Modelling Change-based Software Evolution

Romain Robbes *and* Michele Lanza (Advisor)

Faculty of Informatics
University of Lugano, Switzerland

Abstract. More than 90% of the cost of software is due to maintenance and evolution[1]. We claim that the commonly held vision of a software as a set of files, and its history as a set of versions does not accurately represent the phenomenon of software evolution: Software development is an incremental process more complex than simply writing lines of text. To better understand and address the problem of software evolution we propose a model of software in which *change* is explicitly represented to closely match how object-oriented software is implemented. To validate our approach we implemented this model in an Integrated Development Environment (IDE) and used it to perform software evolution analysis. Further validation will continue with the implementation of tools exploiting this change-based model of software to assist software development.

1 Introduction and Problem Statement

Once implemented, a software system has to adapt to new requirements to stay useful [1]: Over time, systems grow and become more complex. Maintaining these systems is hard since developers deal with a large code base they might not fully understand when performing modifications. They have to identify where in the system to apply the change, and keep track of numerous parameters to not introduce bugs during these interventions. 40 % of bugs are indeed introduced while correcting previous bugs [2].

Several approaches exist to assist software evolution. Agile methodologies [3] acknowledge that change is inevitable, rather than attempting to prevent it, and hence ensure that a system is as simple and easy to change as possible. Refactorings [4] are program transformations improving the structure of code, without modifying its behavior, making it easier to maintain. The research field of Software Configuration Management (SCM) [5] built tools to ease versioning, configuring and building large software systems. Finally, reverse engineering environments [6] use the structure and the history of a system to ease a subsequent reengineering effort, the history being often extracted from a versioning system. Indeed, the history of a system contains valuable information [7], [8].

We approach the problem of software evolution at a more fundamental level. Our thesis is that *an explicit model of software evolution improves the state of the art in software engineering and evolution.* Such a model must closely reflect

[1] http://www.cs.jyu.fi/ koskinen/smcosts.htm

the reality of software development, which is very incremental in nature. Hence, our model describes the phenomenon of change itself with great accuracy, *i.e. treats change to the system as a first-class entity.* In the following we describe the model we defined, then present our ongoing validation of it.

2 Our approach

Our thesis is that a change-based model of software evolution is beneficial to software development and maintenance activities. By change-based, we mean that the incremental development activities of maintainers and developers must be modelled as changes to closely reflect what happens.

The model of the evolution of software systems we defined follows these principles (more detail can be found in [9]):

Program Representation: We represent a state of a program as an abstract syntax tree (AST) of its source code. We are hence close to the actual system, at the price of being language-dependent. In our model, a program state (of an object-oriented program), contains packages, classes, methods, and program statements. Each program entity is a node of the tree with a parent, 0 or more children, and a set of properties (such as name, type of the entity, comments, *etc.*).

Change Representation: We represent changes to the program as explicit change operations to its abstract syntax tree. A change operation is executable, and when executed takes as input a program state and returns an altered program state. Since each state is an AST, change operations are tree operations, such as addition or removal of nodes, and modifications of the properties of a node.

Change Composition: Low-level change operations can be composed to form higher-level change operations typically performed by developers. For example, the addition of a method is the addition of the method itself, as well as the statements defined into it. At a higher level, refactorings are composed of several "developer-level actions" (i.e. renaming a method actually modifies the renamed method and all methods calling it). At an even higher level, our model includes development sessions, which regroup all the changes performed by a developer during a given period of time.

Change Retrieval: This model of the change-based evolution of programs is extracted from IDE interactions of programmers, and stored into a change-based repository. Advanced IDEs such as Eclipse, VisualWorks or Squeak contain enough information and the necessary infrastructure to gather the evolutionary data. This repository is then accessible to tools which can use it to provide useful information to programmer or help him or her performing change requests.

3 Validation

To validate our ideas, we are employing the following 4-step approach:

1. Comparison to other models of evolution: We first surveyed existing models of evolving software used by evolution researchers. These models are closely based on the underlying model of versioning systems, hence we reviewed those in [10]. We concluded that versioning systems used by researchers (because of their popularity among developers, such as CVS or SubVersion) all have similar characteristics: They version systems at the file level to be language-independent, and take snapshots of the system at the developer's request. [10] also outlines why these two characteristics make them not accurate enough to perform finer-grained evolution research.

2. Feasibility: We implemented our model and populated it by developing an IDE plug-in which listens to programmer interactions in the IDE and stores the changes corresponding to these interactions. This IDE plug-in was implemented for the Squeak Smalltalk IDE. It has been ported to VisualWorks Smalltalk, and is being ported to Eclipse for the Java language.

3. Analysing evolution: We then used the data gathered in our repository to perform software evolution analysis, in order to improve program comprehension and reverse engineering. We implemented several visualization and exploration tools on top of the repository. These tools show promising improvements with respect to traditional software evolution analysis based on versioning system data. The information we gather is more precise. Since each change is analysed in context, origin analysis [11] is simplified. We can record precise time information for each change, and reconstitute accurate development sessions [12], whereas traditional approaches can only recover the end result of a session. Analysing such precise sequences allowed us to define new metrics on the sequence of changes itself, measuring for example the activity (number of changes per hour), or the number of changes per entity.

4. Assisting Evolution: If we can record changes to a system, it is also possible to generate these changes. To validate our model in a forward engineering context, we will implement tools designed to ease changing the software itself, using our change-based representation. Several tools can be implemented to validate our approach. One obvious possibility is the implementation of a language-level undo system. We also think a change-based representation could act as a common platform between refactoring tools and other program transformation tools [13]. Code clones could have changes to one instance applied to other instances, in the spirit of [14].

4 Conclusion

To reflect on the phenomenon of software evolution more accurately, we introduced a model of the evolution of programs, in which changes are first-class entities. We represent programs as ASTs and their history as change operations on the AST. These change operations can be composed to represent higher-level

changes such as refactorings, or development sessions. Our ongoing validation shows that our approach recovers more information than found in classical versioning system repositories. However, our approach has several drawbacks: it is language-dependent and requires the presence of an IDE to be accurate. To pursue our validation we are currently porting the approach from Smalltalk to Java, in order to isolate the language-independent parts of our model. We are also building tools exploiting our change-based model to assist software evolution rather than analysing it.

References

1. Lehman, M., Belady, L.: Program Evolution: Processes of Software Change. London Academic Press, London (1985)
2. Purushothaman, R., Perry, D.E.: Toward understanding the rhetoric of small source code changes. IEEE Transactions on Software Engineering **31** (2005) 511–526
3. Beck, K.: Extreme Programming Explained: Embrace Change. Addison Wesley (2000)
4. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison Wesley (1999)
5. Estublier, J., Leblang, D., van der Hoek, A., Conradi, R., Clemm, G., Tichy, W., Wiborg-Weber, D.: Impact of software engineering research on the practice of software configuration management. ACM Transactions on Software Engineering and Methodology **14** (2005) 383–430
6. Nierstrasz, O., Ducasse, S., Gîrba, T.: The story of Moose: an agile reengineering environment. In: Proceedings of the European Software Engineering Conference (ESEC/FSE 2005), New York NY, ACM Press (2005) 1–10 Invited paper.
7. Gîrba, T., Lanza, M., Ducasse, S.: Characterizing the evolution of class hierarchies. In: Proceedings of 9th European Conference on Software Maintenance and Reengineering (CSMR'05), Los Alamitos CA, IEEE Computer Society (2005) 2–11
8. Zimmermann, T., Weißgerber, P., Diehl, S., Zeller, A.: Mining version histories to guide software changes. In: 26th International Conference on Software Engineering (ICSE 2004), Los Alamitos CA, IEEE Computer Society Press (2004) 563–572
9. Robbes, R., Lanza, M.: A change-based approach to software evolution. In: ENTCS volume 166, issue 1. (2007) 93–109
10. Robbes, R., Lanza, M.: Versioning systems for evolution research. In: Proceedings of IWPSE 2005 (8th International Workshop on Principles of Software Evolution), IEEE Computer Society (2005) 155–164
11. Tu, Q., Godfrey, M.W.: An integrated approach for studying architectural evolution. In: 10th International Workshop on Program Comprehension (IWPC'02), IEEE Computer Society Press (2002) 127–136
12. Robbes, R., Lanza, M.: Characterizing and understanding development sessions. In: Proceedings of ICPC 2007. (2007) to appear
13. Robbes, R., Lanza, M.: The "extract refactoring" refactoring. In: ECOOP 2007 Workshop on Refactoring Tools. (2007) to appear
14. Duala-Ekoko, E., Robillard, M.P.: Tracking code clones in evolving software. In: ICSE. (2007) 158–167