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

Martin Otth

Supervisors:
Dimitar Asenov
Prof. Dr. John Tang Boyland
Prof. Dr. Peter Müller

Chair of Programming Methodology
Department of Computer Science
ETH Zürich

1. October 2014
"No, our only chance to repair the present is in the past, at the point where the time line skewed into this tangent."

Dr. Emmett "Doc" Brown

*Back to the Future Part II*
Abstract

The Envision project aims to develop an integrated development environment (IDE) for object-oriented languages that features a visual structured code editor and is used for large-scale software development. To achieve this goal Envision works directly on an abstract syntax tree instead of a text-based source code representation.

This thesis features the design of a version control system based on an abstract syntax tree designed for Envision. The resulting system is more fine-grained than traditional text-based systems since changes are tracked on the basis of nodes in the abstract syntax tree and not lines as in a text file.

Nevertheless traditional text-based systems contain functionality which can be used as foundation for a version control system for abstract syntax trees. Therefore Envision’s version control system is build on top of a Git back-end. The system features a comparison algorithm which is able to detect move operations, a history functionality which is able to track substructures and a merge algorithm which automatically resolves conflicts on list types.

The major benefit of a version control system based on an abstract syntax tree is its syntax awareness. As a consequence the comparison algorithm is not prone to formatting changes and the merge algorithm produces always a syntactically correct program. In addition a more fine-grained change categorization in terms of granularity and change types positively affects the quality of the comparison, history and merge algorithms. Further visual improvements in the presentation provide the user with more essential information.
Contents

1 Introduction 10
  1.1 Motivation ................................................. 10
  1.2 Related Work .............................................. 12
    1.2.1 Overview .............................................. 12
    1.2.2 Change Detection .................................... 14
    1.2.3 Merge .................................................. 15
    1.2.4 Refactoring ........................................... 16

2 Requirements and Analysis 17
  2.1 Requirements on a Version Control System .................. 17
  2.2 Using a Traditional Text-Based Back-End .................... 18
  2.3 Requirements on the Interface ................................ 19
    2.3.1 User Interaction ..................................... 19
    2.3.2 Visualization ......................................... 20

3 Algorithmic Approach 21
  3.1 Overview .................................................. 21
  3.2 Universally Unique Identifiers ............................... 21
  3.3 Diff ........................................................ 23
    3.3.1 Change Categories .................................... 23
    3.3.2 Node Matching ........................................ 27
    3.3.3 Processing Text-Based Diff Data ....................... 27
    3.3.4 Efficient Marking of Child Structure Updates .......... 30
  3.4 History .................................................... 31
    3.4.1 Revision Graph ........................................ 31
    3.4.2 Tracking of Subtrees .................................. 31
    3.4.3 Detecting Relevant Revisions ......................... 33
  3.5 Merge ...................................................... 33
    3.5.1 Merge Scenarios ...................................... 34
    3.5.2 General Merge Setting and Idea ....................... 36
    3.5.3 Conflict Detection .................................... 37
    3.5.4 Automatic Merge ..................................... 42
    3.5.5 Manual Merge ......................................... 42
  3.6 Renaming .................................................. 44
    3.6.1 Detection .............................................. 44
    3.6.2 References ............................................ 44
3.6.3 Renaming Support in Core Algorithms

4 Visualizations and User Interactions
4.1 Diff
4.2 History
4.3 Merge

5 System Design
5.1 Git Back-End
5.1.1 Basic Git Internals
5.1.2 Libgit2
5.2 Core Classes
5.2.1 Focus on Diff
5.2.2 Focus on History
5.2.3 Focus on Merge

6 Evaluation
6.1 Diff Tool
6.2 History Tool
6.3 Merge Tool

7 Future Work
7.1 Partially Completed Work
7.2 Further Directions

8 Conclusion
# List of Figures

1.1 Git Diff Excerpt: Stable Brackets .......................... 11
1.2 Git Diff Excerpt: Nested If-Statements .................. 12

3.1 Envision’s Version Control System .......................... 22
3.2 Insert Operation .............................................. 23
3.3 Delete Operation .............................................. 24
3.4 Move Operation ................................................. 24
3.5 Reorder Operation .............................................. 25
3.6 Change Classification Overview .............................. 26
3.7 Envision’s Source Tree Format ............................... 28
3.8 Envision’s Source Tree Example ............................. 28
3.9 Line-based Diff on Envision’s Abstract Syntax Tree ........ 29
3.10 Line-based Diff without Parent Node ....................... 29
3.11 History .......................................................... 32
3.12 Tracking Insert Change Type ................................. 33
3.13 Diamond-shaped History Graph .............................. 34
3.14 Merge scenario: Already up-to-date ....................... 35
3.15 Merge scenario: Fast-Forward ............................... 36
3.16 Merge scenario: Fast-Forward with additional commit ....... 36
3.17 Merge scenario: True merge situation ..................... 37
3.18 Merge Setting .................................................. 37
3.19 LoopStatement ................................................ 39
3.20 LoopStatement with Conflict Units ....................... 40
3.21 Conflict Region Example .................................... 41
3.22 Automatic Merge .............................................. 43
3.23 Manual Merge with Conflict Region Swapping .......... 45
3.24 Line-based Diff of a Renaming in Envision’s Source Tree 45
3.25 References inside Envision’s Source Tree ............... 46
3.26 Renaming Support in Merge ................................. 47

4.1 Comparison Mock-up .......................................... 49
4.2 History Mock-up ............................................... 51
4.3 Merge Interaction Model ...................................... 52
4.4 Merge Mock-up ................................................. 53

5.1 Git Objects* .................................................. 55
5.2 UML of Envision’s Version Control System ............... 56
5.3 UML of Diff Mechanism ...................................... 58
List of Tables

6.1 History Outputs Using Git’s log Command .................... 71
6.2 Fine-Grained History Outputs Using Envision’s history Command ... 72
List of Algorithms

3.1 Change Type Classification ........................................... 26
3.2 Node Matching ......................................................... 27
3.3 Diff ................................................................. 30
3.4 Building a Revision Graph ............................................ 31
3.5 Detect Relevant Revisions ............................................ 35
3.6 Apply Changes to an Abstract Syntax Tree ......................... 44
Acknowledgements

I would like to thank Prof. Dr. Peter Müller for the opportunity to work on an interesting master thesis topic. Furthermore my thanks go to everybody at the Chair of Programming Methodology at ETH Zürich for valuable feedback after presentations.

Also I wish to acknowledge the help provided by Prof. Dr. John Tang Boyland. His insights and suggestions on version control approaches were an immense benefit. I highly appreciate his willingness to spend his time generously in a variety of discussions.

In particular I am very grateful for the assistance and support given by Dimitar Asenov. His advises on programming methodology, visualizations, development techniques and the scientific way of working were of enormous value.

Finally I would like to thank my family and friends for their unbroken support.
Chapter 1

Introduction

This chapter aims at showing the relevance of software configuration management systems and in more detail the importance of revision control. Further it explains the motivation to move from purely text-based approaches towards approaches which are based on the tree structure of a program. The last section discusses related work that builds the foundation of this thesis and was a source of inspiration.

1.1 Motivation

During the years, software development evolved its paradigms from unstructured, close to the system approaches 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. In many modern approaches the text-based storage of source code is an artifact of a text editor and not motivated by state-of-the-art development methods. One project which aims to supersede the text-based approach is Envision\(^1\) - an IDE for object-oriented languages that features a visual structured code editor and is used for large-scale software development [AM14, Ase11]. It features a fast and scalable visual programming interface that can present a program’s structure in a flexible combination of text and graphical objects. Internally a program is not represented as a text-based file but as a source tree. The source tree format used by Envision is an enhanced version of an abstract syntax tree. In addition to the abstract syntax tree the source tree can also contain documentation and diagrams. Unless it is not especially mentioned it is fine to think of the source tree in the same way as of an abstract syntax tree. Envision is specifically designed with professional software engineers in mind and strives to offer flexible visualizations and powerful interaction mechanisms that will boost productivity. In this setting an efficient and powerful version control system is indispensable.

Increasingly larger software engineering projects require, without any doubt, multiple developers. As a direct consequence it is essential to manage and coordinate the changes performed by several developers to the same source code. According to Altmanninger et al. [ASW09] version control systems target mainly three different aims:

\(^1\)http://www.pm.inf.ethz.ch/research/envision
• Maintain the history of the evolution of artifacts like files and directories by logging of changes.

• Support, mainly asynchronous, collaboration between users.

• Manage different development branches.

The need to maintain the history of the evolution of artifacts is not limited to computer science. Nevertheless the revision control systems used in software development are the most powerful and complex ones. The main reason for this progress is that in software engineering lots of developers concurrently evolve the same system, in many cases even exactly the same entities. Furthermore limiting the access to a file to one developer at a time is far less efficient than resolving conflicting changes from several developers by allowing concurrent access [CSFP04].

There are various state-of-the-art version control systems which perform arguably well for source files from text-based development environments. The following two motivating examples show some drawbacks of text-based version control systems and reveal possible room for improvement. Both examples are not artificial and occurred during the development of Envision’s version control system. The Envision project uses a Git repository for versioning. Figure 1.1 and Figure 1.2 show examples of a common problems in text-based systems.

```
-CommitFile::~CommitFile()
+const char* CommitFile::content() const

{  
-   delete[] content_;  
+   if (content_)
+       return content_.get();
+   else
+       return contentWithDeleter_.get();
}
```

Figure 1.1: Git Diff Excerpt: Stable Brackets

The text-based diff detects the brackets { and } as stable lines. Apart from the brackets the two versions do not share any content and are not related in this part of the diff. It just happens that the code lies in the same file.

The limitations showed in the previous figures have several sources. Traditional text-based systems see a line in a file as a unit. Changing a line slightly results in a change of the entire line. For example the if-statements in Figure 1.2 were not completely replaced. Just the condition changed. This information is not visually represented and can not be detected because the if-statement and the condition are located on the same line. Therefore the user has to figure it out. The same also holds for the the deleted and reinserted variable declarations. Both declarations were moved and slightly updated. Another problem is the focus on text and not source code. Source code follows clear structural rules which can be used in a version control system.
if (!persistentUnitGuess.isNull())
-
-
-


- GenericPersistentUnit* unit = persistentUnit(persistentUnitGuess);
- if (unit)
-
-
-
-

+ if (auto unit = persistentUnit(persistentUnitGuess))
+ if (auto node = unit->find(id))
+
+
+

- } return node;
-
-
Figure 1.2: Git Diff Excerpt: Nested If-Statements

The nested if-statement is modified to be more concise. The local variable definitions are moved into the if-conditions and unnecessary brackets are removed. The line-based change detection is suboptimal in this setting because the relocations of the local variables are not detected.

Handling Envision’s source trees directly with a text-based version control system would result in suboptimal results or even worse — invalid source trees. Moreover a version control system based on an abstract syntax tree has several benefits compared to a text-based system. Some of them are:

- No false or imprecise change reports due to file formatting
- A merge results in a syntactically correct output
- Improved comparisons due to finer resolution (smaller units than a text line)
- Provide more power for effective conflict and inconsistency detection [ASW09]

Hence a version control system which performs on a program’s abstract syntax tree is not only demanded by Envision but also could lead to major improvements regarding functionality.

1.2 Related Work

The field of version control systems offers lots of interesting work in a broad context. Section 1.2.1 tries to give a short but rather broad overview on different ideas and approaches present in the field. The subsequent sections focus on work which is closely related to this thesis.

1.2.1 Overview

The terminology and concepts introduced in this overview are based on the surveys of Altmanninger et al. [ASW09], Mens [Men02] and Conradi and Westfechtel [CW98].
As already briefly mentioned one of the problems a version control systems has to deal with are concurrent changes performed by several users. In general there are two approaches to tackle this issue.

**Pessimistic:** The pessimistic approach follows the lock-modify-unlock paradigm. As a consequence only one user at a time is allowed to modify an entity.

**Optimistic:** The optimistic approach follows the copy-modify-merge paradigm where each user works on his own local copy. After the work is completed the local copy is merged into the shared original entity.

Following an optimistic approach requires a merge mechanism. There are several approaches to merge different entities. The distinction here shows three different variants.

It is assumed that version A and B are merged.

**Raw:** A raw merge simply applies the sequence of changes which lead from version A to version B. This approach is fairly simple but lacks accuracy. Namely not all conflicting concurrent changes can be detected. In such situations the later applied changes often overwrite the previously applied concurrent change operations.

**Two-way:** In a two-way merge a comparison between version A and version B is used to identify conflicts. But the provided information from the comparison is not enough to detect if both versions or just one version modify an element due to the lack of information on a common base.

**Three-way:** A three-way merge includes the common ancestor version and is a more powerful variant of the two-way merge. The versions A and B are not compared directly but both are compared to the common ancestor to figure out applied changes more accurately.

The comparison methods are one of the most important mechanisms in version control systems. They can be classified into two categories:

**State-based:** A state-based comparison approach requires two versions and a common ancestor as input. No additional log file of the performed operations is required. Differences are then computed by comparing the versions. To identify related entities it is required to have a reliable matching technique.

**Change-based:** A change-based comparison requires additional input in the form of a log of performed changes. Often the term operation-based comparison is used to denote a special form of change-based comparison where more complex high-level operations like renaming are considered in contrast to very fundamental changes like add, update and delete. Change-based approaches feature improved conflict detection due to the exact log of changes in correct order. Nevertheless these approaches require a close coupling to a development environment and the number of operations can become quite large. Some changes might even be reverted later in a log which turns them unnecessary.

Fluri and Gall [FG06] classified high-level operations based on elementary tree changes on an abstract syntax tree. The fundamental changes they use are insert, delete, move and update.

Wagner and Graham [WG97] propose a fundamentally different approach. Whereas most other approaches use time as primary index and store an entry at desired points in time they use space as primary index. In this way each versioned object records its own history.
as a sequence of applied operations. To support fine-grained change capture the ability to efficiently produce persistent linked data structure is required. A document is represented as a collection of nodes which have content stored in versioned or unversioned fields. Every version of the document is identified with an unique integer which is stored in a global version tree. The theoretical foundation of Wagner and Graham's work is the concept of the 'fat node' for persistent linked data structures developed by Driscoll et al. [DSST89].

Nguyen et al. [NMBT04,NMBT05a,NMBT05b] developed Molhado to enhance the Software Concordance environment. With Molhado it is possible to manage the evolution of software at several different structural levels. The fine-grained tools support the comparison of structural units, documents and entire projects. The Molhado system is based on the concepts used by Wagner and Graham. Unlike other systems Molhado focus on versioning an entire system and not just a collection of units. This includes the versioning of software artifacts like XML documents, graphics and UML diagrams as well as links. Molhado also features simple visualizations for the comparison of project structures like documents and directories.

1.2.2 Change Detection

In List Structures

Lists structure the information they contain as a sequence. Content is stored at a position in the list which is referenced by an index. Approaches which detect changes in lists need to feature some robustness against the shifting of content in the list. For example: inserting a new item at the beginning of a list results in a shift of the entire content by one index position. However only the first element is newly added. All other items in the list remain unchanged. These algorithms build the foundation for text-based version control systems since a text file is nothing but a list containing strings. One of these algorithms is Diff3. A Unix utility tool to compare and show differences in files. The tool performs a three-way comparison based on the detection of a longest common subsequence. Khanna et al. [KKP07] did a formal investigation on the properties of Diff3.

In Tree Structures

Trees are a form of hierarchically structured information. In such structures changes do not only affect the values stored in the nodes but also their relationship. In flat information structures the deltas between two structures may be reasonably well represented by using an insert notion, a delete notion and an update notion. For hierarchical structures it is of interest to relocate a substructure without changing each node of the substructure. Hence the notion of a move operation is required. Chawathe et al. [CRGMW96] describe an approach which does exactly this. Their approach does not rely on additional information like object identifiers or change logs. Hence they solve two problems which are at the essence of change detection in tree structures. First they have to find a matching between nodes in two versions. Afterwards they have to categorize the deltas based on the found matching. To find a matching they define a matching criterion for leaf nodes based on value similarity for leaves which carry the same label. A second matching criterion is defined for internal nodes which is a threshold on the percentage of their matching leaves.
Additionally their approach is based on two assumptions. They assume the existence of an ordering on the labels of the nodes and that a value stored in a leaf is similar to at most one leaf in the other version. Under those two assumptions a unique maximal matching exists and can be found by a simple algorithm. The resulting implementation is called *LaDiff* and is designed with a focus on LaTeX documents. To extract the edit script the approach uses several phases. Each designed to detect a specific change type. These phases are: update phase, align phase, insert phase, move phase and delete phase. During the align phase the correct sequence of children for matching nodes has to be established in the edit script. To find the shortest sequence of moves to align the children a longest common subsequence is used. But Chawathe's approach performs suboptimally on source code since the second assumption may not hold in this case. Several leaves in an abstract syntax tree often do carry the same values.

Fluri et al. [FWPG07] improved the algorithm by Chawathe et al. in a program called *ChangeDistiller* to solve this problem. They focus on improving the matching procedure. According to them the major shortcomings of the original algorithm are: “1) inadequate matching of node values, 2) using the first match instead of finding the best match, and 3) the propagation of mismatches in small subtrees.” The matching of node values is improved by using *n-Grams* as string similarity measure [AB74]. Additionally the Dice Coefficient [Dic45] is used to calculate the matching of inner nodes. Baxter et al. [BYM+98] used this approach for code clone detection. A weighting is used to tailor the balance between inner nodes and leaves in the matching algorithm. By introducing a best match criteria in the algorithm and adapting the thresholds on the matching for smaller subtrees they achieve an improved result on source code.

Nguyen et al. [NNP+09b] follow a similar approach with the development of *Treed* but with a different matching strategy. *Treed* tries to keep as many nodes as possible, rather than deleting and inserting nodes. Hence they use the comparison feature in Subversion (SVN) to compare text-based representations of the trees. In a first step the unchanged leaf nodes are mapped and after that a mapping between changed leaf nodes is computed. Inner nodes are mapped in a bottom-up fashion using Exas characteristic vectors. Exas is a structural similarity measure on trees [NNP+09a]. Some nodes might still be unmapped after this procedure due to severe changes. Those nodes are finally mapped in a top-down manner and compared based on the already mapped nodes.

### 1.2.3 Merge

Kögel et al. [Koe08,KHvWH10] follow a diff-and-apply approach for merging. First they extract a sequence of change operations by computing differences between versions. Then the extracted changes are applied to the state of the local workspace. During that process they are able to detect two kinds of conflicts. The first type is caused by concurrent changes which result depending on the order of serialization in a different outcome. The second one is caused by operations which would through their application violate the integrity of the model or operations which are just not applicable to the current state of the model. By defining a *conflicts* relation between operations they are able to detect directly conflicting operations. But conflicting operations can not be simply discarded in a merge setting since there might be operations which are dependent on the conflicting operations.
Therefore they introduce a \textit{requires} relation that determines whether a operation can not be applied without another operation. Their resulting conflict resolution process has two phases. First, for each operation the user decides to include they first calculate the transitive closure of the \textit{required} relationship to decide which operations should be accepted. By then inspecting the \textit{conflicts} relation they find directly conflicting operations which should be rejected. Additionally all operations which require such a conflicting operation need to be rejected too. This set of operations is again computed by the use of the transitive closure. Second, all operations which are not rejected are applied to the workspace. The operations which are rejected are either not applied to the workspace or are reverted.

1.2.4 Refactoring

Dig et al. [DMJN08, DMJN07] extended the Molahdo system to be refactoring-aware and called the extension MolhadoRef. The system follows an operation-based approach by representing a version as a sequence of change operations. During a merge this operations are replayed while treating refactorings separately. Their work distinguishes between refactorings and edits. Refactorings are change operations that preserve the semantics of a program while edits usually change the semantics. The merge algorithm works in five steps. Step 1 detects the changes that happen in version $V_1$ and version $V_2$ by comparing the versions to a common base version. The categorization of edits and refactorings is built with the use of the extracted changes and additional refactoring logs. Step 2 detects conflicts and searches for dependences between operations. Step 3 inverts the refactorings in in version $V_1$ and $V_2$ by applying additional refactorings. Step 4 merges the version $V_1$ and $V_2$ which do not contain any refactorings. Step 5 replays all previously inverted refactorings in the correct order regarding dependences. The core idea of MolhadoRef is to shift refactorings to the end of a change sequence and therefore obtain no partially completed refactorings.

Kawrykow and Robillard [KR11] build a system to detect non-essential changes in version histories. They define non-essential differences as being cosmetic in nature, generally behavior-preserving and unlikely to yield further insights into the program. To detect such differences they need a level of granularity finer than statement differences and type-sensitive. Their techniques are implemented in a tool called DiffCat. The tool uses partial program analysis to recover missing binding information and ChangeDistiller to facilitate the detection of fine-grained differences between abstract syntax trees. Based on manual investigation they state that DiffCat classified 98.8\% correctly as non-essential. In an empirical study they additionally discovered that between 2.8\% and 22.9\% of modified code lines were non-essential. Of these non-essential updates more than 85\% were rename induced updates.
Chapter 2

Requirements and Analysis

This chapter states a selection of the essential requirements for a version control system in the context of Envision. In the second section an analysis is presented that focuses on how these requirements can be fulfilled by using a traditional text-based back-end. Additional requirements are presented in the last section of the chapter which intend to direct the development of visualizations and user interactions.

2.1 Requirements on a Version Control System

As a modern software development tool Envision has to be able to deal with large projects containing an enormous amount of code. Since Envision is a tool which works directly on the source tree of a program some of the requirements differ from the ones used in text-based systems. However the typical properties of a version control system are assumed and the following requirements are a selection of the essential ones in the context of Envision.

Commit linearly: Envision needs a way of storing new versions of a project. The new versions need to be stored in such a way that a link to previous versions can be established.

Branching: It must be possible to diverge from the current track of development at any point by creating a new branch. A branch is a new independent track of development and must behave like an independent copy of the project.

Merging: Envision requires a procedure which joins two development tracks into a single one. Therefore functionality is required to join two source trees in such a way that the resulting merge is still a valid abstract syntax tree. Reliable detection of conflicts is a must have and flexible and user-friendly support for conflict resolution is desired.

History: A mechanism must be available to view the evolution of an entire project. Additionally it is desired that the evolution can be viewed in a fine-grained manner by only looking at certain substructures. In this case the history should only contain versions which are relevant for the selected substructure.

Comparison: A comparison mechanism is needed to compare two versions of an Envision project. The result of a comparison should contain more user-friendly information than
traditional comparison approaches. Hence more fine-grained change categorization is desired.

**Commit information:** Each version stored should hold some additional information. Required are a message, date and time of the version submit and information on the user which performed the submit.

**Labels / Tags:** There should be a mechanism to mark a specific commit in the commit history with a tag. The tag should carry additional information such as a message or information on the user who tagged the commit.

**Support Refactoring:** The version control system should have smart support for common forms of refactoring like renaming.

**Relocation / Reordering:** The relocation of nodes in Envision’s source tree should not result in combinations of insert and delete operations but result in a relocation operation.

### 2.2 Using a Traditional Text-Based Back-End

There are quite a few well-established text-based version control systems which feature a lot of the functionality that is required in section 2.1. Many of them perform well on different platforms and often have some sort of web-based hosting service. Therefore it is reasonable to analyse to what degree a traditional text-based versioning system can be used on Envision’s source trees. The analysis in this thesis has a focus on Git, a state-of-the-art distributed version control system.

**Commit linearly:** As a distributed revision control system, Git has a fast repository architecture since most of the operations are performed locally. This includes commits. Hence this requirement is fully covered by Git without any further work.

**Branching:** Git is known for its very lightweight branching mechanism. Branching for the storage of abstract syntax trees does not differ from branching for text-based documents.

**Merging:** Git’s line-based merge mechanism might result in invalid source trees. Hence the mechanism can not be reused and the merge concept has to be improved and adapted to Envision’s source tree.

**History:** The log feature of Git is good enough to track the history of an entire project. However if a substructure of the project needs to be tracked then the granularity of Git’s log feature is too coarse. It is only possible to track the history of files but not of the content inside a file. Hence additional effort is required.

**Comparison:** The line-based comparison mechanism of Git can be reused but is not enough in its current state. The information of inserted and deleted text lines is not appropriate for Envision’s source tree.
Commit information: Every commit in Git has a message, information on the committer and the author and a timestamp. The information on the committer and author contains the name and email of the users. Therefore Git already supports enough commit and user information.

Labels / Tags: Git already features good support for tags. There are two forms of tags. A lightweight one and annotated one. The annotated form allows the use of a message and additionally stores a timestamp and user information at the creation of the tag. These tags can also be signed with GNU Privacy Guard. Hence no additional measures are required here for Envision.

Support Refactoring: Git does support the renaming of files in the repository but apart from that no refactoring support is provided by Git. This is clearly not enough and Envision’s version control system has to improve on this requirement.

Relocation / Reordering: Git does not feature any support for the relocation of content inside a file or even across files.

This brief survey shows that Git already offers a lot of the functionality which is required in Envision’s version control system. Therefore the developed version control system uses Git as a base and builds additional functionality where the functionality of Git is not covering the requirements.

2.3 Requirements on the Interface

As a visual structured code editor Envision also has requirements on the visualizations and the user interactions for the version control system since those are key for the success of the project. The following requirements try to capture the most essential design ideas and are motivated by the main goals and core concepts on which Envision is built

2.3.1 User Interaction

Convenience: The user interactions should be convenient and intuitive to use. In other words the commands and interactions design for the version control mechanisms should be simple to use. Frequently used operations, like the diff between the current work directory and the latest commit, should be quick and easy to perform. In addition the tools for more complicated tasks, like a merge of two versions, should provide interactions which are very efficient to use.

Abstraction: It is essential that the interaction with Envision’s version control system are integrated directly into Envision’s user interface. The developer should not need to worry about the underlying mechanism in the version control system nor should the user care about the realization of the back-end. In other words the user should not feel any difference when loading a version between loading from a file on disk or loading from the repository. Generally said the user should not need to understand how a version control system works in detail.

\footnote{http://dimitar-asenov.github.io/Envision/motivation.html}
2.3.2 Visualization

**Comprehensibility:** The visualizations should simplify the understanding of the evolution of a software system. Important content should be stressed while inessential content should be hidden. For example, the comparison of two versions of a method should not involve differences outside of the method.

**Rich visuals:** The visualizations should not be limited to text and should feature a mixture of textual and visual elements. Different colors should be used during the visualization of a diff to indicate various kinds of differences. Moved code, for example, could be visualized by the use of connecting lines.
Chapter 3
Algorithmic Approach

This chapter contains the key ideas behind the core algorithms which are used in Envision’s version control system. For the sake of comprehensibility the presented algorithms are simplified to their very essence. More detailed information on implementation and Envision’s system design can be found in chapter 5.

3.1 Overview

As seen in section 2.2 a text-based version control system already fulfills a substantial amount of Envision’s requirements presented in section 2.1. Therefore it is reasonable to establish Envision’s version control system based on a text-based system. Similar to the analysis in section 2.2 this chapter assumes a Git back-end. Under slight adaptations most of all traditional text-based version control systems could be used as a back-end. Figure 3.1 shows the overall architecture of the resulting system. Additional algorithms to the ones provided by Git are required to perform reasonable comparisons, track the history of objects and merge Envision’s source trees (similar to abstract syntax trees). The algorithmic details are explained in the following sections. No additional log files are stored into the Git repository, only the abstract syntax trees. Hence the system follows a purely state-based approach. The result is a powerful state-based fine-grained version control system for abstract syntax trees.

3.2 Universally Unique Identifiers

A reliable, precise and complete comparison mechanism is one of the core tools a version control system needs. The produced change descriptions are the foundation for more complex tasks like specialized history logs, conflict detection and resolution in a merge scenario. In a state-based system the input to a comparison algorithm are two versions of the abstract syntax tree. Additionally a common ancestor can be found but no additional information in form of a change log is available. Hence the system highly depends on matching techniques to identify similar structures in the abstract syntax tree.

In general there are two approaches to establish a suitable matching. One is to use universally unique identifiers (UUID) inside the abstract syntax tree for each node. The
Envision is enhanced with a version control system via VCS Manager. Regular Git repositories are used as back-end. All access to the Git repositories from Envision operate through the VCS Manager. This approach isolates Git from the remaining system.

Another one is to establish the IDs by using a heuristic based on similarities inside the abstract syntax tree.

**Universally unique identifiers:** Each node inside the abstract syntax tree has a UUID which is established at node creation. Two nodes in different versions with the same UUID are considered as matching. The advantages of this approach are its simplicity, efficiency and scalability. Additionally two nodes can be matched even if they went through major changes and the only similarity remaining is the UUID. But the IDs have to be available for all elements which essentially requires a close coupling to the development environment.

**Heuristic:** The idea behind a heuristic is to find a matching by the use of some similarity measure. Several approaches have been proposed which define a similarity measure for leaf nodes and one for inner nodes. These measures are then used to build the best possible matching [CRGMW96, FWPG07, NNP+09b].

A heuristic does not require direct support from the development environment which is a clear advantage. However, the matchings might not be unique and might fail under severe changes. In addition the performance may also be an issue.

While heuristics may fail to capture drastic changes to a structure the UUID approach is still able to match the structures. Depending on the situation this might be desirable or unwanted. However it is possible to combine both approaches and use the advantages of both mechanisms [ASW09].

Envision’s source tree already contains unique IDs for each node in the tree. The format used is a 128-bit number that is meant to guarantee that the ID is unique in a distributed computing environment and follows the UUID standard documented in ITU-T Rec. X.667\(^1\).

\(^1\)[http://www.itu.int/rec/T-REC-X.667/en]
3.3 Diff

Comparison algorithms are an important part of a version control system and build the foundation of history logs and merge support. In a traditional text-based notion a diff is used for file comparison and visualizes the differences between two files. In the context of this thesis the diff needs to compare two abstract syntax trees and categorize their differences. This section describes the used change categories and how they are detected.

3.3.1 Change Categories

The change operations defined on an abstract syntax tree are similar to the edit operations defined on ordered trees by Chawathe et al. [CRGMW96]. One difference is the additional distinction between a move operation and a reordering of nodes. Another key difference is a flag which denotes the update of the child structure of a node.

Insert Operation

An insert operation adds a new node to the abstract syntax tree. Since every node has a unique ID the newly added node holds an ID which is not existent in the previous version of the abstract syntax tree. Also the insertion of a new node does affect the child structure of the parent node. Figure 3.2 shows an example of an insert operation in a tree structure.

![Figure 3.2: Insert Operation](image)

The green node with unique ID D is inserted into the abstract syntax tree and therefore only exists in the resulting tree. This change results in a child structure change of node B which is marked blue in the figure.

Delete Operation

A delete operation removes a node from the abstract syntax tree. The unique ID of the the removed node is not present in the new version of the tree. Removing a node also affects the child structure of the parent node. Figure 3.3 shows an example of a delete operation in a tree structure.
The red node \( D \) is removed from the abstract syntax tree and as a result the ID \( D \) only exists in the previous version of the tree. This operation triggers a child structure change in the parent node \( C \).

**Move Operation**

A move operation is a relocation of a node inside the abstract syntax tree. The unique ID of the moved node exists in both, the old and the new version. This operation is characterized by a change of the parent node and can therefore be detected by looking at the parent of the moved node. The child structure of the parent in the previous location and parent in the new location is affected. Figure 3.4 shows an example of a move operation.

The yellow node with ID \( D \) is moved in the abstract syntax tree. During this move operation the parent \( B \) of \( D \) in the old version is changed to \( C \) in the new version. Therefore the child structure of \( B \) and \( C \) is changed and marked with a blue highlight.

**Stationary Nodes**

Nodes which are neither inserted, deleted or moved are called stationary. These nodes exist in both versions of the abstract syntax tree under the same parent.
Update flags

The information in moved or stationary nodes can be updated since the nodes are present in both versions of the abstract syntax tree. Every node in the tree has a type describing its purpose. The edges inside the abstract syntax tree carry labels to identify the different children of a node. The label is not stored on the edges but in the child nodes. Therefore every node in the tree also carries a label. In addition every leaf node in the abstract syntax tree carries a value. Every change on the node information is captured with update flags.

**Label:** An update of the label of a node can be seen as a local move operation under the same parent. Nevertheless this information is kept in a separate update flag. Such an operation does not change the number of children but the order in which they appear in the parent. Hence the child structure of the parent node is affected. The change of the label in a node is called reordering. Figure 3.5 shows an example of a reorder operation.

![Figure 3.5: Reorder Operation](image)

The node with ID B changes its label from $l_0$ to $l_1$. In addition the node C is reordered from label $l_1$ to $l_0$. These operations result in changes in the child structure of blue node A.

**Type:** A type update is a change of the type information of a node. This change does not affect the child structure of the parent.

**Value:** A value update does change the value of a leaf node in the abstract syntax tree. This operation does not affect the child structure of the parent node.

**Child Structure:** A child structure update flag is set as soon as the number or order of a node’s direct children is changed.

Change classification

Once the matching between the nodes is established the change classification only requires the information stored in a pair of matching nodes. From this information a change description is built for every node affected by a change operation. The change type depends on the existence of the unique IDs in the versions as well as the parent nodes of the current node pair. Algorithm 3.1 describes how a change description is computed. The resulting categorization of changes between two different abstract syntax trees is shown in Figure 3.6.
The function ChangeDescription takes two nodes from different abstract syntax trees with identical unique ID as argument. If such a node does not exist in the tree NULL is passed as argument. A structure containing the changeType and the update flags is returned as result. The function ComputeFlags sets the flags for Label, Type and Value update based on the comparison of the information provided with the nodes. The update flag for Child Structure is set at a later point according to section 3.3.4.

Algorithm 3.1: Change Type Classification

The change classification used distinguishes between four different change types. A change description which is of change type Move or Stationary contains a node in both abstract syntax trees. Hence such a description also carries four different update flags.
3.3.2 Node Matching

Since Envision stores a unique ID with each node the matching procedure is simple and efficient. Algorithm 3.2 describes the matching procedure used in Envision.

```
function NODEMATCHING(nnodesA, nodesB)
    onlyInB ← nodesB
    for all nodesA do
        nodeB ← FIND(nodeA.id, nodesB)
        change ← CHANGEDESRIPTION(nodeA, nodeB)
        ADD(change, changes)
        if nodeB then
            REMOVE(nodeB, onlyInB)
    for all onlyInB do
        change ← CHANGEDESRIPTION(NULL, nodeB)
        ADD(change, changes)
    return changes
```

Algorithm 3.2: Node Matching

3.3.3 Processing Text-Based Diff Data

As previously mentioned programs created inside Envision are stored as source tree. The source tree representation on disk contains an enhanced version of the abstract syntax tree of the program. Envision tries to avoid storing the entire source tree in one single file on disk. Therefore persistent units are used to subdivide the source tree into smaller subtrees. Each persistent unit contains one of these subtrees and is stored in a separate file on disk. The source tree is stored in a pre-order fashion while each line holds exactly one node. A line contains the label, type and unique ID of a node. The indentation in front of the label is used to store the depth of a node inside the source tree. This information makes it possible to assign a node to its parent. In case of a leaf node the unique ID is followed by a dot which denotes the presence of a value. The value is stored at the end of a line. Figure 3.7 shows a leaf node which carries a string value. Figure 3.8 shows a larger example of Envision's source tree which also demonstrates the interconnection of nodes in the format.

By looking at a single line inside a file holding the source tree it is possible to extract almost all information needed to classify the change on the node represented by the line. The only information which is not stored directly in the line is the parent. Nevertheless it is safe to say that every change to a node in the source tree has an effect on the line.
A line in a source tree file in Envision representing a leaf node. The node has a label, a type and a unique ID. The dot after the unique ID shows that the node is a leaf and therefore contains a value.

```
0 Method {5b1ee...49c70}
  name NameText {a061a...b7f48}. S_enroll
  items StatementItemList {79e8c...7d4f0}
    0 ExpressionStatement {e4de3...7864e}
      expression AssignmentExpression {c6c34...ab8ef}
        left ReferenceExpression {29427...9a394}
          ref OOReference {801c5...4c56f}. S_{f4d3b...dbf9b}:isStudent
        right BooleanLiteral {d363e...ef4a1}
          value Boolean {42305...de317}. I_1
```

Figure 3.8: Envision’s Source Tree Example

A simplified excerpt of a method stored in Envision’s source tree format. The children of a node are indented one tab more than the node itself and referenced by their labels. For example the AssignmentExpression has two children. A ReferenceExpression under the label left and a BooleanLiteral under right.

holding exactly this node. Changing the parent the line is deleted in one location in the file and inserted in another location under a new parent. Hence it is possible to use a traditional text-based comparison algorithm to narrow down the number of lines which have to be checked for changes. Figure 3.9 shows a text-based diff example of a string value change in a source tree node.

The displayed code section in Figure 3.9 holds all relevant information to fully classify the field renaming. But a single diff excerpt is not always enough to classify a change. In general it is required to look at the entire diff of all files which contain parts of Envision’s source tree since locally, move operations cannot be distinguished from inserts and deletes. Even looking at the entire text-based diff it might not be possible to distinguish a moved node from a stationary node. Because the surrounding content might not contain the required parent node. This is the case for nodes with lots of siblings which are listed in front of the changed node. Figure 3.10 shows an example of such a case.

As seen in Figure 3.10 it is required to look at all diff excerpts to get a complete picture what changes were applied to the source tree. Furthermore some of the required parent nodes might not occur in any diff excerpts. This issue can be solved by an additional traversal of the source tree files which contain nodes with missing parent knowledge. In some cases these parents need to be marked with a child structure update flag hence
Welcome Project {7124a...2cb9b}
   _ext_Position_y Integer {cf5db...9694e}. I_1
fields TypedListOfField {0ec91...be77e}
   0 Field {f4d3b...dbf9b}
-      name NameText {fab00...4b6e3}. S_welcome
+      name NameText {fab00...4b6e3}. S_isHello
modifiers Modifier {6eeed...959e7}. I_10
annotations StatementItemList {874e9...1e204}
subDeclarations TypedListOfDeclaration {dd298...acec8}

Figure 3.9: Line-based Diff on Envision’s Abstract Syntax Tree

Example of a text-based diff performed on a source tree file with Git. On a high level this change represents a renaming of a field from welcome to isHello.

ref OOReference {fe5a0...ecf65}. S_____NULL____:pieceShape
typeArguments TypedListOfExpression {37bc8...83566}
   opr Integer {bcc27...5cb73}. I_0
-      4 ExpressionStatement {7d10b...ce619}
-      expression EmptyExpression {bfbf9...706a0}
typeArguments TypedListOfFormalTypeArgument {5c575...d9ebe}
arguments TypedListOfFormalArgument {8889c...9effd}
results TypedListOfFormalResult {050fd...d6f1b}

Figure 3.10: Line-based Diff without Parent Node

The node with unique ID {7d10b...ce619} and its child {5c575...d9ebe} are removed at this location in the file. The information provided in this diff excerpt is not enough to categorize the change since no knowledge is provided if the nodes are inserted at another location. In addition the parent of the ExpressionStatement node is not part of the excerpt which makes it impossible to flag a child structure update in the parent node.
it is required to add those parents to the change affected nodes. Also special care is required for these extra added parent nodes since they may not be affected by a change but are always stationary. This case occurs for inserted or deleted child nodes. As an example let's consider again Figure 3.10 and assume the ExpressionStatement is actually removed from version A of the source tree to version B. Hence the parent node of node \{7d10b...ce619\} is extra added to the affected nodes in version A. However the list of affected nodes in version B does not contain this parent node unless another change by coincidence affects it. Following the change categorization the parent is only present in the change affected nodes of version A and hence categorized as deleted. This categorization would be wrong. Therefore parent nodes which are extra added to the sets of change affected nodes are always added in both versions.

One of the advantages of using the text-based diff is that the extra tree traversal is only required for files in which changes are detected. Algorithm 3.3 shows how the lines received from a Git diff are categorized into nodes from version A and nodes from version B. The difference is based on the line origin which is marked in git with a - for lines in the file of version A and a + for lines in the file of version B.

```
function Diff(GitDiffLines)
    for all GitDiffLines do
        node ← Node(GitDiffLine.content)
        if GitDiffLine.origin = LineAddition then
            Add(node, nodesB)
        else
            // GitDiffLine.origin = LineDeletion
            Add(node, nodesA)
        end if
    end for
    return NodeMatching(nodesA, nodesB)
```

The algorithm Diff uses as input a data structure storing the content of a Git diff and returns a set of change operations which turns version A into version B.

Algorithm 3.3: Diff

### 3.3.4 Efficient Marking of Child Structure Updates

Directly setting a child structure update flag in the call ComputeFlags in Algorithm 3.1 would require the information on all direct children of nodeA and nodeB. Similar to the previously mentioned knowledge of the parent node this information might not be completely contained in the diff. In addition to decide if the setting a child structure flag is required all changes on the direct children have to be categorized first. Instead of taking care of the child structure of potentially changed nodes and the order in which they are categorized it is simpler and more efficient to delay the placement of the flag until all changes are categorized. By iterating over all change descriptions it is possible to set the flag depending on the change type. The child structure update flag is set for the parents of nodes which were inserted, deleted, moved or reordered. In this approach no tracking of child nodes is required and the order of change detection is irrelevant.
3.4 History

Another useful feature in a version control system are histories or logs. These tools are used to have a look at the evolution of a project. Nevertheless most version control systems only feature logs on an entire project and on files. In many cases a file is still too coarse. This section shows an approach which allows the tracking of arbitrary subtrees inside an abstract syntax tree. As a result it is possible to track the history of methods, for example, even if they change their location across files.

To achieve this a revision graph is computed in a first step which contains all potentially relevant revisions in the repository. In addition all node IDs of the selected subtree are tracked in a set. By traversing the revision graph step by step and computing the differences between connected revisions it is possible to detect if a tracked node is modified. Every time the tracked subtree is modified a new set of tracked node IDs is computed and the revision is added to the set of relevant revisions.

3.4.1 Revision Graph

The revision graph is defined by two revisions inside a Git repository. It contains all commits which occur on a path connecting those two revisions. The graph is built by traversing the commit history in the Git repository. The commits in the Git repository only contain a reference to their parents whereas a commit in the revision graph hold references to child and parent commits.

Algorithm 3.4 shows the function \textsc{RevisionGraph} which is used to build a revision graph from a Git repository. To get an excerpt of the commit history the algorithm is called

\begin{algorithm}
\begin{algorithmic}
\Function{RevisionGraph}{\textit{graph}, \textit{currentRevision}, \textit{targetRevision}}
\If{currentRevision \neq targetRevision}
\ForAll{parents of currentRevision}
\If{targetRevision is ancestor of parent}
Add (currentRevision, parent) to graph
\EndIf
\EndFor
\EndIf
\EndFunction
\end{algorithmic}
\end{algorithm}

The function \textsc{RevisionGraph} takes a graph data structure which is used as out parameter for the revision graph. The \textit{currentRevision} points to the current commit in the Git repository and \textit{targetRevision} is used to mark the beginning of the revision graph. \textsc{RevisionGraph} uses recursive calls to walk backwards through the Git commit history and fills the graph data structure on the way.

Algorithm 3.4: Building a Revision Graph

with an empty \textit{graph}, the start point of the required history as \textit{targetRevision} and the end point as \textit{currentRevision}.

3.4.2 Tracking of Subtrees

The ability to track an arbitrary subtrees through the history of revisions is necessary to support a fine-grained history feature. Only changes which affect nodes inside the subtree
are considered as relevant. Detecting changes to a subtree in the abstract syntax tree
and thereby detecting the relevance of a change requires to track all the nodes inside the
subtree. Figure 3.11 shows an example of a tacked subtree.

![Figure 3.11: History](image)

In this tree example the subtree of node B is of interest. To detect all changes
to the subtree a set of nodes is constructed which contains only the nodes of
the subtree.

The function TrackSubtree takes as argument the node at which the subtree to track
is rooted. By recursively traversing the subtree the algorithm builds the set of tracked
nodes.

It is enough to track the nodes in the subtree to detect relevant changes. This can be
seen by looking at all possible change types applicable to nodes in the tracked subtree:

- **Insert**: Inserting a new node to the subtree cannot be detected directly because
  the node is not yet part of the tacked nodes. But the parent or an ancestor further
  up the tree is marked with a child structure update flag. The node which receives
  this flag update is going to be tracked in the set and hence the change is detected
  when looking at this node. An example of this case can be seen in Figure 3.12.

- **Delete**: The deletion of a node inside the tracked subtree is detected because the
  node is contained in the set of tracked nodes.

- **Move**: In this case an additional distinction is required. Depending on the source
  and target of the move operation one of the following two cases applies.
    - **Target in tracked subtree**: If the node is moved into the tracked subtree then
      the moved node is not yet part of the tracked set of nodes. However the move
      operation triggers a child structure update on its new parent node. Hence the
      reasoning is similar to the insert change type.
    - **Source in tracked subtree**: The moved node is already tracked in the set of
      tracked nodes. Hence the move is correctly detected as relevant change.

- **Stationary**: A stationary node is part of the tracked node set and therefore any
  change to such a node will be detected as relevant.
The nodes tracked in this example are B and its subtree. The newly added node X is not part of the set of tracked nodes. However its parent node Y is part of the subtree and therefore in the set of tracked nodes. Hence a change will occur for Y with a child structure update flag. This change will be detected and recognized as relevant.

### 3.4.3 Detecting Relevant Revisions

Any revision in the revision graph which performs a change on one of the tracked nodes is relevant for the history because it modifies some of the tracked content. This is clearly true in a linear history. However if a revision has several parents in the history graph additional care is required. Figure 3.13 shows an example of such a scenario. It is safe to say that a revision is relevant if all of its direct parents perform relevant changes. This is not the case in Figure 3.13 and therefore Merge is not considered relevant. An alternative formulation of this statement is: a revision is irrelevant as soon as one of its direct parents does not perform relevant changes. This is used in Algorithm 3.5 to detect relevant revisions.

### 3.5 Merge

In a system that follows an optimistic versioning approach a merge mechanism is required. This section presents the different merge cases which can occur when using a Git back-end. It is shown that already a simple merge approach produces valid abstract syntax trees but might result in unexpected outcomes. The remaining section presents an approach which prevents these unexpected results by enhancing the concept of conflicts.
A common base version is modified into two different versions and merged again. It is assumed that the tracked subtree is only modified in Common Base Version and Version A. Of special interest is the revision Merge because it has two parents. The tracked subtree in Merge is identical to the one in Version A but differs from the one in Version B. Even though some of the changes from Version B to Merge are relevant Merge is not a relevant version in the history because of Version A. There are no relevant changes from Version A to Merge hence Merge does not contain new information regarding the subtree of interest.

### 3.5.1 Merge Scenarios

Using Git as back-end it makes sense to follow the same categorization as Git concerning merge cases\(^2\). This section presents the four different merge cases used in the categorization. There are two branches in the examples presented in the following cases. In all scenarios the branch fix is merged into the branch master.

**Already Up-To-Date**

In the already up-to-date case the merge does not need to adapt anything. This scenario is depicted in Figure 3.14.

**Fast-Forward**

In the Fast-Forward scenario the merge can be perform by simply adapting the reference of the current branch. This situation can be seen in Figure 3.15.

**Fast-Forward with Additional Commit**

The setup in this case is identical to the previous fast-forward case but in this case instead of an adaptation of the branch reference an additional commit is desired. This merge scenario is depicted in Figure 3.16.

\(^2\)http://git-scm.com/docs/git-merge
The function `DetectRelevant` marks the relevant revisions on a revision graph. The argument `current` denotes the current revision item in the revision graph. A set `visited` is used to mark the already visited revisions in the graph. A second set `tracked` is used to store the nodes of the tracked subtree. The last parameter `subtreeRoot` stores the root node of the tracked subtree. The relevant revisions in the revision graph are detected by walking backwards through the entire graph and computing the changes between each consecutive pair of revisions by a diff. If all of the parents have performed a change to the tracked set of nodes then the revision is considered relevant.

Algorithm 3.5: Detect Relevant Revisions

In this merge scenario the branch `fix` is already an ancestor of the branch `master` because both references point to the commit `01a27f`. Hence the current branch `master` is already up-to-date and no further actions are required to perform a merge.
In this merge scenario the branch \textit{master} is merged with the branch \textit{fix}. Since the commit \texttt{01a27f} referenced by \textit{fix} is upstream in the commit history it is possible to perform the merge by only adapting the reference \textit{master}. The result of the merge in a fast-forward setting is showed as dashed \textit{master} reference.

The branch \textit{fix} is merged into the branch \textit{master}. Even though the setup would only require to reset the branch reference \textit{master} it is in some cases desired to use an additional commit to stress the occurrence of a merge in the commit history. Hence as merge result a new commit \texttt{fbb77a} is created which carries the same content as \texttt{01a27f} and the reference \textit{master} is modified. In the figure the merge result is depicted with dashed lines.

\textbf{True Merge}

The true merge case is usually referenced simply as merge. This is the case in which the two branches are not trivially connected to each other and the merge is not just a simple modification of references. Hence this situation requires additional conflict detection and resolution between the two branches to achieve an automatic combination of the two versions as far as possible. Figure 3.17 illustrates the setting.

\section*{3.5.2 General Merge Setting and Idea}

The following sections only consider the true merge scenario since in all other merge scenarios the source tree is entirely given by one of the two versions. Hence no additional modification of the source tree is required. Figure 3.18 depicts the merge setting discussed in the remaining part of the merge section.

The approach realized in Envision’s version control system is based on a sequence of steps. One of the main goals during these steps is to keep a valid abstract syntax tree.
Neither of the two branches master and fix is upstream of the other one. Hence an additional merge commit 01a27f is required to merge fix into merge. The dashed lines depict the situation after the completion of the merge.

There are two revisions Version A and Version B which depict a true merge scenario. In addition a third revision Common Base is present which is the most recent common ancestor of Version A and Version B. The aim is to create an abstract syntax tree which is the merge of Version A and Version B.

1. **Conflict Detection:** In this step conflicts are detected based on the changes which were performed from the Common Base towards Version A and Version B. In addition it is decided if a conflict can be automatically resolved.

2. **Automatic Merge:** The conflict detection is followed by the automatic merge where all changes which do not unresolvably conflict are combined to an abstract syntax tree.

3. **Manual Merge:** This step allows the user to manually adapt the merge. Remaining conflicts are resolved in this stage.

The following section describe these three steps in more detail.

### 3.5.3 Conflict Detection

This section first describes a simple conflict detection mechanism which is able to detect concurrent change operations which would result in an invalid abstract syntax tree. The
simple mechanism allows all concurrent changes which produce a valid abstract syntax tree. However some of the valid concurrent changes result in undesired effects. To remove these undesired effects an additional measure is presented which extends the idea of the simple conflict detection.

The simple way to detect conflicts in an abstract syntax tree is to avoid concurrent changes to the exact same node. This means two concurrent changes can not affect the same node directly and identical labels under a node are also prohibited due to concurrent child structure updates on the node. The following paragraph shows that all syntactical conflicts can be detected on a node level.

Assuming that there are two concurrent changes which need to be classified as conflicting or non-conflicting fixing one of the concurrent change type results in the following case distinction:

- **Insert**: Since the ID of the inserted node is unique it is impossible to get a conflict directly on the new node. Nevertheless it is possible that the inserted node holds the same label in the parent as a concurrently modified node. This conflict is detected on the parent node since every label change (reordering) triggers a child structure update flag in the parent. Operations which affect the set of labels in the parent are: Insert, Delete, Move, Reorder. Therefore the parent receives two concurrent changes. If the concurrent change does not affect the label then no conflict regarding the set of labels in the parent is possible since such an operation would be invalid without the existence of a concurrent change.

- **Delete**: To get a conflict with a delete it is required that exactly the same node is modified by a concurrent change. This can be detected directly with the unique ID of the deleted node.

- **Move**: A move operation can be seen as an insert since the moved node is inserted into the child structure of the new parent. Because the moved node still exists in the abstract syntax tree the updates performed directly on the moved node are not conflicting with the move of the node. Conflicts regarding the label are detected similarly as in the insert case.

- **Reorder**: A reorder operation can also produce a conflict where two concurrent changes try to get the same label in the parent’s child structure. As in previous cases the child structure update flag on the parent does result in a node-based conflict for the parent node.

- **Other**: All other changes (type, value and child structure) do not modify the label of a node. Therefore these changes are only resulting in a problem if the exact same node is concurrently modified. This concurrent modification of the same node will be caught as conflict because it is performed on the same unique ID.

This short case distinction shows that all conflicts which can occur on a node basis are detected by this simple approach. Hence it is always possible with this approach to merge two version to a valid abstract syntax tree. But as the example in Figure 3.19 shows this approach might be too fine-grained. Some combination of changes may result in undesired results and therefore introduce severe and hard to track bugs.
The example shows a LoopStatement from Envision’s source tree with two concurrent updates. Update 1 changes the initialization step and Update 2 adapts the loop condition. With the simple conflict notion these two updates do not conflict and the merge of both changes results in a syntactically valid LoopStatement. There are two reasons why this setting does not have to result in a conflict. First initStep and condition are subtrees and not a single node. Changes in these subtrees do only result in a child structure update flag in the direct parent hence the LoopStatement is only affected if the root nodes of these subtrees are modified. Second not all change operations lead to a child structure update in the parent node.

The problem seen in Figure 3.19 is related to the issue of inserting two different nodes at the same label in a parent node. As previously mentioned this does not result in a conflict on the nodes itself but a conflict in the parent node. In Figure 3.19 LoopStatement is not notified that concurrent changes happen in its substructures because the changes happen further down in the subtrees. This issue can be solved by propagating change notifications further up the tree similar to the child structure update flag but with a larger range. This idea is further clarified in the following sections.

**Conflict Units**

Propagating change notifications helps detecting concurrent changes to closely related nodes in the abstract syntax tree. In general any two concurrent changes in an abstract syntax tree can result in a conflict regarding the semantics of the program. Therefore in a comprehensive approach the propagation of a change notification goes all the way up and stops at the root node of the tree. However detecting semantic conflicts in general based on the information in an abstract syntax tree is impossible, even on a small scale. It is possible to detect simpler kinds of potentially conflicting concurrent changes in local settings. Since the local approach only focuses on a limited neighborhood of a changed node it is not required to propagate the change notifications all the way up to the root node of the tree. Following this approach a conflict can only occur between two concurrent changes in a limited neighborhood of nodes. Such a set of neighboring nodes is called a conflict unit.

A conflict unit is defined by a node in an abstract syntax tree which does not propagate the change notifications received from its children further upwards in the tree. The set of nodes which pass their change notifications to the same node are seen as a conflict
unit. By stopping the propagation of change notifications the top node of the conflict unit gathers all potentially conflicting concurrent changes. In addition it is important to define the conflict units on the Common Base version. Nodes which would define a conflict unit in Version A but do not yet define a conflict unit in Common Base must not be considered as conflict unit definition. The same holds for nodes in Version B. The reason for this is that nodes which are only a conflict unit in Version A do stop change notifications at a node which either lets them propagate or does not even exist in Version B. Change propagation stopped at such a node are not able to participate in a conflict with any change in Version B. Therefore the conflict units are defined on the Common Base version. However note that it is still possible that Version A deletes a conflict unit definition. This is fine since such a deletion is propagated to other conflict units due to the child structure update flag in the parent node.

Applying this idea to the previous example from Figure 3.19 it is reasonable to define the node LoopStatement as a conflict unit node. Since the body of a loop can contain lots of statements it is not desired that changes inside the body conflict with the definition of the loop. Therefore body is a conflict unit node itself. Figure 3.20 shows the result and the effect on concurrent change operations.

Figure 3.20: LoopStatement with Conflict Units

Update 1 is performed in the subtree under initStep and Update 2 in the subtree under condition. Both changes issue a change notifications which is propagated to LoopStatement. Since both changes notifications are stopped at the same node the two concurrent changes happen on the same conflict unit. Hence Update 1 and Update 2 are seen as conflicting changes.

The general idea is to allow only one version to modify the nodes in a conflict unit and thereby preventing conflicting changes. However this idea needs additional care as the next section shows.

Conflict Regions

The overall goal of the merge mechanism is to present the best possible automatically merged abstract syntax tree which is still valid. Additional manual user interaction might still be required to improve the present result. For a conflict unit there exist two possible solutions which are directly valid. By either allowing only the changes from Version A or only the ones from Version B the unit is guaranteed to be syntactically valid. The user should be able to switch between both options manually if the conflict unit contains a
conflict. One problem which shows up when selecting only one version in a conflict unit and therefore rejecting the other one is that the changes performed in an abstract syntax tree are not necessarily independent of each other. Hence the selection inside a conflict unit might also affect validity of operations in other conflict units. An example of such a case can be seen in Figure 3.21.

This example of an abstract syntax tree shows two IfStatement nodes at two independent positions in the tree. One version performs an update on the condition of the IfStatement on the left side of the tree. The other version deletes the old condition in the left IfStatement and replaces it with the one from the right IfStatement. The remaining nodes of the right IfStatements are removed from the abstract syntax tree. The Update and the Delete on the left condition are obviously conflicting operations. Hence only one of the two operations should be performed since both perform modifications to the same conflict unit. But by selecting the Update operation the Delete of the condition and the Move of the right condition are rejected. Under these circumstances the right condition stays in the same position which in turn requires the existence of the right IfStatement. Hence selecting the Update operation turns the Delete operations in the right if statement invalid.

This means in effect rejecting a change operation requires the rejection of other change operations. To achieve this it is necessary to look at the dependencies of the change types. One solution is to keep conflict units as a detection unit for conflicting change operations. In a next step it is required to discover all conflict units which are affected by the conflicting changes. The set of all affected conflict units build a conflict region. Inside a conflict region only the modifications from one version are allowed. The solution in Figure 3.21 is to connect both conflict units of both IfStatement to a conflict region.
The dependencies between operations are based on the requirements of single change operations. Relevant requirements for the success of the application or rejection of a change type are listed below. Note that these requirements are meant for other changes from the same version and do not affect changes from the other version.

**Insert:** To apply an insert the existence of the parent to which the node is attached to is required. Rejecting an insert does not lead to direct requirements.

**Delete:** Applying a delete operation does not place a direct requirement on other changes. The rejection of a delete requires the parent node to be existent.

**Move:** Applying a move requires the existence of the target to which the moved node is attached. Rejecting the operation requires the existence of the original parent node.

**Reorder:** The application and rejection of a reordering places requirement on the labels. Two changes can’t use the same label under the same parent.

It is possible to detect depending changes by checking these requirements when rejecting a change operation. Following these dependencies allows the collection of dependent conflict units which build a conflict region. Note that since move operations on an abstract syntax tree can occur between regionally independent conflict units it is possible that the resulting conflict region is a discontinuous set of nodes in the tree.

### 3.5.4 Automatic Merge

The goal of the automatic merge is to create a valid abstract syntax tree which merges all non-conflicting changes correctly. In addition the resulting tree should build a good foundation for a manual merge of the operations which resulted in conflicts. Figure 3.22 illustrates the automatic merge setting and sketches a solution.

The application of the non-conflicting changes requires additional care. Some of the changes might be dependent on other changes. For example an insert cannot be performed before the parent node exists which might not yet be inserted into the abstract syntax tree. Hence the order in which the changes are applied matters. Establishing this order requires a dependency analysis of all non-conflicting changes. A simpler approach tries to apply a change and reclines the change if it is currently not applicable. The change application mechanism can be seen in Algorithm 3.6.

Extra care is required for concurrent changes on nodes of a list type. The previously mentioned approach is applicable to list types but performs suboptimal since it is too restrictive on these node types. Lots of the conflicts which appear because of list type nodes can be resolved in a suitable way without additional user interaction. This can be done by the same approach text-based version control system. By the detection of a largest common subsequence it is possible to decide which operations are really conflicting and which ones can be automatically resolved.

### 3.5.5 Manual Merge

The automatic merge already provides a valid abstract syntax tree. However the solution produced by the automatic merge is suboptimal since it does ignore all changes of Version
Version A is selected as starting point for the merge. The idea is to merge the changes from Version B into Version A which are not participating in a conflict. These changes are the ones which do not affect a node which is part of a conflict region. The resulting automatic merge is a valid abstract syntax tree which contains all changes which can be merged without additional user support. This is the equivalent of selecting in all conflict regions the Version A option.

B which issue a conflict. Even though the user might need to manually modify the merge result the provided starting point by the version control system should be as good as possible. Therefore additional functionality is required to provide a good merge result and simplify the manual modifications. The previously described approach of using conflict regions that only allow the changes of one version is quite restrictive and forces the user to manually repeat all desired modifications from the other version in the conflict region. To provide a good starting point the system allows the user to swap between possible solutions for each conflict region. The default solution contains only the changes from Version A. By user interaction the region can be changed to only contain changes from Version B. This can be seen in Figure 3.23.

Since a conflict region contains all dependencies regarding the changes the interconnection points between the conflict region and the residual tree are uniform and stable under the swap operation. Therefore it is possible to plug in both versions of the conflict region solution by accordingly connecting the solution to the interconnection points in tree. The mechanism of swapping conflict regions improves the starting solution for the user. But manual merge edits by the user are still required to combine both versions in a conflict region. Swapping just helps the user to decide with which version of the conflict region to start.
function APPLYCHANGESToTREE(tree, changes)
  while ¬changes.empty do
    change ← changes.takeFirst
    success ← false
    if change.type = ADD then
      success ← APPLYADD(tree, change, changes)
    else if change.type = DELETE then
      success ← APPLYDELETE(tree, change, changes)
    else if change.type = MOVE then
      success ← APPLYMOVE(tree, change, changes)
    else
      success ← APPLYSTATIONARY(tree, change, changes)
    if success then
      changes.remove(change)
    else
      changes.append(change)

The function APPLYCHANGESToTREE gets an abstract syntax tree and a list of changes. Following a trial-and-error approach the function tries to apply the first change in the list. If the attempt is successful the change is removed from the list of pending changes. The algorithm continues until all changes are applied. Termination of the algorithm is guaranteed since there are no circular dependencies regarding the applicability of changes.

Algorithm 3.6: Apply Changes to an Abstract Syntax Tree

3.6 Renaming

A change operation which is semantics-preserving to the system is called refactoring. A renaming changes the name of an entity which does not alter the semantics of the system. This is by far the most common kind of refactoring. This section describes an approach to detect renamings in Envision’s source tree and a way to treat references to the changed names. The presented approach is closely coupled to Envision’s source tree format.

3.6.1 Detection

Envision’s source tree format uses a dedicated node type for the names of entities. Nodes which carry the type NameText hold a string value which is the name of an entity in the source tree. A Diff will categorize this change as a value update on a node with type NameText. An example of such a setting can be seen in Figure 3.24. It is safe to say that in the context of Envision every changed name in a program will result in a value update in the diff on a node with type NameText. By checking the results of a diff for this type of change it is possible to reliably detect renamings.

3.6.2 References

The renaming detection focus on the definition of the name of an entity. However there are other occurrences of the name in the source code where it is used as a reference to
Node C is the root node of a conflict region. The tree contains the default solution of the conflict region with changes from Version A. By a swap operation this can be changed to the solution containing the changes from Version B.

The two lines feature an excerpt of a Git diff between two versions of an Envision source trees. By using the previously present comparison approach a change description can be computed. The diff lines represent a value update on a node of type NameText in the source tree. The value changed from welcome to isHello. On a high-level view this node change represents the renaming of a class field.
an already defined entity. References in Envision’s source tree format are of a special
node type Reference or one of its subtypes. Hence it is easy to recognize a reference.
Reference nodes do store their reference target as value. The stored string value is a
combination of the unique ID of the target node and its name. Therefore the reference
can be easily be matched to a target node. Figure 3.25 shows an example of a reference
in Envision’s source tree format.

```
0 Field {f4d3b824[...]9b}
    name NameText {fab00131[...]e3}. S_isValid

[...]

ref OORefERENCE {24b1fe72[...]75}. S_{f4d3b824[...]9b}.is_valid
```

Figure 3.25: References inside Envision’s Source Tree

The first two lines show the declaration of a field with the its name isValid. The
last line shows a reference at an independent location in the source tree. A
node of type OORefERENCE which is a subtype of Reference. The node carries
as value a string which is a combination of the unique ID {f4d3b824[...]9b}
and the name isValid of the reference target.

3.6.3 Renaming Support in Core Algorithms

The previously described core algorithms can be enhanced with the ability to detect
renamings. This section briefly mentions some ideas how each core algorithm can be
improved.

**Diff**

The comparison approach presented in section 3.3 remains exactly the same. However it is
possible to add a post-processing step which enhances the change categorization provided
by the diff. A renaming operation appears as several value updates on nodes in the source
tree. There is one value update to the node which declares the name of the renamed
entity and a value update for each reference node to the renamed entity. The value update
on the references appears due to the change of the second part in the reference value.
By introducing high-level operations it is possible to group low-level changes together
depending on the high-level effect they achieve. The previously mentioned node updates
can be grouped together under a rename operation. An advantage of high-level operations
is that they group related information together and are therefore simpler to understand.

**History**

For the purpose of histories a renaming is still seen as a essential change operation.
Therefore the approach presented to compute relevant revisions in section 3.4 is used
without the previously mentioned post-processing of the diff. Section 7.2 mentions
briefly an interesting idea to further distinguish the relevant changes between inessential refactorings and essential operations.

**Merge**

The ability to detect a renaming and find the corresponding references is most essential for the merge mechanism. Since Envision works based on unique IDs it is always clear on which entity a change needs to be applied regardless of its name. Hence it is possible to perform the renamings on all nodes where they are required. Figure 3.26 shows a general merge setting.

![Diagram of Merge Process](image)

**Figure 3.26: Renaming Support in Merge**

General three-way merge setting with Version A and Version B. By computing the diffs between Version A and Common Base as well as Version B and Common Base it is possible to detect the renamings performed in Version A and Version B. Note if an entity is renamed in Version A and a new reference to this entity is introduced in Version B then in the merge the newly introduced reference is not adapted to the new name of the entity. The reference in this case carries the correct unique ID but the wrong name string. This need to be fixed to support renaming properly.

As explained in the description of Figure 3.26 it is possible that newly introduced reference carry the wrong target name after a merge. By detecting the renamings in both versions it is possible to replay them on the newly introduced references which still carry the old name string and thereby turning the partially renamed merge into a fully renamed merge.
Chapter 4

Visualizations and User Interactions

This chapter explains the main visualization and interaction ideas designed for Envision’s version control system. The ideas are based on the visualization and interaction goals of Envision. The most relevant goals are shortly explained in section 2.3. Each of the next three sections focus on a version control core functionality. The first of them explains the visualization of a diff and its change types. This section also features information on which types of revision specification is supported during interactions. The second one explains the interactions and visualization which are used for the history. The third section presents the interactions required to perform a merge and the consequent visualizations.

4.1 Diff

A Diff between two versions might contain a lot of changes especially when the entire project is affected and displayed. This might be desired by a user to explore the evolution of the entire project. However as soon as the user is only interested in a particular construct the entire diff contains too much unimportant information which distracts the user. It is therefore essential that the interactions and visualizations allow an efficient and easy way for the user to filter out all unimportant changes. The result is a diff which can be limited to a substructure chosen by the user. Namely performing a diff command on a class will display all changes which affect the selected class. Performing the same command on a method will only display the changes affecting the method.

User Interaction

Diffs are performed between the currently version loaded and another version specified by the user. The resulting command can be performed on the entire project but also on several subunits like classes and methods. The syntax of the command is `diff <revision>`. The revision parameter is a string and uses extended SHA-1 syntax. Some of the valid ways to specify a revision are:

- `<sha1>`: The full SHA-1 object name or a leading substring which is unique within the used repository.

- `<refname>`: The name of a symbolic reference. This includes tags, branch names and remotes.
• `<refname>@{<date>}`: A symbolic reference followed by an @ is used to specify the value of the reference at a earlier point in time. It is possible to use an exact date (\{2014-06-18 13:00:00\}) or a relative timespan (\{1 hour ago\}).

• `<rev>^\langle n\rangle`: A caret can be used to specify the first parent of the revision `<rev>`. If a revision has multiple parents an additional number can be used to specify the exact parent. `<rev>^\langle n\rangle` denotes the n-th parent of revision `<rev>`.

• `<rev>~\langle n\rangle`: A tilde is used to specify the n-th generation ancestor of the names revision. It is possible to combine a caret with a tilde to select the n-th generation ancestor of the m-th parent.

More details on supported revision formats can be found in the Git documentation of [git-rev-parse](http://git-scm.com/docs/git-rev-parse). The SHA-1 values, tags and reference names support auto-completion. However the suggestions made by the auto-completion mechanism are not enforced. Hence the user can use the regular expression like way to specify a revision.

The contents of the revision specified by the revision parameter are read-only. Nevertheless it is possible to perform non-modifying commands like find on the read-only content.

Visualisation

![Visualization](image)

Figure 4.1: Comparison Mock-up

The very general visualization mock-up in Figure 4.1 shows how a comparison could look like. The left side shows a read-only version of the revision specified by the user. The shown section only contains the structures on which the user performed the diff command. On the right side is the current version. To simplify navigation both versions carry a visual element which mentions the version (HEAD and 1cf4e0). Since the diff command is
performed on an entity selected by the user only the changes which affect the selected entity are visualized. The changes are visualized in four different categories:

- **Insert:** Newly added constructs are marked in the current version (HEAD). A inserted entity is highlighted with a green background.

- **Delete:** A removed construct is displayed in the read-only revision (1cf4e0). The removed entity is highlighted by a red background.

- **Move:** An entity which changed its location in the structure is shown in both versions. The relocated entity is highlighted with a yellow background and the highlights in both version are connected by a yellow arrow.

- **Update:** An updated entity is marked in the previously loaded version (HEAD) and highlighted with a blue background.

Changes are visualized in a manner that only the largest structure affected by the same change is visualized. For example for a newly added method a highlight is created for the entire method but not for every single statement in that method unless the statement is not inserted but moved. In that case a statement inside the green highlight would get an additional yellow highlight.

### 4.2 History

To understand how a certain state of an entity came into existence it is important to have a flexible mechanism to visit a project’s history. The history contains previous revisions of the entity in a structured way. Depending on the structure of interest many of the revisions in the history log may be irrelevant since they do not contain any changes to the desired structure. Therefore the relevant entries in the history are emphasized in a way that the user quickly sees where the interesting changes occurred.

#### User Interaction

The history is computed from the currently loaded version backwards. An additional revision is specified to define a range of history. Hence the history command has the form `history <rev>`. After calling the command on an entity in the program structure the user can select revisions in the provided history range. A read-only preview displays the selected revision of the entity on which the command was executed. Similar as in the Diff the user can run non-modifying commands on the read-only content.

#### Visualization

Figure 4.2 shows a mock-up of the history visualization. On the right side is the version which was previously loaded by the user. The left side shows the revision preview on top and the history list (Log) underneath. The entity on which the history command was executed is highlighted in the previously loaded version with a yellow background and the preview only shows other versions of exactly this entity. As the user selects a version in the history list (Log) the selected version is displayed in the preview and marked gray in
the history list. Revisions which changed the entity on which the history command was performed are marked with a yellow background in the history list.

### 4.3 Merge

The Merge requires additional case distinctions since the visualizations and user interactions depend on the merge scenario. Section 3.5.1 shows an overview of the merge scenarios in Envision and under which circumstances they occur.

#### User Interaction

A merge is always performed on the currently loaded version and affects the entire application tree. To merge another revision \texttt{rev} into the current version the command `merge [noFF] \texttt{<rev>}` is used. The option \texttt{noFF} can be used to prohibit fast-forward merging. If no such option is provided a fast-forward merge is applied whenever possible. Figure 4.3 shows the interaction model for all different merge scenarios. In case the current version is already up-to-date or a fast-forward merge is applicable there will be no additional commit. Therefore the system does not enter a merge state. If an additional commit is required the system enters a merge state which is highlighted with a red color in Figure 4.3. To exit the merge state the user has two options. Either performing a \texttt{commit} or \texttt{abort} the merge. To perform a commit all conflicts have to be resolved. Each conflict can be selected in a conflicts list. For the selected conflict the user can \texttt{swap} between both versions and has to decide which one to keep. Of course the chosen version...
can be modified afterwards. To choose a version the user calls `resolve` on the selected conflict. The currently active version of the conflict region will then be selected as merge result. As soon as the user resolved all conflicts it is safe to commit. In addition to the conflict resolution the user can perform manual edits. The merge is completed by using the `commit` command.

![Merge Interaction Model](image)

**Figure 4.3: Merge Interaction Model**

**Visualization**

Since the already up-to-date and fast-forward merge scenario do not require additional user interaction no additional visualization is performed. Figure 4.4 shows a mock-up
of the merge state screen. The right side of the screen shows the result of the merge with all conflicts highlighted with a red background. A large red box depicts that the system is currently in a merge state. The currently active conflict is marked with a red dashed border. This is also the conflict which can be swapped. The other possible merge solution is depicted on the left side and connected by a red arrow. The various conflict versions carry their revision string in the lower right corner of the highlighting box. A list of conflicts allows quick access to all conflicts. The currently active conflict is marked with a red highlight and already resolved conflicts are marked with a green highlight.
Chapter 5
System Design

This chapter reveals how the algorithmic ideas of chapter 3 and the visualization and interaction concepts of chapter 4 are realized from a more technical point of view. Since the system is build on a Git back-end the first section contains all essential information on Git and its internals as well as the library libgit2 which is used to connect to the Git repository. Section 5.2 explains the most important design choices made during the implementation of Envision’s version control system.

5.1 Git Back-End

As already motivated in section 2.2 the version control system works with a Git back-end. This section describes which functionality of Git is used and how the connections to Git are realized.

Git is a well established state-of-the-art distributed version control system\(^1\). Because Git is used as back-end it is important to understand some of the basic Git internals.

5.1.1 Basic Git Internals

This information on Git internals is based on Git’s reference\(^2\) and Scott Chacon’s Pro Git\(^3\). As a Git repository is created in or cloned into the local file system the directory .git is created. For the purpose of Envision the following entries in this directory are important.

- **HEAD**: The HEAD file contains a symbolic reference to the current branch. If the current branch is master then the HEAD file contains refs/heads/master.
- **index**: The index file is used to store the staging area information.
- **objects**: The objects directory contains all Git objects.
- **refs**: The refs directory stores references into objects stored in the objects directory. All branches are stored here.

---

\(^1\)http://git-scm.com/
\(^2\)http://git-scm.com/docs
\(^3\)http://git-scm.com/book
Git Objects

Git objects are used to store data into a repository. The most important objects are blobs, trees and commits. As previously mentioned all of them are stored in the objects directory. Git is designed in such a way that the content of the data stored determines the key under which it is stored. To achieve this Git uses the SHA-1 value of the data content as key. The first 2 characters in the SHA-1 value are used to name a directory and the remaining 38 characters are used as filename. For example the commit 82525f...9804 is stored as commit object under .git/objects/82/525f...9804.

The simplest Git object is the blob. This object is used to store the content of data into the repository. A SHA-1 value is calculated based on the content and the blob is stored as previously mentioned into the objects directory. A blob does not contain anything else than the content. Hence another object is required to store the filename and to have a convenient way to access the stored content.

A tree object is similar to a directory in a file system. This objects can hold several blobs and has the filenames stored for them. Git uses blob and tree objects to describe the files and directories which are stored in a repository. Multiple versions of a file system are stored by using multiple trees.

Commit objects are used as a convenient way to store versions. Every commit has a reference to a tree which stores the version represented by the commit. In addition the commit object holds essential information on the commit like a message, a timestamp and information on the committer and author.

Figure 5.1 illustrates how the three different types of Git objects interact with each other.

Figure 5.1: Git Objects

5.1.2 Libgit2

To access Git’s core functionality the external library libgit2 is used. Libgit2 is a portable, pure C implementation of the Git core methods provided as a linkable library.

---

5https://libgit2.github.com/
5.2 Core Classes

This section describes some of the important design choices made during the development of Envision’s version control system regarding class design. It starts with a high-level system overview followed by sections of which focus on one core mechanism. Figure 5.2 shows a UML diagram of the entire version control system realized in Envision.

![UML Diagram of Envision’s Version Control System](image)

**Figure 5.2: UML of Envision’s Version Control System**

The UML diagram contains all relevant classes and their relations in the context of Envision’s version control system. Classes with a green background are not directly a part of the version control system but are used to represent Envision’s source tree at a low level. These classes are also used by the file parsing functions of Envision. The classes with a blue background take care of the user interaction and the visualization of one specific mechanism in the system. The remaining white classes build the version control system manager in Envision.

**GitRepository**

The class `GitRepository` is used as an abstraction layer for the Git back-end hence all direct connections to Git via libgit2 occur in this class. Therefore it features a lot of
functionality and is rather big. For the sake of a clearer understanding GitRepository is presented in different sections with a focus on one particular mechanism in Envision's version control system.

5.2.1 Focus on Diff
ChangeDescription
The class ChangeDescription builds the foundation of the diff mechanism. It takes care of the identification and storage of the change category between two source tree nodes as described in section 3.3.1. This functionality is separated from the Diff because describing the differences between two GenericNode objects is possible for any two GenericNode objects. In the context of diff a ChangeDescription is created from two GenericNode objects which represent two versions of the same node (matching ID) in the source tree. The constructor of ChangeDescription directly computes the change type and almost all update flags. The flag for child structure updates is not set by the constructor due to the efficiency issue presented in section 3.3.4 and therefore has to be set manually by inspecting change types. In the setting of the diff mechanism this is done by the Diff class.

Diff
The Diff class captures an entire comparison between two versions of Envision's source tree. A Diff is constructed by passing a list of GenericNode objects for each version. In the general diff setting in Envision these lists are established by the GitRepository class. How the lists are constructed in detail can be seen in section 5.2.1. In general the way the lists are constructed is irrelevant for the Diff class. The lists are simply containers for the GenericNode objects. The Diff class computes a set of ChangeDescription objects from the input lists. By the encapsulation of this functionality it is possible to modify the way the GenericNode objects are extracted from the repository without the need to adapt the entire diff mechanism.

The main tasks of Diff are the filtering of persistent units and the finding of matching pairs of GenericNode objects. As previously mentioned a source tree in Envision is stored on disk in several persistent units. A GenericTree object holds one source tree which may have several GenericPersistentUnit objects. The GenericPersistentUnit objects store the GenericNode objects of the source tree. It is required to filter the persistent units since the nodes on the border of two persistent units appear in both units. The function nodeMatching performs the node matching and implements the algorithm explained in section 3.3.2. An additional GenericNodeHash is required to store newly created nodes which are not contained in the lists of GenericNode objects obtained at construction. Therefore a reference to the GitRepository is required to query these additionally required nodes.

Figure 5.3 also shows the connection of the Diff class to other classes of Envision. The Diff class holds two references to GenericTree objects but does not own them since these objects are shared with other entities. These GenericTree objects own their GenericPersistentUnit objects which own their GenericNode objects. The
Diff
-changeDescriptions: QHash<Model::NodeIdType, ChangeDescription*>
-treeA: std::shared_ptr<GenericTree>
-treeB: ...

Diff
-changeDescriptions: QHash<Model::NodeIdType, ChangeDescription*>
-treeA: std::shared_ptr<GenericTree>
-treeB: ...

Figure 5.3: UML of Diff Mechanism
ChangeDescription objects which are established during the construction of a Diff object are owned by the Diff object itself. There is an additional connection to a GitRepository.

GitRepository

The most important function regarding the connections to Diff is the function diff which is the interface to get the diff between two revisions in the repository. It is the comparison interface of Envision’s version control system. This function creates and returns a Diff object from the Git back-end. The information required to do this is extracted from several possible sources depending on the requested comparison. The three possible sources are the work directory of the project, the index of the Git repository and a commit in the Git repository. Depending on the source a different access approach is required to get the source tree. The work directory requires a simple file access while a commit requires to read the content out of a blob stored in the Git repository. The function diff in GitRepository runs a text-based diff on the specified revisions and fills the lists with nodes which appear as modified in the text-based diff. Section 3.3.3 explains the algorithmic details how the nodes are extracted from the text-based diff. The diff data structure provided by libgit2 uses callback functions to iterate over the text-based diff output. A line-based callback is used to parse the line to a GenericNode and insert it to the node lists. The lists are directly passed to the constructor of the Diff object.

5.2.2 Focus on History

CommitGraph

The CommitGraph class is used to store a part of the commit history of a Git repository. This class is a container and manager for the CommitGraphItem struct. The interconnection information in CommitGraphItem is stored in the children and parents lists. The main motivation to use a separate class CommitGraph from History and GitRepository is to store commit graphs independent of History and to abstract from the low level Git notion of commit connections. New connections can be established by using the add function in CommitGraph. By using find it is possible to obtain the corresponding CommitGraphItem to a commit string from CommitGraph.

History

The History class does not need a direct connection to the Git repository since the CommitGraph contains the required commit history. It is the task of the History class to classify the relevance of the commits in a CommitGraph object. This is achieved by following the approach presented in section 3.4.3. In addition to the CommitGraph the History class requires access to the content of commits to build a new set of tracked nodes. For this the GitRepository is used. Instead of getting the entire commit of the source tree the History object knows from the Diff object used to classify the relevance which files contain the tracked subtree. Only those files are loaded from the GitRepository by using getCommitFile. A History object is constructed from a CommitGraph object and the ID of the root node of the tracked subtree in the source tree. During construction of the object
Figure 5.4: UML of History Mechanism
the set of relevant commits is established by recursively calling `detectRelevantCommits` on `CommitGraphItem` objects.

**GitRepository**

Since the `History` class works directly on a `CommitGraph` object there is no direct function to query a history in `GitRepository`. However building of a `CommitGraph` is closely coupled to the commit history in the Git repository. Hence the function `commitGraph` in `GitRepository` can be used to construct a `CommitGraph`. The function iterates through the libgit2-based commit history and extract all information required to insert `CommitGraphItem` objects into a `CommitGraph`. This is an implementation of the `REVISIONGRAPH` algorithm presented in section 3.4.1.

### 5.2.3 Focus on Merge

**GitRepository**

It is essential for the proper performance of Envision’s version control system to keep the Git repository in a correct state. This is the task of the `GitRepository` class which takes care of the low-level merge operations on the Git level. The `GitRepository` creates the Git objects described in section 5.1.1 and keeps the Git index up-to-date. Furthermore, this class takes care of the adjustment of Git references if needed. In addition, the `GitRepository` loads the `GenericTree` objects from Git commits. The common merge based used to perform a three-way merge is directly calculated on the Git repository by the use of libgit2 functionality. Due to the clear isolation of libgit2 from other classes this is also part of the `GitRepository`.

It is crucial for the system’s correctness that only one merge can be performed at a time. To achieve that the interactions between `GitRepository` and `Merge` are designed in a way that allows only the existence of one active `Merge` object at a time. The `GitRepository` keeps a reference to the `Merge` object which was created last. As soon as the merge is completed the reference in `GitRepository` expires. A new `Merge` object is created via `merge` in `GitRepository` only if the merge reference has expired. If the reference is still valid then the `Merge` object in the reference is returned. This concept and the design choice that a lot of the previously mentioned functionality required for merging is private since it is closely connected to the internals of Git. These functions in `GitRepository` can turn a Git repository into an invalid state if called in the wrong fashion. Thus the `GitRepository` and the `Merge` class are closely connected and mutual friends.

**Merge**

The `Merge` class is a rather big class featuring all functionality required to perform the merge approach presented in section 3.5. After calling `merge` on `GitRepository` all interaction to complete the merge is performed on the received `Merge` object. The `Merge` class is organized as state-machine since some operations are only valid in certain states. For example it is only possible to commit successfully in the state `ReadyToCommit`. This prevents the commit of an incompletely merged source tree. The transitions can be seen in Figure 5.6.
Figure 5.5: UML of Merge Mechanism
Every constructed Merge object starts in state Initialized. The merge scenario is detected in Classified according to section 3.5.1.

A Merge object has two options to complete the merge. One is by calling commit. The other is by calling abort. If a Merge object is deleted but not yet completed then the merge is automatically aborted and reverted before deletion of the object. The remaining functions implementing the previously presented merge approach are isolated in Merge and independent of other classes.
Chapter 6

Evaluation

This chapter features a brief presentation of the results achieved with Envision’s version control system. To evaluate the system the results are compared to a traditional text-based system. The focus is on the three core mechanisms, diff – history – merge, described throughout this thesis.

6.1 Diff Tool

This sections presents Envision’s version comparison functionality on a small Java example. Afterwards the result is compared with the output of Meld\(^1\). Meld is a text-based visual comparison tool which runs under GNU GPL.

The evaluation uses two versions of a simple code example. The initial version A of a class called Student can be seen in Figure 6.1. Version A has some issues which are fixed in version B. To do that a second class Account is created. In detail the changes which lead from version A to version B of the example are:

- Rename name to familyName
- Add a field called givenName
- Replace the boolean familyNameFirst
- Adapt the constructor of Student
- Adapt the implementation of function familyName
- Move user account related functionality to Account class
- Add account field to Student

The improved version B of Student can be seen in Figure 6.2 and Figure 6.3 shows the newly created class Account.

\(^1\)http://meldmerge.org/
public class Student {

    public Student(String name, boolean familyNameFirst) {
        name_ = name;
        familyNameFirst_ = familyNameFirst;
    }

    public String familyName() {
        String[] names = name_.split(" ");
        if (familyNameFirst_)
            return names[0];
        else
            return names[1];
    }

    public int getStudentID() {
        return studentID_;}

    public boolean validPassword(String password) {
        return accountPassword_.equals(password);
    }

    private String name_;  
    private boolean familyNameFirst_; 
    private int studentID_; 
    private String accountUserName_; 
    private String accountPassword_; 
}

Figure 6.1: Version A of Student

Initial version of Student used for the comparison of the diff functionality. The class stores the name of a student as string and uses a boolean to memorize the order of the names. The function familyName can be used to split the name into the family name and given name. In addition the Student has a user account with a password.
public class Student {

    public Student(String familyName, String givenName) {
        familyName_ = familyName;
        givenName_ = givenName;
    }

    public String familyName() {
        return familyName_;
    }

    public int getStudentID() {
        return studentID_; 
    }

    private String familyName_;
    private String givenName_;
    private int studentID_; 
    private Account account_; 
}

Figure 6.2: Version B of Student

The improved version of Student stores the given name and the family name in separate fields.

public class Account {

    public Account(String userName, String password) {
        userName_ = userName;
        password_ = password;
    }

    public boolean validPassword(String password) {
        return password_.equals(password);
    }

    private String userName_; 
    private String password_; 
}

Figure 6.3: Version B of Account

The newly created class Account isolates the functionality of the user account from Student.
To compare both versions a Git repository was created outside of Envision. Both versions were implemented in Envision and stored to the repository. The stored source trees were in some parts manually adapted since Envision currently does not carefully maintain the unique node IDs under certain changes. For example changing a part of an expression results in an entirely new expression with new IDs. These changes are still captured using Envision’s version control system as delete and insert operations. However the changes cannot be represented as updates or moved code since the unique IDs are not matching. This is a problem of the IDE and not the version control system directly. To fully show the potential of the developed system the unique IDs of the nodes in version B were manually modified to match the corresponding nodes in version A.

After loading version A of the example in Envision a diff is performed using the command diff on the entire project. Figure 6.4 shows the output produced by Envision. Version A is depicted on the left side and version B is depicted on the right side of the figure. Deleted code is marked in version A by using a red highlight. Inserted code is marked in version B by using a green highlight. Code which was moved is highlighted in both versions by using a yellow highlight. Updates are visualized by using a blue highlight in version B.

The resulting visualization seen in Figure 6.4 captures the actually performed changes quite well. There is a clear distinction between renaming the field name_ to familyName_ and the replacement of familyNameFirst_ with givenName_. It is also directly visible that the class Account is new in version B but some of the code, like validPassword, was moved and not newly created.

As comparison Figure 6.5 shows the same diff in Meld. The figure only shows the class Student since Meld performs a file-based comparison the diff for Account is simply a insertion of the entire file content. Meld uses stable chunks in the code to match modified parts. Even though the chunks are matching quite well, as in the example, it is not directly evident what kind of changes occurred.

In addition to an entire diff, Envision features the functionality to perform partial diffs on substructures. Figure 6.6 shows exactly the same setting as previously used but this time the diff command is executed on the class Account. The visualization shows only the relevant changes which affect the selected substructure and ignores all other changes. Hence the only changes which are visualized in Student are the code parts which are moved to Account. Such a partial diff on the class level is also possible in Meld. However Meld is not able to track moved code from other structures and Envision allows even finer substructure like methods.
Figure 6.5: Diff of Student in Meld

```java
public class Student {
    public Student(String name, boolean familyNameFirst) {
        name_ = name;
        familyNameFirst_ = familyNameFirst;
    }

    public String familyName() {
        String[] names = name_.split(" ");
        if (familyNameFirst_)
            return names[0];
        else
            return names[1];
    }

    public int getStudentID() {
        return studentID_;  // Added underscore
    }

    public boolean validPassword(String password) {
        return accountPassword_.equals(password);
    }

    private String name_;      // Added underscore
    private boolean familyNameFirst_; // Added underscore
    private int studentID_;      // Added underscore
    private String accountUsername_; // Added underscore
    private String accountPassword_; // Added underscore
}
```
Figure 6.6: Partial Diff in Envision
6.2 History Tool

To evaluate the history mechanism a third version C is added to the previously introduced versions A and B. In version C an additional function called \texttt{changePassword} is added to \texttt{Account}. Figure 6.7 shows the code of version C.

![Figure 6.7: Version C in Envision](image)

The history tool in Envision does not yet support graphical visualization. But with the use of a textual output it is possible to show the improvements over traditional history features. As comparison the Git function \texttt{log} is used. In the current setting Git’s \texttt{log} can be used in the three different ways seen in Table 6.1

<table>
<thead>
<tr>
<th>(a) Repository</th>
<th>(b) On Student</th>
<th>(c) On Account</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>C</td>
</tr>
</tbody>
</table>

Table 6.1: History Outputs Using Git’s \texttt{log} Command

Column (a) shows the versions which \texttt{log} returns when the entire repository is considered. Column (b) shows the output of \texttt{log} if only the file \texttt{Student} is passed as argument. Column (c) contains the versions when \texttt{log} is only used on the class \texttt{Account}.

The \texttt{log} command doesn’t allow more fine-grained histories since it follows a file-based approach. Using Envision’s \texttt{history} command the same results are achieved. Column (a) is the output of calling \texttt{history} on the entire project. Column (b) and (c) are the

\footnotesize{\texttt{http://git-scm.com/docs/git-log}}
result of calling the command on the class entities. But Envision’s history approach is not restricted to structures. However for the ease of use the command can currently only be called on major structures in the source code. This includes functions of a class. Table 6.2 shows the output of Envision’s history for finer structures.

<table>
<thead>
<tr>
<th>(a) Account constructor</th>
<th>(b) validPassword</th>
<th>(c) changePassword</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.2: Fine-Grained History Outputs Using Envision’s `history` Command

Column (a) shows the relevant versions returned when running `history` on the constructor of `Account`. Column (b) shows the output of `history` performed on `validPassword` in `Account`. Column (c) contains the versions when `history` is used on `changePassword` in the class `Account`.

### 6.3 Merge Tool

In the current version the merge mechanism does not support all functionality presented throughout the thesis. Due to the missing functionality it is not possible to resolve conflicts. The following example presents a simple merge case which has no conflicting changes but requires the mechanism to replay all the detected changes from one version onto the source tree of the other version.

In the example the previously presented version C of the `Account` class is used as common base. The branch `accessLevel` introduces an access level in form of an integer field to the class. Additionally the constructor is modified and the setter in Figure 6.8 is introduced.

```java
public void setAccessLevel(int level) {
    accessLevel_ = level;
}
```

Figure 6.8: Simplified Change applied in branch `accessLevel`

The branch `accessLevel` modified Version C by introducing the field `accessLevel_`. In addition new method `secAccessLevel` is introduced.

A second branch `pwLength` modifies the `Account` to prevent passwords shorter than six characters. This modification can be seen in Figure 6.9.

To perform a merge in Envision one of the two branches needs to be loaded first. This example shows the setting where the branch `accessLevel` was loaded. After that the command `merge pwLength` is executed. The `merge` command always affects the entire project. Figure 6.10 shows the merge solution computed by Envision.
public boolean changePassword(String from, String to) {
    if (to.length() < 6)
        return false;
    if (validPassword(from)) {
        password_ = to;
        return true;
    }
    else
        return false;
}

The branch **pwLength** adds an additional check of the password length to Version C of class **Account**.

Envision’s result when merging the previously mentioned branches **accessLevel** and **merge pwLength**.
Chapter 7

Future Work

This chapter lists work which could only be partially completed due to the lack of time. It also includes ideas for additional work in the context of Envision’s version control system.

7.1 Partially Completed Work

The merge algorithm does currently not support the detection of conflicts since the detection of conflict regions is not implemented. Conflicting conflict units are detected but the additional step of resolving the dependencies between operations is missing.

The user interaction and visualization of the history and merge is not completely implemented. The current implementation are experimental and serve as proof of concept.

The renaming support mentioned in Section 3.6 is not implemented at the current state of the project.

7.2 Further Directions

This section includes some ideas for additional work which were not further investigated during the thesis.

Support for Additional Refactorings

The current design and algorithms feature renaming support to some degree. The idea of using the change descriptions produced by the diff mechanism and establish more high-level change characterizations is not limited to renaming. It would be interesting to which degree high-level operation detection, like the one by Fluri and Gall [FG06], can be incorporated into Envision.

Merging Identical Changes

One of the weak points of the system is that the creation of two identical structures in two separate versions result in two identical structures with different IDs. However this
is not a severe weakness since in a real world environment this is unlikely to happen. Nevertheless this cases can be detected by a heuristic since it only affects the adding of new nodes. Additional care is required to be able to track the full commit history in both branches.

Automatic Conflict Resolution Plug-ins

The merge algorithm currently support automatic conflict resolution for list structures. There are certainly other constructs which allow automatic conflict resolution to some degree. In addition it would be convenient if these automatic resolution approaches can be programmed in a plug-in-like style.

History Limited to Essential Changes

The presented history approach is able to compute a log for a subtree that contains only the relevant commits. Relevant in the current setting means that there where changes performed on the subtree. Motivated by the work of Kawrykow and Robillard [KR11] it could be interesting to provide an option to the history mechanism which ignores inessential changes. One possible definition of inessential changes could be a restriction to simple refactorings.
Chapter 8

Conclusion

This thesis extends Envision – a visual structured code editor for object-oriented languages – with a version control system. The presented version control system focuses on the core tasks which differ from traditional text-based systems in the context of Envision.

An approach is presented which uses a Git back-end to perform basic version control tasks. On top of that a number of algorithms are presented to tackle the problems of version comparison, version history and version merging in the context of Envision’s source tree. An additional visualization and user interaction concept is developed for each of these tasks.

The provided approach is partially implemented in Envision and a short evaluation showed some of the advantages of Envision’s version control system towards the text-based approaches used in Git.
Bibliography


77


# Eigenständigkeitserklärung


Die Dozentinnen und Dozenten können auch für andere bei ihnen verfasste schriftliche Arbeiten eine Eigenständigkeitserklärung verlangen.

Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuer und Betreuerinnen der Arbeit.

**Titel der Arbeit** (in Druckschrift):

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

**Verfasst von** (in Druckschrift):

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

<table>
<thead>
<tr>
<th>Name(n):</th>
<th>Vorname(n):</th>
</tr>
</thead>
<tbody>
<tr>
<td>Otth</td>
<td>Martin</td>
</tr>
</tbody>
</table>

Ich bestätige mit meiner Unterschrift:
- Ich habe keine im Merkblatt „Zitier-Knigge“ beschriebene Form des Plagiats begangen.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu dokumentiert.
- Ich habe keine Daten manipuliert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Plagiate überprüft werden kann.

**Ort, Datum**

Zürich, 1. Oktober 2014

**Unterschrift(en)**

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.