signature it comes from. The second fact implies that the analysis may explore uninteresting parts of the system that the developer knows they can not be impacted by its change.

III.5.4 Model Strategies

The way to explore the model is implemented following the strategy design pattern. This design pattern allows to easily change the behaviour of an algorithm which is required for the tool implementation. Indeed, this allows to provide an algorithm independent of the software model used.

Figure III.7 shows the UML diagram of model strategies. The abstract class *ModelStrategy* defines the common behaviour that all its subclasses have to override. This common behaviour is represented by the following functions: *containedEntities* which returns the entities contained in the entity given as parameter, *containerEntity* which returns the entity containing the entity given as parameter, *inheritanceSubEntities* which returns the inheritance super entities of the entity given as parameter is the entity is subject to inheritance or an empty collection if it is not, *inheritanceSuperEntities* which returns super entities of the entity given as parameter if the entity is subject to inheritance or an empty collection if it is not, *isSubjectToInheritance* which returns true if the entity given as parameter is subject to inheritance and false if it is not and *referencesToEntity* which returns the entities referencing the entity given as parameter.

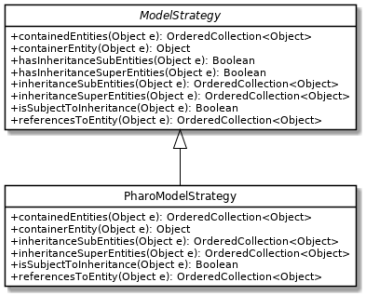


Figure III.7: UML diagram of model strategies.

The methods *hasInheritanceSubEntities* and *hasInheritanceSuperEntities* return true of false depending if the collection resulting from the call of respectively *inheritanceSubEntities* and *inheritanceSuperEntities* returns a non-empty collection or an empty collection. *PharoModelStrategy* class implements the abstract methods of its super class in order to allow the tool to explore a Pharo software.

III.5.5 Goal Strategies

Goal strategies allow to define a specific way to perform impact analysis. In the context of this work, a "developer strategy" has been developed. As written before, this strategy aims to help the developer of a software to evaluate the impact of a change to perform before performing it. Nevertheless, the tool has been developed in a generic way in order to allow one to create a new goal strategy. For example, it may be needed to develop a strategy allowing to find the tests to re-execute if a change is performed (or after a change has been performed). Goal strategies are as model strategies an implementation of the strategy design pattern.

Figure III.8 shows the UML diagram of the implementation of change impact strategies. The *impactForChange* method implements the Algorithm 1 but instead of using a switch it used dispatching on change objects in order to make the potential add of a type of change easier. The method call for dispatching is *impactForStrategy* and is defined in each change class. Depending on the change class, one of the following methods of *Strategy* is called: *impactForAddChange*, *impactForRemoveChange*, *impactForMoveChange*, *impactForRenameChange* and *impactForBodyChange*.

In order to know how to explore the software model being analysed, the *modelStrategy* instance variable of *GoalStrategy* class stores a *ModelStrategy* instance allowing to browse the software model given as parameter to *impactOfChanges*. The *filter* instance variable stores a *Filter* allowing to decide if a change should be treated or not during the exploration of the software model.

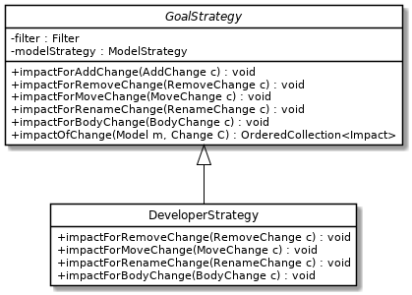


Figure III.8: UML diagram of goal strategies.

III.5.6 Graphical User Interface

A graphical user interface has been developed in order to make the tool more user-friendly. It is composed of two parts. First, a widget to let the user explore the result of a change impact analysis and second, an integration of the widget in the integrated development environment of Pharo.

The integration of the tool in Pharo integrated development environment is made by adding an item to the contextual menu of the class browser as shown in Figure III.9. This item is named "What if..." and provide a sub-menu allowing the user to choose the change for which the impact has to be computed on the software. This contextual menu appears when right-clicking on the packages list (1), on the classes list (2) and on the methods list (3). Depending on what list is clicked by the user, the change impact analysis will be performed on a package (1), a class (2) or a method (3). To analyse the impact of a modification on an instance variable, an additional menu item is available in the contextual menu of the classes (2).

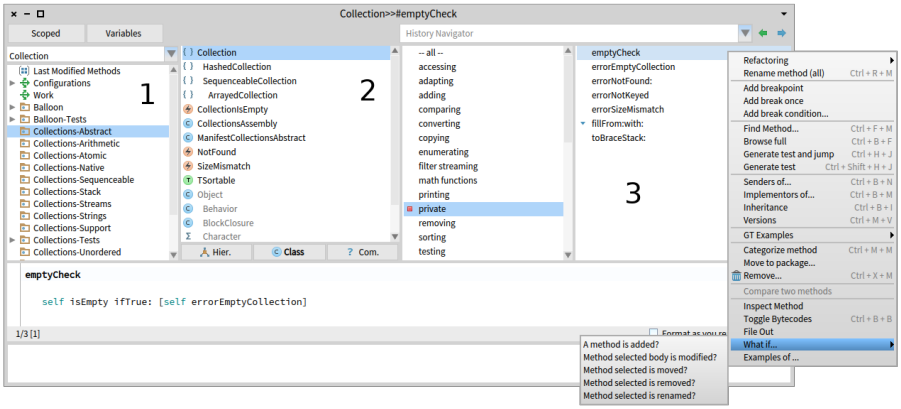
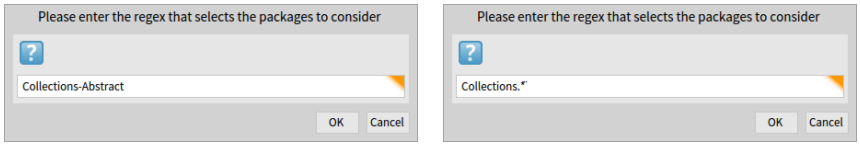


Figure III.9: Screenshot of the contextual menu that has been integrated in Pharo integrated development environment. The contextual menu appears when right-clicking on the packages list (1), on the classes list (2) and on the methods list (3).

Once the user has chosen a change to perform on the entity selected, a dialogue box appears asking the user to enter the regular expression that the *PharoPackagesFilter* should use. As illustrated in Figure III.10, by default the regular expression provided only select the package in which is located the entity selected. In our example, we compute the impact of renaming *emptyCheck* method of *Collection* class. Since this method is in the private protocol, we know that only subclasses of *Collection* class will use it. Therefore, we enter a regular expression that will allow to check the impact of this change in all package with a name beginning with "Collections".



(a) Regular expression proposed by default, the name of the current package.

(b) Regular expression used to allow the analysis to check all "Collections" packages.

Figure III.10: Screenshots of the dialogue appearing to configure *PharoPackagesFilter*.

Finally, the impact of the change inside the packages selected by the regular expression is computed. Once the computation is complete, a browser is opened on the result. Figure III.11 shows this widget made of three main parts. Part (1) is a tree list containing the result of the change impact analysis. When an impact of this tree is unfolded, its induced impacts appear. Selecting an impact in the tree list has two effects. First it displays the source code of the entity concerned by the change in the text panel (2). Second it modify the selected entity in the tree visualisation panel (3). In this visualisation panel, the red circle represent the entity initially impacted by the user and the blue circle is the impacted entity currently selected by the user in the tree list (1). The options at the bottom of the visualisation panel allow to configure the layout and the radius of the nodes in (3).

III.6 Discussion

In this chapter, a generic approach to perform Change Impact Analysis is proposed. To do that, concepts related to the approach have been presented, an extensible change model has been proposed as well as a results model allowing to store meaningful information on impacted entities, the approach has been described with an implementation in the Pharo environment. This implementation has then been specialised to compute change impact analysis on Pharo software.

The objectives fixed in the last chapters have been fulfilled. Indeed, the approach proposed is adaptable to different software paradigm and languages and can use different software model by implementing a specific model strategy. The approach can be used for different use cases by implementing a specific goal strategy. The results returned by the approach provide more information than just the entities impacted. Finally, the change model proposed can be extended

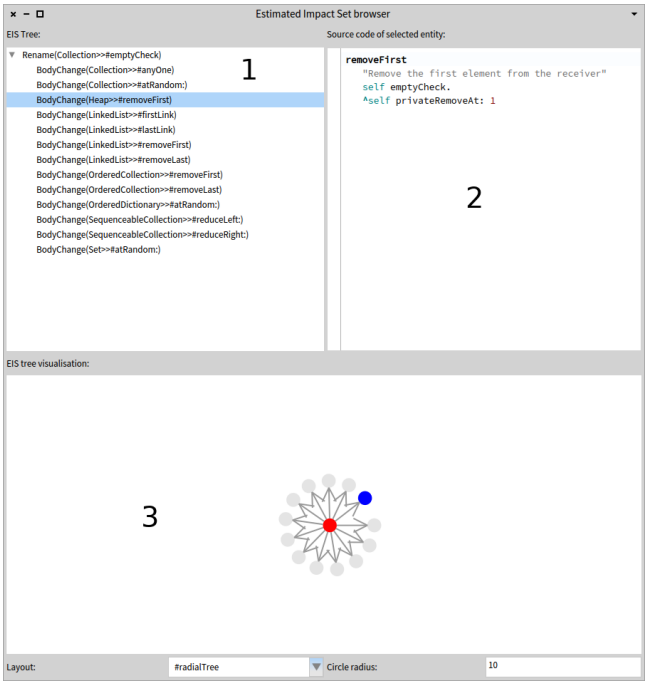


Figure III.11: Screenshot of the widget allowing to explore the result of a change impact analysis in Pharo.

to add more specific change types and compute their impact. The next chapter will be interested in the evaluation of the approach and tool proposed and its classification using Lehnert's taxonomy [15].

Chapter IV

Evaluation

This chapter evaluates the approach proposed in the previous chapter. First, the complexity of the generic approach and the developer strategy that is implemented in Pharo are assessed. Next, the approach is classified using Lehnert taxonomy. Finally, a protocol to assess the performance of the approach is presented, without being applied because of a lack of time to realise it.

IV.1 Complexity Estimation

In this section, the time complexity of the approach presented in the Chapter III is evaluated. This complexity will be expressed in big-O notation. With this evaluation, it will be possible to fulfil the scalability criterion in Lehnert Taxonomy. Thus, the approach will be comparable to others approaches using the same technique on this criterion.

IV.1.1 Generic Approach

Let E be the set of entities composing the model. Let $n_e = |E|$ be the number of entities present in the model. Let n_t be the number of change types available. The main algorithm of the approach (Algorithm 6) that compute the impact of a change concerning an entity of the software model given as parameter has a process similar to a breadth-first exploration of the graph formed with entities and their relations starting from the node containing the changed entity. Nevertheless, in contrast to such exploration, a node of the graph can be explored more than once. Indeed, at each iteration of the while loop (line 6 to 17) a change is removed from the FIFO queue and treated. This treatment will eventually add new changes in the FIFO queue to be treated in next iterations. Nevertheless, when a couple (change type, entity) that has already been treated before by the algorithm has to be added to the queue, it is ignored. But if the couple was not treated before, it is added to the FIFO queue. Because of this fact, in the worst case, a node of the graph can be visited n_t times. Thus, in the worst case, all the nodes of the graph will be treated n_t times.

Let φ_i be the worst-case complexity of the method that treat the i^{th} change

type. Therefore, the worst-case complexity of the generic algorithm is

$$O(\sum_{i=1}^{n_t}(n_e\varphi_i))$$

ALGORITHM 6: Generic algorithm that computes the Estimated Impact Set of a change on a software.

Input: A graph built from a software model s and a change c on it.
Output: The set of changes induced by the change c.

```
1:  procedure impact(s, c)
2:      toProcess := new FIFO
3:      toProcess.add(c)
4:      impacted := {c}
5:      while s.toProcess ≠ ∅ do
6:          c' := toProcess.next()
7:          switch c do
8:              case Add:
9:                  processAdd(c', s, toProcess, impacted)
10:             case Remove:
11:                 processRemove(c', s, toProcess, impacted)
12:             case Move:
13:                 processMove(c', s, toProcess, impacted)
14:             case Rename:
15:                 processRename(c', s, toProcess, impacted)
16:             ... // Other kinds of changes
17:             case Body:
18:                 processBody(c', s, toProcess, impacted)
19:     end while
20:     return impacted
21: end procedure
```

IV.1.2 Approach Implemented in Pharo

It is possible to apply the preceding formula to the Pharo case. In the implementation, five change types are provided (*i.e.*, $n_t = 5$). To fulfil the last formula the time complexity of each *process** algorithm has to be computed which is done below.

- *processAdd* (Algorithm 2) has a worst-case complexity in $O(n_e)$. Indeed, in the worst case, the entity currently treated is subject to inheritance and its super-class is referenced by all the entities of the system.

- *processRemove* (Algorithm 3) has a worst-case complexity in $O(n_e + (n_e - 1))$. Indeed, in the worst-case each entity references all the others entities of the software. This cause the n_e term. The $n_e - 1$ term is due to the fact that, in the worst-case, the entity currently treated by the algorithm

contains all the others entities of the system. Using the properties of big-O notation, the complexity of the algorithm is:

$$O(n_e + (n_e - 1)) = O(n_e + n_e) = O(2n_e) = O(n_e)$$

- *processMove* and *processRename* (Algorithm 4 and Algorithm 5) both have a worst-case complexity in $O(n_e)$. Indeed, in the worst-case each entity references all the others entities of the software.
- *processBody* has a worst-case complexity in $O(1)$ since it does nothing.

Knowing the worst-case complexity of each algorithm and since there are 5 change types ($n_t = 5$), we deduce that the overall complexity of the main algorithm is:

$$\begin{array}{rcl} O(\sum_{i=1}^{5}(n_e\varphi_i)) & = & O(n_e n_e + n_e n_e + n_e n_e + n_e n_e + n_e) \\ & = & O(4n_e^2 + n_e) \\ & = & O(n_e^2) \end{array}$$

IV.2 Classification in Lehnert Taxonomy

This section classifies the approach proposed in the previous chapter in Lehnert taxonomy described in Chapter I. The scope of the analysis, the granularity of entities used, the technique utilized, the style of analysis, the tool support, the supported languages, the scalability and the experimental results of the approach are evaluated.

IV.2.1 Scopes of Analysis

The approach main scope is the *source code*. More precisely, it works on models generated from the source code of the software being analysed. Thus the name of the scope of this analysis in Lehnert's taxonomy is "Code Static". Nevertheless, it should be possible to adapt it to analyse an architectural model.

IV.2.2 Granularity of Entities

This criterion concerns the granularity of the different types of entities used by the approach. The granularities of artefacts, changes and results are described below.

- *Artefact granularity*: The granularity of the artefacts depends on the software analysed and the model strategy implemented. In the model strategy implemented to analysed Pharo software, the granularity is at the package/class/method/instance variable level.
- *Change granularity*: The approach proposes 5 kind of changes by default: *add*, *remove*, *move*, *rename* and *body*. Nevertheless, the approach let the possibility to extend easily the change types.
- *Results granularity*: The results of an analysis of the approach contain the following information: Whatever the impacted entity has been induced by another impacted entity or directly by the user; the change to apply on the impacted entity in order to keep the software behaviour as it is before the modification; the impacted entity and the impacted entities induced.

IV.2.3 Utilized Technique

The approach uses a combination of two techniques. The first one is the *Call Graphs* technique. Indeed, to compute the impact on methods of an object-oriented software for example, the approach considers the call between methods. The second is *Dependency Analysis* technique. Indeed, the approach also uses dependencies like *reference*, *inheritance* and *composition* to compute the impact of a change on an entity.

IV.2.4 Style of Analysis

The style of analysis used by the approach is the *Global Analysis* style. Indeed, the approach analyses the whole system to find the impact of a change on an entity.

IV.2.5 Tool Support

The approach has been implemented in a tool inside the Pharo environment to analyse Pharo software. This tool allows to check the impact of an eventual change on an entity of a Pharo software and to browse the result of this analysis. Despite the fact that the tool has been specialised for Pharo software analysis, it has been implemented in a generic way in order to let the possibility to adapt it to another software.

IV.2.6 Supported Languages

The approach can potentially support any software language or model because of its generic nature. Nevertheless, the current implementation only support the analysis of Pharo software.

IV.2.7 Scalability

The scalability of the tool has been evaluated in the preceding section. The worst-case complexity of the generic approach is in

$$O(\sum_{i=1}^{n_t}(n_e\varphi_i))$$

where n_e is the number of entities in the software model, n_t is the number of change type used for the analysis and φ_i is the complexity of the algorithm that treat the change type number i.

For the approach implemented for Pharo Software analysis, the worst-case complexity is in

$$O(n_e^2)$$

where n_e is the number of entities in the software model.

IV.2.8 Experimental Results

No experiment were conducted in the context of this work. This is mainly due to the fact that it is *complicated to assess the precision and recall of an approach* as discussed in the next section. Nevertheless, an experimental protocol is proposed in the next section in order to assess the precision and recall of a Change Impact Analysis approach.

IV.3 Assessing Precision and Recall

Given an approach that perform Change Impact Analysis, it is complicated to assess its performances in terms of precision and recall. As a remainder, in this context, the precision is defined as the number of elements in the Estimated Impact Set (EIS) plus the number of elements in the Actual Impact Set (AIS) divided by the number of elements in the EIS (*i.e.*, $\frac{|EIS|+|AIS|}{|EIS|}$) where the recall is defined as $\frac{|EIS|+|AIS|}{|AIS|}$.

The problem to estimate precision and recall of a Change Impact Analysis tool comes from the fact that it is impossible to compute the AIS in an exact manner. Indeed, every tool implemented aims to compute an EIS as close to the AIS as possible but it is in fact actually impossible to know the real difference between the EIS and the AIS.

This problem can not be overcome by asking the opinion of a developer on the quality of the results produced by a tool. Indeed, as highlighted by Lindvall [18] and Tóth et al. [24], developers are not good at predicting the impact of a change on a software. Thus, it is not possible to, given a change on a software, ask them the impact of it and to compare it with the results of a tool.

What is often done by approaches that provide a precision or recall measure of their implementation is that they consider a certain set of impacted entities as the AIS.

IV.3.1 Experimental Protocol Proposed

To evaluate the precision and recall of the approach, two information are needed: the Starting Impact Set (SIS) which is the set of changes made on the software system that were directly performed by the user and the Actual Impact Set which is the set of changes that were induced by another change. To retrieve these information, the developer of the software analysed will be helpful.

The experimental protocol proposed to assess the performances of the approach uses the hypothesis that the AIS is the set of changes that have been performed by the developer and that are not marked as belonging to the SIS. The protocol steps are the following. First a set of developers assigned to projects are be selected for the experiment. To be able to evaluate the generic nature of the approach, these developers should work on projects that are using different software languages and different paradigm.

Then, a tool to monitor developers activity by recording the changes that are performed and the order in which they are performed is set up in the development environment of developers. Knowing the order in which the changes have been performed is important for the analysis of developers activity as explained after.

Once data from developers-activity has been collected, a discussion has to be organised with each developer. This discussion is about the activity recorded and aims to discover which changes performed by the developer belong to the SIS. That is to say, which changes have not been induced by another change. The changes that are not marked as belonging to SIS by the developers will be considered as belonging to the AIS. In this step, the fact that changes are ordered allows to help the developer not to do mistakes. Indeed, with this information and knowing the fact when a change induces another change it is performed before, the detection of the SIS should be easier.

Finally, Change Impact Analysis on the SIS retrieved from developers knowledge is performed. For each change in the SIS, the results of this analysis can be compared to the set of changes not belonging to the SIS that appeared after the change analysed.

IV.3.2 Threats to validity

In the protocol described in the preceding section, all the changes that do not belong to the SIS are considered to be the AIS. This might be an under estimation of the AIS. Indeed, it is possible that changes required to ensure the good behaviour of the software are needed but are not performed because they haven't been detected yet (by unit tests or by the developer).

Another point is that the changes belonging to the AIS are selected by the developer which may do mistake when selecting them. Nevertheless, the fact that the change log of developers includes the order in which these changes were performed should help the developer to do as little mistakes as possible.

Conclusion

The evolutionary nature of software motivated by its adaptation to requirements that are continuously evolving makes the use of Change Impact Analysis crucial in the process of change integration. Indeed, changing a part of the software often requires to adapt the other parts in order to keep it in a working state.

The work presented in this document aims to provide a unified technique to compute the impact of a change on a software and could be integrated in the works of Gustavo (which consists in improving refactoring techniques), Vincent (which consists in selecting unit tests to re-execute after a change has been performed) and Véronica (which consists in integrating features from a software in another software) presented in the Introduction.

For this purpose, the Change Impact Analysis research field has been explored and a state of the art of the domain has been realised. This state of the art allowed to explore the approaches existing in the literature and defined the the essential features a Change Impact Analysis approach has to provide.

Then, an approach to perform Change Impact Analysis has been developed. This approach aims to be independent of the software language, paradigm and model used. The framework implementing this approach can thus be adapted to these different parameters depending on what the user wants to do. In the context of this work, the framework has been specialised to analyse Pharo software and a graphical user interface has been developed to make its utilization more user-friendly.

Finally, the approach complexity has been evaluated, it has been classified in Lehnert taxonomy and an experimental protocol to assess the performance of the approach in terms of precision and recall has been developed.

Perspectives to this work are three fold. First, the experimental protocol described in Chapter IV could be set up in order to assess the performance of the approach. Concerning this experiment, the data gathered from developers activity could be shared to other researchers as open-data. Indeed, gathering this kind of information is complicated and requires many resources (time, energy and probably money to motivate developers to use the tool). Thus, providing these data for future research would be greatly help the scientific community.

Second, the approach presented in this document could be integrated to Gustavo, Vincent and Véronica. Then, a comparison of the results they got before

the integrations and the results get after the integration could be compared. Furthermore, this could give more information about the support of different use cases by the approach.

Finally, if the results of the experiment are good, the tool could be integrated in Pharo development environment and shipped with it to help developers. Nevertheless, this future work open a practical question which is *What kind of graphical user interface support is required to make the tool useful?* Indeed, such graphical user interface should help the developer without requiring much actions from him.

Bibliography

[1] Robert S Arnold and Shawn A Bohner. Impact analysis-towards a framework for comparison. In *Software Maintenance, 1993. CSM-93, Proceedings., Conference on*, pages 292–301. IEEE, 1993.

[2] Linda Badri, Mourad Badri, and Daniel St-Yves. Supporting predictive change impact analysis: a control call graph based technique. In *Software Engineering Conference, 2005. APSEC'05. 12th Asia-Pacific*, pages 9–pp. IEEE, 2005.

[3] Arpád Beszédes, Tamás Gergely, Judit Jász, Gabriella Tóth, Tibor Gyimóthy, and Václav Rajlich. Computation of static execute after relation with applications to software maintenance. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pages 295–304. IEEE, 2007.

[4] Haider Zuhair Bilal and Sue Black. Computing ripple effect for object oriented software. *Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE)*, 2006.

[5] Luke Bishop. *Incremental impact analysis for object-oriented software*. PhD thesis, Citeseer, 2004.

[6] Sue Black. Computing ripple effect for software maintenance. *Journal of Software: Evolution and Process*, 13, 2001.

[7] Lionel C Briand, Jurgen Wust, and Hakim Lounis. Using coupling measurement for impact analysis in object-oriented systems. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, pages 475–482. IEEE, 1999.

[8] Kunrong Chen and V Rajich. Ripples: tool for change in legacy software. In *Software Maintenance, 2001. Proceedings. IEEE International Conference on*, pages 230–239. IEEE, 2001.

[9] Martin Fowler, Kent Beck, J Brant, William Opdyke, and Don Roberts. Refactoring: Improving the design of existing programs, 1999.

[10] Steve Gwizdala, Yong Jiang, and Václav Rajlich. Tracker-a tool for change propagation in java. In *Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on*, pages 223–229. IEEE, 2003.

[11] Judit Jász, Árpád Beszédes, Tibor Gyimóthy, and Václav Rajlich. Static execute after/before as a replacement of traditional software dependencies. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 137–146. IEEE, 2008.

[12] Malia Sofia Kilpinen. *The Emergence of Change at the Systems Engineering and Software Design Interface*. PhD thesis, University of Cambridge, 2008.

[13] David Chenho Kung, Jerry Gao, Pei Hsia, F Wen, Yasufumi Toyoshima, and Cris Chen. Change impact identification in object oriented software maintenance. In *ICSM*, volume 94, pages 202–211, 1994.

[14] Michelle Lee, A Jefferson Offutt, and Roger T Alexander. Algorithmic analysis of the impacts of changes to object-oriented software. In *Technology of Object-Oriented Languages and Systems, 2000. TOOLS 34. Proceedings. 34th International Conference on*, pages 61–70. IEEE, 2000.

[15] Steffen Lehnert. [acm press the 12th international workshop and the 7th annual ercim workshop - szeged, hungary (2011.09.05-2011.09.06)] proceedings of the 12th international workshop and the 7th annual ercim workshop on principles on software evolution and software evolution - iwpse-evol '11 - a taxonomy for software change impact analysis. 2011.

[16] Steffen Lehnert. A review of software change impact analysis. *Ilmenau University of Technology, Tech. Rep*, 2011.

[17] Offutt Li. Proceedings of international conference on software maintenance icsm-96 - algorithmic analysis of the impact of changes to object-oriented software. 1996.

[18] Mikael Lindvall. Evaluating impact analysis – a case study. *Empirical Software Engineering*, 2, 06 1997.

[19] Maksym Petrenko and Václav Rajlich. Variable granularity for improving precision of impact analysis. In *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*, pages 10–19. IEEE, 2009.

[20] Guenter Pirklbauer, Christian Fasching, and Werner Kurschl. Improving change impact analysis with a tight integrated process and tool. In *Information Technology: New Generations (ITNG), 2010 Seventh International Conference on*, pages 956–961. IEEE, 2010.

[21] Vaclav Rajlich. A model for change propagation based on graph rewriting. In *Software Maintenance, 1997. Proceedings., International Conference on*, pages 84–91. IEEE, 1997.

[22] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G Ryder, and Ophelia Chesley. Chianti: a tool for change impact analysis of java programs. In *ACM Sigplan Notices*, volume 39, pages 432–448. ACM, 2004.

[23] Barbara G Ryder and Frank Tip. Change impact analysis for object-oriented programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 46–53. ACM, 2001.

[24] Gabriella Tóth, Péter Hegedűs, Árpád Beszédes, Tibor Gyimóthy, and Judit Jász. Comparison of different impact analysis methods and programmer's opinion: an empirical study. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, pages 109–118. ACM, 2010.

[25] Franck Xia. A change impact dependency measure for predicting the maintainability of source code. In *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, volume 2, pages 22–23. IEEE, 2004.

[26] Stephen S Yau, James S Collofello, and T MacGregor. Ripple effect analysis of software maintenance. In *Computer Software and Applications Conference, 1978. COMPSAC'78. The IEEE Computer Society's Second International*, pages 60–65. IEEE, 1978.

[27] Marcin Zalewski and Sibylle Schupp. Change impact analysis for generic libraries. In *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on*, pages 35–44. IEEE, 2006.