Assessing Changeability by Investigating the Propagation of Change Types

Beat Fluri
s.e.a.l. – software architecture and evolution lab
Department of Informatics
University of Zurich, Switzerland
fluri@ifi.unizh.ch

Abstract

We propose an approach to build a changeability assessment model *for source code entities. Based on this model, we will assess the changeability of evolving software systems. The changeability assessment is based on a taxonomy of more than 30 change types and a classifation of these in terms of change significance levels for consecutive versions of software entities. We consider change type propagation on different levels of granularity ranging from method changes to interface and class changes. We claim that this kind of assessment is effective in pointing to potential causes of maintainability problems in evolving software systems.*

1. Motivation

Assessing the maintainability of software systems is a crucial task in software engineering. Most of the existing approaches in this research area are limited to software measurement. Since traditional software measurement does not take the evolution of software systems into account, it does not put the *change* in the center of the investigated development process.

Research in the area of software evolution analyzes the change history of software systems to obtain deeper insights in their changeability and discover maintainability problems. These approaches rely on information provided by versioning systems such as CVS [2, 7]. They usually track changes of files on a *text basis* and are unable to enrich changes with structural source code information such as the method declaration. That means, when using changes gained from CVS, the corresponding source code entities have to be reconstructed as done by [7]. Such approaches neither extract the kind of changes, *e.g.*, condition expression change of an if-statement, nor their significance.

2. Proposed Solution

Our vision is to build a *changeability assessment model* for source code entities to assess the changeability of evolving software systems. In particular, this consists of three contributions: (i) A *taxonomy of source code changes* that defines change types according to tree edit operations in the abstract syntax tree. In addition, we provide a change extraction benchmark for assessing fine-grained change detection. (ii) A catalogue of changeability criteria to enable the changeability assessment of source code entities and software systems as a whole. (iii) A *changeability assessment model* for source code entities to assess the changeability of evolving software systems based on changeability criteria and the analysis of change patterns.

Building a taxonomy of source code changes. In [4] we showed that it is worthwhile to extract fine-grained source code changes for investigating hidden dependencies that lead to increasing maintenance effort. To provide an elaborate set of fine-grained source code changes and their extraction for further analysis, we have built a *taxonomy of source code changes* in [3].

Extracting change patterns. For a source code entity E (class or method) we obtain the entities from the source code model that depend on E (dependency set D_E) in a particular release. Since the data stored in our release history database (RHDB) [2] provide the information which entities have built a transaction (change coupling), we get the part of the dependent entities with which E has formed transaction t. Combining this information with the change type data for each entity we can build the particular *change pattern* for the whole transaction t.

We aim at extracting such change patterns automatically as well as cluster and filter them to analyze the change patterns with the changeability criteria.

A catalogue of changeability criteria. Currently we focus our research on a catalogue of changeability criteria to enable the changeability assessment of source code entities and software systems. We start with criteria for methods and continue to collect criteria for classes and packages until we reach the system level. A possible criterion can be: Changes in the pre- and postcondition of a method must not occur frequently. We claim, that violations of such a criterium lead to extra change effort in the callers of a method.

A changeability assessment model. Based on changeability criteria and the analysis of change patterns, we define a *changeability assessment model* of source code entities to assess the changeability of evolving software systems. To build the model we address questions such as the following: Does the dependency set D_E of an entity for a change pattern also repeat when the change pattern repeats?

Assume the entity dependency set comprises the callers of the *public* method M. By investigating the research question above, a possible result assessing the changeability of method M can be: Whenever only the body of M changed, the same set of callers has also changed. Based on the frequency of such changes and how many caller the method has, changes in the body of the method lead to extra change effort.

With this model we claim to be able to classify the changeability of a source code entity as *low*, *medium*, or *high*. Using such a classification a software architect is able to decide which parts of a software system have to be adapted to reduce or limit further maintainability effort. With the integration of our approach in a development environment, the developer gets feedback about the possible impact of a change. The feedback may be used as indicator for other source code entities one has to consider when applying a certain change.

3. Evaluation

We split the evaluation of our changeability assessment approach into two parts. The first part evaluates our source code change extraction algorithm. We have built an elaborate benchmark for which we have manually classified 1,056 changes from open source projects. With our benchmark, we are able to show that our source code change extraction algorithm achieves the minimal set of source code changes with an accuracy of 77%.

The second part, the evaluation of the model for assessing changeability, will comprise a set of case studies on open source software projects. We plan to validate the changeability classification of entities by manually inspecting the entities, their dependencies, and the changes among them to determine the correctness of the classification. We interpret the results of both the automatic and the manual classification and discuss differences between them. In addition, we intend to use defect tracking data to validate whether or not those entities characterized as problematic also have defects.

4. Related Work

The field of software evolution analysis research encouraged various researches to contribute to the detection of maintainability hot-spots. For instance, Fischer *et al.* [2] analyzed the history of changes in software systems to detect the hidden dependencies between modules. Such approaches do not take fine-grained changes into account, but rely on the insufficient information *that* a change happend.

In the area of change propagation analysis, researchers focus on the prediction of change propagation and on providing recommendation for change tasks [1, 7]. Neither of the change propagation analysis approach focuses on the investigation *how* fine-grained changes propagate, nor are they use to assess the changeability of a software system.

In the area of change impact analysis Stoerzer *et al.* developed an approach to aid developer by finding changes that induce failures in test cases [6]. Change impact data can be generated using approaches such as [5]. In contrast to our approach, which also uses change impact data, they focus on a particular development task. We want to assess the changeability of source code entities in general.

Acknowledgment. I would like to thank Michael Würsch for his support in developing the change extraction benchmark, Martin Pinzger for his valuable feedbacks, and my advisor Harald C. Gall.

References

[1] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth. Hipikat: A project memory for software development. *IEEE Trans. Software Eng.*, 31(6):446–465, June 2005.

[2] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proc. Int'l Conf. Software Maintenance*, pages 23–32, September 2003.

[3] B. Fluri and H. C. Gall. Classifying change types for qualifying change couplings. In *Proc. Int'l Conf. Program Comprehension*, pages 35–45, June 2006.

[4] B. Fluri, H. C. Gall, and M. Pinzger. Fine-grained analysis of change couplings. In *Proc. Int'l Workshop Source Code Analysis and Manipulation*, September 2005.

[5] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *Proc. European Software Eng. Conf. and Symposium Foundations Software Eng.*, pages 128–137, September 2003.

[6] M. Stoerzer, B. G. Ryder, X. Ren, and F. Tip. Finding failure-inducing changes in java programs using change classification. In *Proc. Symposium Foundations Software Eng.*, pages 57–68, November 2006.

[7] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Trans. Software Eng.*, 31(6):429–445, June 2005.