Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master Thesis

Description and Schedule

Fine-grained Software Version Control Based on a Program's Abstract Syntax Tree

Martin Otth

Supervisors:
Prof. Dr. Peter Müller
Dimitar Asenov

Chair of Programming Methodology
Department of Computer Science
ETH Zurich

1. April, 2014

1 Introduction and Motivation

Increasingly larger software engineering projects require, without any doubt, multiple developers. As a consequence it is essential to manage and coordinate the changes performed by several developers to the same source code. This coordination work is supported by a version control system.

During the years, software development evolved its paradigms from an unstructured approach towards structured approaches with higher abstraction, such as object-oriented programming. Despite research in software development methods, modern programming tools like IDEs are still based on a text editor. As a result, still today, a program remains a collection of text files. A version control system for such an IDE works directly on the text file. The smallest change unit which is recorded by these systems is usually a text line. One project which aims to supersede the text-based approach is Envision - an IDE for object-oriented languages that features a visual structured code editor and is used for large-scale software development [1]. In Envision a program is internally not represented as a text file but as an abstract syntax tree. Thus a version control system which works directly on the abstract syntax tree of a program is desirable.

The use of the abstract syntax tree in version control results in several benefits. It is possible to narrow down conflicts occurring during concurrent modifications more precisely to a particular construct. Also the issues connected to formatting and white spaces disappear, which results in less false positive conflicts.

The goal of this project is to design a fine-grained version control system for Envision which works directly on the abstract syntax tree of a program.

1.1 Related Work

This section gives a short overview of related work in the area. Software version control is an essential part of software configuration management. Whereas software configuration management also covers issues connected to product management, version control only deals with the version management of one single product. Conradi and Westfechtel provide a classification of versioning paradigms and define fundamental concepts used in software configuration management [3]. One of the concepts they point out is the difference between a state-based version system and a change-based version system. A state-based approach works only on the different versions without additional information on the performed changes. In contrast a change-based version system works with a baseline and a recording of the change history.

Chawathe et al. proposed a state-based approach for extracting changes in hierarchically structured documents [2]. Their algorithm is designed to work on LaTeX documents. In structured text documents the leaves of the syntax tree cover most of the text semantics. But considering source code also inner nodes carry valuable information. Fluri et al. adapted the version by Chawathe et al. in such a way that the algorithm performs well on source code [4].

Kögel et al. proposed an operation-based approach which reliably detects conflicts between composite changes [5]. By defining a *conflicts* relation and a *requires* relation they can define a conflict detection strategy pattern which is used by a conflict detector. The *conflicts* relation defines when two operations are conflicting and the *requires* relation states that one operation can not be applied without the other. They achieve significantly less false positive conflict detections compared to a traditional file-based approach.

2 Core Tasks

This project is split up in several core tasks.

2.1 Analysis of existing work / Requirements definition

In order to perform version control between two versions of the same program we need to be able to understand differences between their abstract syntax trees. Previously conducted work in this area yields two different classes of approaches [3]. It is necessary to analyse these approaches and, if possible, adapt them to the context of Envision. If this is not possible a new approach needs to be found. The two kinds of approaches are:

- **State-based:** Only the current state of the model is stored. By comparison of two states of the model the differences are derived. With the differences it is possible to retrieve the changes between two versions up to some precision. This approach can be used directly on the abstract syntax tree [4].

- **Change-based / Operation-based:** The changes performed to some baseline to achieve a new version are recorded as they occur and stored. In this approach a change is seen as operation transforming one version to another version. As a consequence there is no need to derive the differences at a later point in time [5,6].

A state-based approach requires less memory but the change history is only computable to a certain approximation. In contrast a change-based approach clearly provides more information on the change history but requires careful serialization when applying to a baseline. One goal of this stage of the project is to analyse currently available approaches as well as their advantages and disadvantages. Another goal is to understand the requirements of the Envision system concerning software version control and define these requirements. The insights and requirements will be documented in the final report.

2.2 Solution design

The insight from the analysis is then used to design a theoretical approach which works in the context of Envision. Concerning the design of the solution there are several important questions.

- **Storage strategy:** Define a storage system to store several versions of the same source code. The storage strategy depends highly on the version control approach.

- **Comparisons:** Define how two different versions of the same program are compared. The goal of the comparison is to output all differences as well as possible conflicts. This requires the definition of the delta granularity, which is the smallest possible change unit recorded by the system.
- **Conflict resolution:** Define how to resolve conflicts which occur when merging two software versions. When merging, the application of one change can turn another change inapplicable. This conflict needs to be resolved either automatically or by user interaction. The level at which conflict resolution is possible depends on the delta granularity during the comparison of two versions.
- **Visualization:** Design basic visualizations which can act as a proof of concept for the designed solution.

All design choices and their motivations will be documented in the final report.

2.3 Implementation

The final step towards a prototype is the implementation stage. The following components will be implemented.

- **Additional storage:** Envision works already with the abstract syntax tree of a program. Additional storage will be required to store different versions of a program, namely to store several abstract syntax trees. Depending on the version control approach additional information might be needed. Such information can be recordings of software changes or operations which denote state transitions between two software versions.
- **Comparison:** The comparison functionality takes two versions of a software and finds differences between the two versions. The resulting list of differences is used to track down possible conflicts between two software versions.
- **Conflict resolution:** Gives the user the opportunity to resolve the conflicts which are detected during the merging of two versions by providing conflict resolution functionality.
- **Visualization:** Basic visualizations allows the user to see conflicts and resolve them within the interface of Envision. Differences and conflicts will be shown in a text-based list.

Decisions regarding the implementation and specific important implementation details will be documented in the final report.

3 Possible Extensions

As soon as the development of the core task is completed and properly documented. The following two extensions can be tackled.

3.1 Reusing an existing file-based version control system

Implementing a complete software version control system which is not limited to core tasks from scratch is, due to time limitations, simply not possible in the context of a master thesis. However, it might be possible to reuse an already existing file-based version control system, such as Git. The goal of this extension is to investigate how to use one of these powerful distributed version control systems as a back end to the developed system in Envision.

3.2 Advanced Visualizations

The core task provides basic functional visualizations of the software version control system. The goal of this extension is to design and implement advanced visualizations for the software version control system in Envision. These visualizations should be more convenient and intuitive for the user and allow an easy understanding of the evolution of a software system. Advanced visualizations should also allow the user to interact with the GUI of Envision in an easy and understandable way to perform version control, conflict detection and conflict resolution. Some examples are:

3.2.1 Visualization of Differences

The visualization of differences between two different versions of a source code can be done by using several approaches, such as:

- The code fragments which are different in the two versions will be highlighted directly in Envision's graphical source code view.
- A list showing the differences is integrated in the IDE. An advantage of the list is that it features a clear summary of all differences between the two versions without showing too much detail.
- Side by side comparison of two versions directly connects the matching differences visually.

3.2.2 Visualization of Conflicts

Some advanced visualizations and methods which can be used for conflicts are:

- Ability to chose one of the code versions or write an own conflict resolution directly inside Envision.
- The Conflicting code will be highlighted in an eye-catching manner. E.g. using a red background color in those code areas.

4 Time Schedule

The following table shows the estimated time required for each phase of the project.

Task	Phase	Stage/Component	Time
Core	Analysis	Analysis of available work	1/2 month
Core	Analysis	Envision's requirements	1/4 month
Core	Analysis	Report Analysis & Requirements	1/4 month
Core	Design	Solution design	3/4 month
Core	Design	Design report	1/4 month
Core	Implementation	Implementation Comparison	3/4 months
Core	Implementation	Implementation Conflict Resolution	2/4 months
Core	Implementation	Implementation Visualization	2/4 months
Core	Implementation	Implementation report	1/4 month
Core / Extensions		Extensions & Extensions report	1 1/2 months
Core / Extensions		Finalization report	1/2 months
			6 months

Meetings with the supervisors will take place roughly once a week. During the project three presentations are scheduled. An initial presentation to inform the research group, an intermediate presentation to show current results and to discuss them with the group and, at the end of the project, the final presentation showing the achieved results.

References

[1] D. Asenov. Design and implementation of envision - a visual programming system. Master's thesis, ETH Zürich, 2011.

[2] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, pages 493–504, New York, NY, USA, 1996. ACM.

[3] Reidar Conradi and Bernhard Westfechtel. Version models for software configuration management. Technical report, ACM Computing Surveys, 1995.

[4] Beat Fluri, Michael Wuersch, Martin PInzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng.*, 33(11):725–743, November 2007.

[5] Maximilian Koegel, Markus Herrmannsdoerfer, Otto von Wesendonk, and Jonas Helming. Operation-based conflict detection. In *Proceedings of the 1st International Workshop on Model Comparison in Practice*, IWMCP '10, pages 21–30, New York, NY, USA, 2010. ACM.

[6] Ernst Lippe and Norbert van Oosterom. Operation-based merging. In *Proceedings of the fifth ACM SIGSOFT symposium on Software development environments*, SDE 5, pages 78–87, New York, NY, USA, 1992. ACM.