Supporting Software Development Process Using Evolution Analysis : a Brief Survey

Samaneh Bayat
Department of Computing Science,
University of Alberta, Edmonton, Canada
samaneh@ualberta.ca

Abstract

During development process, software systems constantly evolve to meet the system's functional and non-functional requirements. Analyzing the series of changes made during software development processes helps extracting best practices to consistently maintain, evolve, test and improve software systems. This paper presents a brief survey on software evolution analysis methods. This study classifies the methods to two classes, model-differencing and code-differencing, based on their approach to the change finding problem. The methods also differ in their abstraction level, way of representing results, computation time, or mining changes affecting clients of a framework. In addition, this paper discusses the progress of research in this area ranging from old to recent. The discussed methods are compared based on their change-finding power and their advantages to each other are shown.

1 Introduction

Analysis of changes made to a software system during the development process has several important results. A system's current maturity level is obtained by a list of changes that evolve elementary versions of a software to the final step. Identifying these changes helps software engineers to analyze the current state of software systems in order to understand design level evolution of a system. It also gives hints on extracting good design practices by identifying changes that were eventually reversed in the development process. In some cases, the reasons of system failure can be found by analyzing the history of changes. In addition, identifying changes that a system has gone through and representing them in logical rules can help finding inconsistent changes, resulting to patterns of consistent system evolution. Moreover, automatically extracting the changes is in favor of creating testing plans that are coherent with the changes and helps automation of maintenance process [3]. For all the above reasons, finding the change chain in software development process is valuable in software engineering research progress. Several approaches have been used to find the differences between two versions of a system. This paper is a brief survey on the fundamental concepts of some of these approaches.

The programming paradigm has evolved from procedural programming to the object oriented (OO) programming, during computing history. One can say that programming in low level of abstraction in the past has now reached a higher level of abstraction. As it is expected, research on different aspects of programming, including program differencing algorithms, have concentrated on the higher levels of abstraction. Different sections provided in this paper make this change in concentration more clear, by representing major researches done in this field in the order of published years. In Section 3, we discuss some old approaches to program differencing. This discussion is mainly presented to show the elementary ideas on program differencing and can be considered as a basis to newer approaches.

In Section 4, recent approaches to this problem are discussed. The presented approaches are concerned with object oriented programming. The OO software development process usually begins by abstracting the software into models and developing the models to an executable software. This executable software can be a result of executable models or codes. Design, structure, and hence coding of the program evolve in each stage of the development process in order to produce a program with a desirable quality. Analyzing the evolution of software development process can be done using the system's modeling components or coding elements. In this paper, we focus on the methods that use model differencing and the methods that are concerned about code changes in Section 4.1 and 4.2, respectively.

In model driven approaches, one major part of the methods is finding corresponding elements in two versions of the model. Since software models have a large set of model el-

ements, the computation time for these approaches is high. Section 4.1.3, presents an idea to improve the time complexity of these methods. Finally, a brief discussion is provided in Section 5 which compares the recent methods.

2 Terminology

In this section, we present a brief explanation of some technical words used in this paper related to software development process. As evident in this survey, the terminology used in this field is basically general computing science terminology. However, a few basic concepts that are essential to begin reading in this field are described in this section.

- **Object oriented programming** In OO programming, the system is described as a collection of *objects* that interact with each other to bring a special behavior to the whole system. Objects are instances of *classes* which are reusable encapsulations of data and behavior.
- **Procedural programming** In contrast with the OO programming, functionality of methods in the procedural programming is the basic concern. Variables mainly hold data to help the methods implement a certain functionality.
- **Software model** A formal language or a visualization used to describe a system is called model. Models are abstractions of the software which make the design and evaluation of software behavior easier.
- **UML** Unified Modeling Language (UML) is a standard general-purpose modeling language to model softwares and software development processes. UML can be used as a set of UML diagrams or UML meta-model (a model to describe the UML itself).
- **Code refactoring** Code Refactoring is a process of modifying some parts of a code. In this process, the external behavior of the system does not change. It is done to improve some quality metrics, such as code readability or maintainability.

3 Program differencing algorithms: In earliest decades

This section provides a view of the earliest attempts to solve the problem of program differencing. Comparing programs were initially seen as comparing texts, while changes made to a program are due to improvements to program behavior or changes in system's objectives. In 1990, S. Horwitz [5] introduces a technique to classify the program changes into semantic and textual differences. Generally, it is undecidable to find exact semantic changes, so Horwitz estimates a set of semantic changes. She identifies a semantic change in two general cases; First case: a semantic change happens when a new version of program (called *New*) introduces a new component that is not present in the old version (called *Old*). Second case: a semantic change happens when sequence of values produced at a component in *New* is different with sequence of values produced at its corresponding component at *Old*.

In this approach, programs are represented as graphs instead of pure text to make it easier to find equivalent components. These graphs are augmented versions of program flow graph and are called Program Representation Graphs (PRGs). In order to find semantic changes, a definition for equivalent behavior of program components is provided which is mainly based on the sequence of values produced in each component. To compare program changes, a partitioner puts components with equivalent behavior in the same partitions. Then, the changes to equivalent partitions are investigated. If a change is just a change in names, it will be classified in textual category, else a sequence of semantic changes is recognized regarding the program flow and will be tagged on various parts of the code.

The program definition used in the Horwitz's method, as a changing software, is a reduced definition of procedural programs. It contains only scalar variables, assignment statements, output statements, conditionals, and loops. Although representation of semantic changes in this method is too basic and simple, it is a good description of low level changes. However, this research is a basic step towards the distinguishing semantic and textual differences.

4 Program differencing algorithms: Recent approaches

This section discusses several recent approaches to the software differencing problem. All of these approaches focus on object oriented programs. The approaches are separately represented regarding their concentration on model or code elements of the program.

4.1 Model differencing

Models are abstractions of a system's behavior and implementation. It is easier to use models to find changes, because dealing with code details makes the differencing algorithms more complex. Models can be represented by meta-model structures or by diagrams. UML is a standard language that can provide both of these representations for a system. Diagrams make understanding of the software changes easier by visualization, while meta-models help finding more conceptual relationships between entities.

In this section, we study two UML-based approaches to analyze history of changes in a software system. Section 4.1.1 explains about finding differences between UML diagrams with respect to different categories of changes, as a representative of changes in the program. Section 4.1.2 elaborates the idea of using UML meta-model to detect the changes in two consecutive designs of the program. In addition, Section 4.1.3 provides a method to decrease computation time of the discussed methods.

4.1.1 Diagram differencing

Ohst *et al.*, 2003, [8], presented a method that finds changes of a software's UML diagrams and visualizes these changes. Here, we concentrate on their approach to finding changes while pointing to a few visualizing ideas.

Ohst *et al.* compared two versions of a UML diagram by considering their physical (diagram layout) and mental (software's entities) models separately. They categorized data that is in a UML file into two categories: layout data and model data, similar to what is described in the UML specification [1]. Layout data holds information that are actually irrelevant to semantic and textual representation of UML's content, for example size and position of nodes in the diagram are not considered in design differencing but are needed to visualize changes in the diagram. In contrast, model data contains all other information that are relevant to semantics of the diagram. This categorization makes it easier to concentrate on logical structure of the UML.

Logical structure of a UML diagram can be better represented by graphs rather than files, because graphs keep diagrams away from low level ordering of entities. So, Ohst *et al.* created an object structure graph. In this object graph, every class component is shown by an object. And the relationships between classes are shown by labeled (kind of relationship) edges.

The method uses a differencing operator to traverse the object graphs of two versions of UML, simultaneously. This operator finds the corresponding objects in two different versions and detects their changes. Result of this operation is a unified graph that illustrates the changes. Each object in the unified graph shows the unchanged parts and gives information about changed parts of two corresponding objects in two object graphs. It is good to know that the layout data is ignored in creating the unified graph. In another phase, the UML graph is traversed again to find the layout changes. In this way, their method finds diagram's semantic changes and layout changes, independently.

Related to finding differences between two diagram versions, Ohst *et al.* have classified the changes that can be made or seen in a UML diagram into the two following classes and have used these definitions in traversing object graphs.

- Intra-node differences: Modifications to inner parts of a node of diagram, for example, changing names or removing an attribute from a class.

- Graph structure differences: Modifications of structure of the representing graph, for example, shifting, creating, or removing nodes.

Both of the above classes can contain physical or mental models of the diagram. Intra-node differences can contain changes in states of a state chart (physical), as well as modifications in attributes of a class (mental). Graph structure differences include shifting classes across the diagram (physical), as well as changes in relationships between classes (mental).

In general, this work presented a practical example of finding semantical changes in a software model while visualizing the changes in the diagram.

4.1.2 Design differencing

Since models are the best representatives for design facts of a software system, analyzing models helps to have a precise picture of structure evolution at the design level.

Xing *et al.*, 2005, [11] presented *UMLDiff*, an algorithm that detects structural changes between designs of two versions of a program. Their approach is based on a UML meta-model of the system. Inputs to the *UMLDiff* are class models (instances of the meta-model) corresponding to two code versions of a software system. This class models are obtained by reverse engineering the code versions. The output of this algorithm is a change tree that shows the changes such as additions, removals, moves, renamings, and changes to relationships between design elements.

The change tree is produced by traversing class models, represented as directed graphs, and identifying corresponding entities. The corresponding entities in two class models are the conceptually equivalent design elements and are detected by two similarity heuristics. These heuristics, based on object oriented design semantics, find entities that are the same even after they are renamed. One of these heuristics uses the number of common adjacent character pairs to find lexical similarities. The other one uses several structural facts of the entities to compute a similarity value. Then, these two heuristics are combined to measure similarity for pairs of entities. Two entities are considered the same if their similarity value is above a predefined threshold that determines the level of accepted risk in finding corresponding entities.

Finally, changes such as renamings, moves, and changes in types are recognized by considering changes in similar entities. This is done by applying several routines such as searching for moves, using the structural heuristic to find

renamings, and comparing entities with each other. Showing the changes in a tree structure helps users of the method find changes in logical positions which makes analysis of changes easier.

4.1.3 Improving computation time

Most model-differencing methods are based on searching and pairing similar elements which inherently have time complexity of $O(n^2)$, where n is the number of elements in the model. Since software models are usually large, this time complexity is not desirable. Treude *et al.*, 2007, [10] introduced a method to reduce the time complexity to $O(n \log(n))$. In this section, their methodology is briefly described.

To find similar model elements Treude *et al.* used a high-dimensional balanced search tree. The properties of this search tree is motivated by disc-oriented search trees that are applied in information retrieval systems [4]. The main advantage of these trees is that they arrange similar elements to be next to each other.

In their method, in order to be able to use the search tree, elements of the model are represented by numerical vectors. Indices of the vectors show model element characteristics. Vectors contain a metrical and a lexical part. Metrical indices represent concepts of software metrics such as number of abstract methods, private or protected attributes, and class or instance attributes. Similarly, lexical indices are used to represent lexical characteristics of elements.

After all, these vectors are located in search tree such that similar nodes be next to each other by assistance of a hashing idea. In this approach, similarity is computed by vector's Euclidean distance which is small if indices are nearly the same. A threshold value is used to define the similarity acceptance risk. Having similar nodes next to each other reduces the time complexity of matching pairs of elements to $O(n \log(n))$.

4.2 Code differencing

Another approach to extract evolutionary history of a software system is to analyze code element changes. This can be done in different levels of abstraction. Finding renaming of variables and methods, or the sequence of values that has been produced in a code element are in low levels of abstraction. An example of this approach was presented in Section 3. Considering object oriented programming paradigm, a higher level of abstraction can be gained by looking at renaming of classes and packages, or changes in relationships among classes. Another high level approach is finding usage changes. This approach is mostly concerned with frameworks and the effect of their evolution on their clients. Section 4.2.1 explains this approach more precisely.

In any level of abstraction, changes are found as a huge set of rules, each one describing an individual code element. Giving an enormous set of changes to programmers does not help them to find out what has happened till now and what is to be done next. Instead, researches suggest grouping these changes. Different approaches to categorize these changes have been followed till now. Kim *et al.* [7] have found the following categorizations in their recent researches:

(i) Based on physical locations (files and directories).

(ii) Based on logical locations (packages, classes, methods): An example of this categorization is *UMLDiff*.

(iii) Based on structural dependencies.

(iv) Based on similarity of names.

Research done by Kim *et al.* uses structural dependencies to group changes. As an improvement to this approach, they identify systematic changes which is discussed in Section 4.2.2.

4.2.1 Framework usage changes

Schafer *et al.*, 2008, [9] suggested to find changes made in the way framework instantiations use the framework instead of looking for changes made to the framework code. They showed that this approach is an improvement to other approaches that compare two versions of the framework because their method finds conceptual usage changes in addition to changes caused by refactorings. Examples of these conceptual changes are modifications to assigning responsibilities to building blocks of the framework, which is not clearly understood by effects of refactoring. Also, a problem resists in approaches that find framework changes made by refactorings. This problem arises because of maintaining backward compatibility in frameworks. Existence of old version elements that are not removed in benefit of backward compatibility prevents detecting refactoring while a usage change has occurred.

Schafer *et al.* have used association rule mining technique [2] to find rules of usage changes. This technique, given a set of transactions - consisting of a set of facts - finds more likely rules that associate a set of facts to another. In the Schafer *et al.*'s algorithm, facts show framework usages such as method calls and inheritances and they are extracted from each instantiation code. Transactions can be simply union of these facts for each instantiation class. But, Schafer *et al.* showed that this way of creating transactions causes several problems such as ambiguity in extracting final rules. So, they suggested grouping facts that are in a class declaration separate from those in a field or

method declaration. In addition, they suggested partitioning facts in each of these groups based on 5 change patterns. These change patterns are recognized by the changes in inheritances, overridings, calls and accesses, and instantiations. Also, they showed that combining these two suggestions helps finding more rules by disambiguating. Finally, transactions are given to the association rule mining algorithm and rules of changes are extracted so. Explaining this machine learning technique is out of the scope of this survey.

They show that their approach finds conceptual changes and does not have difficulty in finding changes even in presence of outdated and deprecated code elements.

4.2.2 Systematic code changes

Kim *et al.*, 2009, [7] introduced *LSDiff* a tool that uses structural dependencies to find systematic code changes. Structural dependencies are found in code behaviors such as field accesses and overridings. *LSDiff* uses code elements and their structural dependencies to make an abstraction out of a program. Then, it uses this abstraction to identify systematic changes. These changes are found by considering consistent changes to code elements that are structurally similar. *LSDiff* employs logic rules (conjunctive logic literals) and logic facts to show these changes and to infer more structural changes. Each rule is a human readable representation of a group of changes that are structurally similar. Grouping changes together helps creating a big picture of change patterns and finding the reasons for the changes.

This approach also helps finding anomalies in changes made to programs. It is a result of the fact that exceptions to logic rules are indicators of exceptions in systematic changes. This assists programmers to avoid inconsistencies in the evolving software system, which is one of the main goals of software differencing.

5 Discussion of the algorithms

In the following subsections, we discuss the results of applying the described algorithms to real case studies. What is presented here is the results obtained by the authors of the described papers joint together.

5.1 Comparing *LSDiff* and *UMLDiff*

Kim *et al.* in their previous work on *LSDiff* [6], 2007, prepared a case study to show their matching power. This earlier version of *LSDiff* is conceptually the same as the recent one so they are the same in the way that was discussed in this survey. In the case study, they analyzed the matching power of their algorithm and *UMLDiff* by having the number of found matches (M) and the number of correct matches (E), where E is manually determined by looking at the case study. Then, *precision* is defined as percentage of found matches that are correct ($\frac{|M \cap E|}{|M|}$). The study showed that *UMLDiff*'s precision was 2% higher. Instead, *LSDiff* found 761 correct matches that *UMLDiff* missed while number of missed matches by *LSDiff* were 199.

In addition, grouping structural changes into logic rules causes the result set of *LSDiff* to be 77% smaller than size of *UMLDiff*'s result set. As mentioned in Section 4.2.2, it was one of main goals of Kim *et al.* research and has a great value to users of the tool.

The aforementioned case study also showed that most of the missed matches by *UMLDiff* were those changes that involved both renaming and moving refactoring. While many of the matches that *LSDiff* missed were the changes that involved very low similarity in names. These show the weaknesses of these two algorithms.

This section showed that *LSDiff* finds more structural changes than *UMLDiff*, while *UMLDiff* is more precise in results. An obvious advantage of *LSDiff* over *UMLDiff* is reducing result set size [6].

5.2 Comparing *LSDiff* and mining framework usage changes method

Both *LSDiff* and mining framework usage changes method (abbreviated to FRAM) consider creating and using logical rules in their algorithms. *LSDiff* uses first order logic with variables, while FRAM uses association rules which are propositional logic rules without variables. The main difference between their logical approach is taking or not taking the advantage of having variables into account. Variables can be used to retain logical references to code elements [7].

In addition, FRAM approach uses a predefined set of rule patterns that are dictated by association rule mining algorithm. This may limit discovery of some changes in contrast with using first order logic [7].

As Kim *et al.* claim, logical rules found by *LSDiff* are more expressive than FRAM's rules, and they can infer a broader class of changes [7]. However, FRAM is more useful for understanding the changes that must be made to the framework's clients. In this point of view, *LSDiff* cannot be easily compared with FRAM, because their environment of study is not exactly the same.

6 Conclusion

This survey presented several methods that successfully touch the problem of analyzing the changes made to a software system during its development process. This analysis helps understanding the evolutionary behavior of the

software development process. In addition, it can help extracting patterns of evolution, plans for future development, maintenance, and testing.

This paper described basic concepts of the methods by order of their published years. This arrangement showed the progress of approaches in this research area and the change in trends. The earliest approaches are representations of basic ideas of the solutions to methods. While newest approaches use matured logical ideas and describe high level changes.

In the recent approaches, two classes of methods, model differencing and code differencing, were identified and discussed seperately. In the described model differencing methods, an earlier research touches the problem of finding differences between two visualized diagrams. While a newer approach looks for design level changes by taking advantage of a meta-model for the software. In this class of approaches, another research has been described that tries to reduce the time complexity of existing approaches by applying search trees and hashing concepts in matching model elements. In code differencing approach, two major research attempts have been discussed. One of them focuses on frameworks and finds software changes by studying the effects of changes to the clients of the framework. The other one, finds systematic changes by grouping structural changes.

A brief discussion on matching power of 3 most interesting and comparable methods, *UMLDiff*, *LSDiff*, and FRAM, showed that *LSDiff* finds a broader class of changes than both *UMLDiff* and FRAM. While the precision of found matches was slightly better in *UMLDiff* than *LSDiff*, number of missed matches were noticeably more in *UMLDiff* than *LSDiff*. Moreover, *LSDiff* produces a smaller and more understandable result set than *UMLDiff*. Although *LSDiff* is claimed to introduce changes more expressively than FRAM, their study bed is not completely comparable because FRAM concentrates on changes forced to clients while *LSDiff* focuses on changes made to the framework itself.

References

[1] OMG. Unified Modeling Language Specification. OMG, March 2003. Version 1.5 formal/03-03-01.

[2] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 207–216, New York, NY, USA, 1993. ACM.

[3] P. Benedusi, A. Cmitile, and U. De Carlini. Post-maintenance testing based on path change analysis. In *Software Maintenance, 1988., Proceedings of the Conference on*, pages 352–361, Oct. 1988.

[4] A. Henrich, H. W. Six, and P. Widmayer. The lsd tree: spatial access to multidimensional and non-point objects. In *VLDB '89: Proceedings of the 15th international conference on Very large data bases*, pages 45–53, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[5] Susan Horwitz. Identifying the semantic and textual differences between two versions of a program. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 1990. ACM.

[6] M. Kim, D. Notkin, and D. Grossman. Automatic inference of structural changes for matching across program versions. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 333–343, May 2007.

[7] Miryung Kim and David Notkin. Discovering and representing systematic code changes. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 309–319, Washington, DC, USA, 2009. IEEE Computer Society.

[8] Dirk Ohst, Michael Welle, and Udo Kelter. Differences between versions of uml diagrams. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 227–236, New York, NY, USA, 2003. ACM.

[9] Thorsten Schäfer, Jan Jonas, and Mira Mezini. Mining framework usage changes from instantiation code. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 471–480, New York, NY, USA, 2008. ACM.

[10] Christoph Treude, Stefan Berlik, Sven Wenzel, and Udo Kelter. Difference computation of large models. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 295–304, New York, NY, USA, 2007. ACM.

[11] Zhenchang Xing and Eleni Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 54–65, New York, NY, USA, 2005. ACM.