Consistent Evolution of UML Models by Automatic Detection of Change Traces

Cristine R. Dantas  Leonardo G. P. Murta  Cláudia M. L. Werner
Federal University of Rio de Janeiro, COPPE – System Engineering and Computer Science
Rio de Janeiro, RJ 21945-970 Brazil – P.O. Box 68511
{cristine, murta, werner}@cos.ufrj.br

Abstract

As software evolves, analysis and design models should be modified, correspondingly. In this scenario, one of the main problems is to detect which elements should be changed due to a given change. This paper presents an approach that applies data mining over a versioned UML repository in order to detect change patterns among model elements at different abstraction levels. These patterns are presented to the software engineer together with contextual information obtained from a change control system, so he can decide the software evolution task at hand.

1. Introduction

The software development process encompasses many distinct phases, each of them working on a specific abstraction level. Some decades ago, completely different paradigms were used to construct artifacts in these different phases of the development process. Fortunately, in the last decades some paradigms, such as object-oriented, aspect-oriented and component-based, have tried to reduce the gap between the problem definition and its solution. Besides the chosen paradigm, UML notation were always a strong candidate for representing analysis and design artifacts [1-3]. However, software engineers must ensure that all UML artifacts are up to date and consistent throughout the abstraction levels to avoid misunderstandings.

Traceability techniques can be used to solve this problem by identifying all UML model elements that shall be updated when a change is introduced [4]. By analyzing the traceability links, the software engineer can identify the potential side-effects of a new change, estimating what should be modified to accomplish a proposed change. Moreover, traceability links detection at analysis and design levels helps software engineers to identify bad design decisions early on the software development process and provides a high level view of the system dependencies [5].

The traceability link detection approaches based on data mining [6-8] usually search for concomitant changes in Software Configuration Management (SCM) systems to discover possible dependencies among software artifacts. SCM techniques are commonly used to control the evolution of software systems by providing versioning, via Version Control Systems (VCS), and support for activities related to change management, via Change Control Systems (CCS).

Our work uses a data mining technique named association rules, which extracts sets of items that frequently occur in the database transactions. Table 1 shows how each data mining term is related to our approach. In the classical usage context of data mining, all sales are analyzed to detect buying patterns among different customers. These patterns indicate which items are frequently purchased together. In our approach, items are represented by UML model elements. Therefore, the mined database is a versioned UML repository that stores changes performed during the development and maintenance phases. Moreover, each database transaction is mapped to a change, implemented via one or more check-in operations.

This paper is organized in four sections besides this introduction. Section 2 describes our approach for change traces detection via association rule data mining technique over versioned UML repositories. Section 3 presents some related works. Finally, Section 4 concludes the paper presenting some future works.

Table 1. Data mining terminology

<table>
<thead>
<tr>
<th>Context</th>
<th>Item</th>
<th>Transaction</th>
<th>Database</th>
<th>Mining Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supermarket</td>
<td>Product</td>
<td>Sale</td>
<td>Purchase database</td>
<td>Customer preference</td>
</tr>
<tr>
<td>Software evolution</td>
<td>UML model elements</td>
<td>Change</td>
<td>Versioned UML repository</td>
<td>Change traces</td>
</tr>
</tbody>
</table>
2. Mining UML repositories

This section introduces the proposed approach for traceability link detection among UML model elements, which encloses mining of change traces and gathering of rationale behind the change traces.

2.1. Change traces

Change traces are defined as traceability links obtained from change information. Throughout the lifecycle of a project, software artifacts are changed when new features are added or bugs are fixed. All these changes are stored in CCS and VCS.

In the context of CCS, changes have a predefined lifecycle, determined by the SCM activities [9]: (1) change request, (2) classification, (3) analysis, (4) evaluation, (5) implementation, (6) verification, and (7) integration. During the implementation phase of this lifecycle, different software artifacts are modified to perform the change. Each version of the modified artifacts is stored in VCS. Along with the versioned artifacts, other information is kept by VCS, such as change description, author, and date of each check-in operation performed during the change implementation.

In a previous work, an infrastructure named Odyssey-VCS [10] was defined to version UML model elements. The integration of information provided by CCS and Odyssey-VCS allows the detection of which artifacts are affected by a given change.

When a requested change is implemented, distinct artifacts are modified at different abstraction levels. Figure 1 shows two types of change traces that can be obtained by our approach: intra-model and inter-model.

The intra-model change trace relates two or more elements in the same abstraction level. For example, it is possible to detect that two use cases are always changed together. On the other hand, inter-model change trace relates two or more elements in different abstraction levels. For example, it is possible to detect that a use case and a class are always changed together.

2.2. Traceability links rationale

While developers investigate change traces, they pose various questions to uncover the rationale of dependencies that were identified. Currently, most approaches that support traceability lack to provide the semantics of the relations among software artifacts. To minimize this problem, our approach provides a view of all related artifacts that shall be updated and the details about the rationale, the history, and the people behind the change traces. Such details are vital in assisting developers to understand the state of the software system throughout the evolution process [11].

This rationale follows an information structure widely used in the Computer Supported Cooperative Work literature to contextualize and provide knowledge about elements [12]. This information structure is known as 5W+1H and comprises the following pieces of information: who, when, where, why, what, and how.

For example, the knowledge of who has performed a change may assist developers to understand if the change trace was introduced by a senior developer. Moreover, even if the change trace were introduced by a senior developer, other questions may indicate that this change trace was introduced to fix a critical bug in the few days/hours before a release, lacking adherence to some coding standards used in the company.

However, in the real world of software development, manually recording such information is neither possible nor practical. On the other hand, SCM repositories store change details obtained from SCM activities presented in Section 2.1. These change details are automatically collected and organized by our approach to provide the required 5W+1H information, as shown in Table 2.

<table>
<thead>
<tr>
<th>Information type</th>
<th>Collection place</th>
</tr>
</thead>
</table>
| Who              | • Check-in information from VCS  
                  | • Implementation activity from CCS |
| When             | • Check-in information from VCS  
                  | • Implementation activity from CCS |
| How              | • Impact analysis from CCS |
| Why              | • Change request activity from CCS |
| What             | • Check-in information from VCS  
                  | • Verification activity from CCS |
| Where            | • Check-in information from VCS |

The automatic gathering of 5W+1H information depends on another previous work named Odyssey-CCS.
[10], which is a CCS that allows process and information customization. Our work uses Odyssey-CCS to collect the rationale information shown in Table 2. For each required piece of information, the gathering place should be defined in terms of the field inside a specific document that stores the information in Odyssey-CCS. For example, to collect the “why” information using a conventional SCM process [9], the field “change description” in the document “change request” should be inspected.

2.3. Process for traceability links detection

The process for traceability links detection comprises two main phases: configuration and querying. The configuration phase is when the configuration manager sets the desired data mining metrics and informs the activities and fields of CCS that should be used to collect rationale information, as mentioned in the previous section.

The querying phase occurs when developers want to know the change traces among UML model elements. Usually, a UML model element is selected and all traceability links for this element are provided. This mechanism helps to answer questions such as “Developers that change a given element also change which other elements?”, providing support for the developers’ future changes.

It is important to notice that this approach does not intend to replace the work of software engineers. Since change traces are based on past experience, they do not constitute absolute truth, but suggestions. To categorize the relevance of these suggestions, each change trace has a probabilistic interpretation based on the amount of evidence from the data that they are derived from.

This evidence can be represented by two data mining metrics: support and confidence. These metrics are used to mine only frequent rules in databases. Support quantifies the significance of the (co-)occurrence of artifacts in implemented changes. Confidence represents how much one artifact depends on others. Typically, association rules techniques are interested in rules that satisfy both minimum support and confidence thresholds. In our approach, such thresholds should be set by the configuration manager at the configuration phase.

The process for mining change traces related to the selected model element comprises two steps: (1) selection of changes that will be analyzed, and (2) mining the transactions, searching for rules that describe relationships between UML model elements.

Only changes that satisfy the following conditions are analyzed by the data mining algorithm: (1) the change should be completely implemented, (2) the change should be older than the UML model element that is been queried by the developer, and (3) the change should affect the UML model element that is been queried by the developer.

After that, our approach can apply data mining over the changes selected according to the above conditions. The data mining algorithm calculates the support and confidence metrics for each pair of artifacts which are included in the changes and, then, prunes the elements with support and confidence lower than the minimum thresholds.

Our approach presents, together with the mined traceability links, the rationale behind each change trace, as shown in Figure 2. The rationale is composed by the 5W+1H information (shown in Table 2) collected from every change that participates in the mining of each detected change trace.

![Figure 2. Detected change traces.](image)

3. Related work

In the last decade, researchers have experienced the use of SCM repositories to understand software as well as their evolution. Gall et al. [13] use release data to detect logical coupling between modules. Ball et al. [14] have performed some cluster analysis of C++ classes stored in SCM repositories. A similar work has been conducted later by Bieman et al. [15] on classes of a commercial system. In this work, they identify change-prone classes that are targets for re-engineering. Some works also perform historical analysis over SCM repositories. Shirabad et al. [7] use inductive learning to find out different concepts of relevance among logically coupled files. Eick et al. [16] argue that code decay is related to the difficulty to perform changes. For this reason, they analyze change history applying decay indexes to identify risk factors. Dra-
heim et al. [17] argue that product quality is dependent of process quality. Due to this argument, the development process activities are analyzed and some metrics are applied over VCS. Finally, Zimmermann et al. [8] have evidenced that mining SCM repositories can be useful for suggesting and predicting likely further changes, detecting hidden dependencies, and preventing errors due to incomplete changes.

However, these related approaches work over file-based SCM repositories. For this reason, they lack support for traceability links detection of fine-grained UML model elements. Moreover, they do not provide change traces rationale, automatically extracted from an integrated CCS and VCS infrastructure.

4. Conclusion

This paper presented an approach for the detection of change traces among UML model elements. This approach can contribute to the impact analysis activity, avoiding under-prediction of the scope of a change and helping to identify critical side-effects. Also, it can indicate anomalies in the design structure which may be subject to restructuring if a change trace between two classes of different components is detected.

Some positive aspects of our approach are: (1) automatic change traces detection; (2) use of UML model elements as mining units, leveraging the state of the art to fine-grained analysis and design artifacts; and (3) description of change traces rationale using the 5W+H structure, automatically collected from an integrated CCS and VCS infrastructure.

Our future work includes the summarization of the reasoning information. This feature will allow, for example, detecting which developer is the main expert for a given artifact due to the number of changes he/she has previously performed in this artifact. Moreover, we intend to perform some case studies to verify how the benefits and limitations of our approach affect the maintenance tasks in a real usage scenario.

5. References