AN ABSTRACT OF THE THESIS OF

Xiangyu Wang for the degree of Master of Science in Computer Science presented on Jan 05, 2017.

Title: Using Change Representations to Analyze Properties of Variational Software

Abstract approved: __________________________________________________________

Martín Erwig

Appropriate representations of variational software simplify the analysis of their properties. This thesis proposes tailored representations of two kinds variational softwares: difference files of merge commits in Git and feature models. For the former, we use the Choice Edit Model, which is based on the choice calculus, to represent changes introduced by merge commits. This approach identifies merge conflicts and automatically detects conflicts that can be ignored because they are semantically irrelevant. An experiment of 50 Git repositories shows that in some cases about 10% of merge conflicts having no semantical effects. For feature models, this thesis provides Choice Dependency Graphs to show relationships of features. Choice Dependency Graphs provide a succinct representation that supports 4 analysis operations for feature models effectively.
©Copyright by Xiangyu Wang
Jan 05, 2017
All Rights Reserved
Using Change Representations to Analyze Properties of Variational Software

by

Xiangyu Wang

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented Jan 05, 2017
Commencement June 2017

APPROVED:

__________________________________________
Major Professor, representing Computer Science

__________________________________________
Director of the School of Electrical Engineering and Computer Science

__________________________________________
Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

__________________________________________
Xiangyu Wang, Author
ACKNOWLEDGEMENTS

First, I would like to acknowledge my major advisor Dr. Erwig. He advises me in my studying of programming language and gives me valuable directions in my research work. He has great patience when I meet difficulties, and always gives me great understanding, especially in the hard time of my life. The most important lesson I learned from him is being dedicated and strict to work. I may not become a researcher in the future, but the professional attitude I learned from him will let me benefit a lifetime.

Second, I would like to thank my parents for always being my side. They give me confidence to do hard work and support me to overcome difficulties. Although they are not with me in the US, I can feel their love every day.

Third, I would like to express my thanks to my husband, Chao, for helping and accompanying me in these 7 years. He is my best friend and lets me be a better human.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>Variational Software</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>Merge Conflicts in Git</td>
<td>2</td>
</tr>
<tr>
<td>1.3</td>
<td>Analysis Operations on Feature Models</td>
<td>3</td>
</tr>
<tr>
<td>1.4</td>
<td>Thesis Outline</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Literature Review</td>
<td>6</td>
</tr>
<tr>
<td>2.1</td>
<td>Feature Models</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>The Choice Calculus</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>The Choice Edit Model</td>
<td>14</td>
</tr>
<tr>
<td>2.4</td>
<td>Version Control Systems</td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td>Edit Representations</td>
<td>21</td>
</tr>
<tr>
<td>3.1</td>
<td>Edit Sequences</td>
<td>21</td>
</tr>
<tr>
<td>3.2</td>
<td>Transforming Edit Sequences to the Choice Edit Model</td>
<td>23</td>
</tr>
<tr>
<td>3.3</td>
<td>Properties of the Transformation</td>
<td>26</td>
</tr>
<tr>
<td>3.4</td>
<td>Choice Dependency Graphs</td>
<td>29</td>
</tr>
<tr>
<td>3.5</td>
<td>Translating Choice Calculus Expressions into Choice Dependency Graphs</td>
<td>31</td>
</tr>
<tr>
<td>4</td>
<td>Analysis of Merge Conflicts</td>
<td>33</td>
</tr>
<tr>
<td>4.1</td>
<td>Identifying Conflicts in Merge Commits of Git</td>
<td>33</td>
</tr>
<tr>
<td>4.2</td>
<td>Identifying Conflicts Without Semantic Relevance</td>
<td>37</td>
</tr>
<tr>
<td>4.3</td>
<td>Experiment</td>
<td>39</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Analyzing Merge Conflicts in Repositories</td>
<td>40</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Data Description</td>
<td>41</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Results</td>
<td>42</td>
</tr>
<tr>
<td>5</td>
<td>Supporting Operations on Feature Models by CDGs</td>
<td>43</td>
</tr>
<tr>
<td>5.1</td>
<td>Translating a Feature Model to a Choice Dependency Graph</td>
<td>43</td>
</tr>
<tr>
<td>5.2</td>
<td>Using Choice Dependency Graphs to Support Analysis Operations for</td>
<td>49</td>
</tr>
<tr>
<td>Feature Models</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Valid Product</td>
<td>49</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Dead Features</td>
<td>50</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Void Feature Model</td>
<td>51</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Redundancy</td>
<td>52</td>
</tr>
<tr>
<td>6</td>
<td>Conclusion</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>54</td>
</tr>
<tr>
<td>Appendices</td>
<td>Inference Rules to translate Edit Sequence to Choice Edit Model</td>
<td>59</td>
</tr>
<tr>
<td>A</td>
<td>Source code of the Coq Program to translate Edit Sequences to Choice Edit Model</td>
<td>62</td>
</tr>
<tr>
<td>C</td>
<td>50 most well-known repositories in Git</td>
<td>76</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A feature model for mobile phone proposed by Benavides et al [10].</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>A simple feature model in which feature MobilePhone has an optional child feature GPS.</td>
<td>13</td>
</tr>
<tr>
<td>2.3</td>
<td>A typical branching and merging workflow in Git [4].</td>
<td>18</td>
</tr>
<tr>
<td>2.4</td>
<td>Merge ordered recipe into master branch.</td>
<td>20</td>
</tr>
<tr>
<td>3.1</td>
<td>A string changes after two edits.</td>
<td>23</td>
</tr>
<tr>
<td>3.2</td>
<td>An example of a CDG</td>
<td>30</td>
</tr>
<tr>
<td>4.1</td>
<td>A file with a merge conflict.</td>
<td>34</td>
</tr>
<tr>
<td>4.2</td>
<td>The file after the merge conflict is resolved.</td>
<td>35</td>
</tr>
<tr>
<td>4.3</td>
<td>A combined diff format of the file.</td>
<td>35</td>
</tr>
<tr>
<td>4.4</td>
<td>A diff file shows reordering lines alphabetically.</td>
<td>37</td>
</tr>
<tr>
<td>4.5</td>
<td>The diff file which encodes choices.</td>
<td>38</td>
</tr>
<tr>
<td>4.6</td>
<td>The output of command line git log --merges --cc for one commit</td>
<td>40</td>
</tr>
<tr>
<td>5.1</td>
<td>Home Integration System (HIS) feature model</td>
<td>47</td>
</tr>
<tr>
<td>5.2</td>
<td>The CDG of HIS feature model</td>
<td>48</td>
</tr>
<tr>
<td>5.3</td>
<td>3 common cases of dead features.</td>
<td>51</td>
</tr>
<tr>
<td>5.4</td>
<td>A void feature model and its CDG.</td>
<td>52</td>
</tr>
<tr>
<td>5.5</td>
<td>Some examples of redundancies.</td>
<td>52</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The mapping of feature models to propositional logic (PL) [10]</td>
</tr>
<tr>
<td>2.2</td>
<td>3 plain programs produced by edit $A$ and $B$ in the linear edit model</td>
</tr>
<tr>
<td>5.1</td>
<td>The mapping of feature models to choice calculus expressions</td>
</tr>
<tr>
<td>5.2</td>
<td>The mapping of feature models to Choice Dependency Graphs</td>
</tr>
<tr>
<td>5.3</td>
<td>The mapping of $Or$, $Alternative$ and $Excludes$ relationship to Choice Dependency Graphs</td>
</tr>
</tbody>
</table>
Chapter 1: Introduction

This thesis addresses two problems in variational software by identifying tailored representations. These two problems are merge conflicts in Git and analysis operations on feature models. In this chapter I first define what is variational software. Then I describe those two problems and I will outline the structure of this thesis.

1.1 Variational Software

First I explain the meaning of “variational”. The adjective “variational” is opposite to “plain” when representing a software. A plain software could be a program, a document, a string etc. And a variational software is a family of plain softwares. For example, a piece of C code which includes #if in preprocessing part implements conditional compilation. It represents several different variants of a C program [26], and each variant can be obtained by selecting a specific configuration of C preprocessor. The variants of this C program are plain programs or non-variational programs.

In the following sections, I will point out variational softwares involved in the two problems my approach handles. Its variants are also explicitly discussed.
1.2 Merge Conflicts in Git

Git is a version control system which manages several versions of files. For one version, it corresponds to a new change in this file. Every time a user changes the file, this change introduces a new version of this file. In Git, a change corresponds to what is called a “commit”. Therefore, a file after a new commit has a new version. Git provides a difference file to show changes of a file after a specific commit. The diff file includes two variants: the old and the new version of this file.

Git supports a branch workflow, which means that users can make an independent line of development which diverges from the main line. The work on a branch does not interfere with that main line. The branches can be merged together to combine works, and in this process it is possible that the same part of a file is edited by two different branches. In this situation, Git is unable to merge them or select one of them. The merging process will pause in this case and wait for users to resolve such conflicts. Detailed information about Git and merge conflicts is given in Section 2.4.

However, in many cases merge conflicts are not semantically relevant. For example, when a user rearranges lines in a file where the order of lines does not matter as in rules in a makefile. Although this operation makes no change to the semantics of the file, it may cause conflicts when the user merges her branch to another. We say that such operations cause “non-conflicting” conflicts. In this thesis, I will provide a statistical analysis of the 50 most well-known repositories in Git. The data show about 27% of merge commits have conflicts, while in conflicts about 3% is caused by such move operation.
Semantically irrelevant conflicts could be eliminated automatically. Therefore my thesis proposes an approach to reduce the incidence of merge conflicts. I apply my approach to the difference file of merge commits, and use the Choice Edit Model (CEM) to represent changes in the difference file and identify the non-conflicting conflicts. I will discuss this method in Section 4.

1.3 Analysis Operations on Feature Models

A feature model describes the relationships and constraints of features for a software product line (SPL). It is variational, because it represents a family of all possible products of a SPL. Each product is a variant, which can be obtained by selecting certain features.

Analyzing feature models is an active area of research. Many researchers contributed a number of operations of analyses [18, 14]. Some of these analyses are for debugging a feature model. For example, we could determine whether a given feature model contains dead features, that is features that cannot be selected. Another operation determines the number of products represented by this feature model. More information about feature models is discussed in Section 2.1.

This thesis proposes Choice Dependency Graphs as a representation of feature models. A CDG is a succinct representation which describes the relationships of features in terms of dependency. By applying a series of rules for CDGs, we can implement several analysis operations for feature models. Especially for debugging feature models, CDGs provide a more obvious structure to detect anomalies.
A feature model is not transformed into a CDG directly. It is first represented as a Choice Calculus expression. The rules converting a feature model to a choice calculus expression are based on propositional logic. This thesis provides an algorithm that transforms feature models to choice calculus expressions to CDGs. I will discuss this algorithm in Section 5.

1.4 Thesis Outline

This section gives an overview of the rest of this thesis.

Chapter 2 (Literature Review) reviews 4 important concepts involved in this thesis from other researchers’ work. It presents the definition and properties of feature models, the choice calculus, the choice edit model, and version control systems. Feature models and version control systems are the objects to which I will apply my representations of changes, while the choice calculus and the choice edit model are the theoretical foundations of my representation.

Chapter 3 (Edit Representations) introduces a formalization of edits on a file: the edit sequence. It also presents how to convert the edit sequence to the choice edit model. It gives inference rules for this conversion and a Coq [1] program to implement machine-checked proofs.

Chapter 4 (Analysis of Merge Conflicts) gives a statistical analysis of 50 well known repositories in Git, which shows the incidence of conflicts in merge operations. Through using the choice edit model to represent changes of a file after
merging, the conflicts without semantical relevance can be identified. This chapter focuses on reducing conflicts that are caused by operations to reorder the contents of a file.

Chapter 5 (Supporting Operations on Feature Models by CDGs) presents the algorithm that transforms a feature model diagram to a Choice Dependency Graph (CDG). This chapter discussed how the analysis operations of a feature model could be implemented by analyzing the transformed CDG. It demonstrates that a CDG provides a succinct representation in which the defects in a feature model are much more obvious.

Chapter 6 (Conclusion) summarizes this thesis.
Chapter 2: Literature Review

This chapter presents several important concepts from other researchers’ work. These concepts are involved in my research work in different ways. In section 2.1, I introduce feature models, which are the objects of my approach’s application. In section 2.2, I present the choice calculus, which is an intermediate representation to translate changes in a program to my representations of edits. In section 2.3, I describe the formalization of an editing model in terms of the choice calculus. In section 2.4, I review several version control systems.

2.1 Feature Models

The Definition of Feature Models

A feature model is a representation of all possible products of a software product line (SPL) in terms of features. Feature models are widely used to manage variability and commonality of products from one SPL. Typically, a feature model has a hierarchical structure, which arranges the features via the relationships among them.

There are different feature model notations, and among them the most well known notations are basic feature models, cardinality-based feature models, and extended feature models [10]. Because the research in this thesis only studies basic feature models, the feature models discussed in this section are all basic feature models.
Relationships in Feature Models

Figure 2.1 shows a simple feature diagram, which is a visual notation of a feature model. A feature diagram depicts features, parental relationships, and cross-tree constraints.

![Feature Diagram]

Figure 2.1: A feature model for mobile phone proposed by Benavides et al [10].

A parental relationship is the relationship between a parent feature and its child feature. It includes:

- Mandatory: If a product has a parent feature, its child features which have a mandatory relationship with the parent feature are also included in this product. For example, in the feature model shown in Figure 2.1, every mobile phone must include the screen feature.

- Optional: When a parent feature appears in a product, the child features may be included in the product. In the example, GPS is an optional feature for a mobile phone.
• Alternative: An alternative relationship is for a set of child features with one parent feature. If a parent feature appears in a product, only one of the child features can be selected. In the example, the screen of a mobile phone can only support one of basic, color, or high resolution.

• Or: An inclusive or relationship is also for a set of child features with their parent feature. When a parent feature is included in a product, at least one of its child features must be selected in the product. In Figure 2.1, when the media feature is selected, camera, MP3, or both of them can be selected.

Cross-tree constraints include:

• Requires: If feature A requires feature B, then the selection of feature A in a product implies the inclusion of feature B.

• Excludes: If feature A excludes feature B, then these two features are incompatible, which means it is impossible to include both A and B.

Properties require debugging

There exist several tools supporting creation and configuration of a feature model. For example, the EMF Feature Diagram Editor [2] is a graphical editor integrated into the Eclipse platform. However, the support for debugging feature models is limited. Typically, the following properties of a feature model need to be validated.

• A feature model is not void: A feature model is void if it represents no product. The main reason for a feature model being void is the incorrect usage of cross-tree constraints. The automated checking of cross-tree constraints is critical for a feature model, especially for big ones [10].
• All features in a feature model are effective: A feature model represents a set of products. However, if a feature appears in no product in that set, this feature is “dead”.

• Optional features are not mandatory: An optional feature is optionally included in the products of a product line. When an optional feature appears in all products of a feature model, it should be modeled as mandatory [20]. This type of defect gives a wrong idea of the domain that represents a feature model.

• A feature model is not redundant: Redundancies in a feature model are the constraints that are implied in multiple ways. The occurrence of redundancy influences the maintainability of the model [24].

**Approaches to debug feature model**

Numerous approaches focus on identifying the above properties of a feature model. These approaches include propositional logic [9], constraint programming [11], and description logic [27]. My work is based on propositional logic. The mapping of relationships in feature models to propositional logic is shown in Table 2.1.

My approach transforms feature models to Choice Calculus expressions and Choice Dependency Graphs. The main goal of my work is improving the effectiveness of debugging feature models. The details about that transformation and applications of my methods will be described in Section 4.
### Feature Relationships

<table>
<thead>
<tr>
<th>Feature Relationships</th>
<th>Feature Diagrams</th>
<th>PL Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mandatory</strong></td>
<td><img src="10" alt="Mandatory Diagram" /></td>
<td>( P \leftrightarrow C )</td>
</tr>
<tr>
<td><strong>Optional</strong></td>
<td><img src="10" alt="Optional Diagram" /></td>
<td>( C \rightarrow P )</td>
</tr>
<tr>
<td><strong>Or</strong></td>
<td><img src="10" alt="Or Diagram" /></td>
<td>( P \leftrightarrow (C_1 \lor C_2 \lor \ldots \lor C_n) )</td>
</tr>
<tr>
<td><strong>Alternative</strong></td>
<td><img src="10" alt="Alternative Diagram" /></td>
<td>( (C_1 \leftrightarrow (\neg C_2 \land \ldots \land \neg C_n \land P)) \land ) ( (C_2 \leftrightarrow (\neg C_1 \land \ldots \land \neg C_n \land P)) \land ) ( (C_n \leftrightarrow (\neg C_1 \land \neg C_2 \land \ldots \land \neg C_{n-1} \land P)) )</td>
</tr>
<tr>
<td><strong>Requires</strong></td>
<td><img src="10" alt="Requires Diagram" /></td>
<td>( A \rightarrow B )</td>
</tr>
<tr>
<td><strong>Excludes</strong></td>
<td><img src="10" alt="Excludes Diagram" /></td>
<td>( \neg (A \land B) )</td>
</tr>
</tbody>
</table>

Table 2.1: The mapping of feature models to propositional logic (PL) [10]

### 2.2 The Choice Calculus

The choice calculus [17] provides a general and formal representation of software variations. The choice calculus expressions represent variational aspects of a program as two
or more alternatives. A code example in which a variable has two possible names (\(a\) and \(c\)) is shown below:

\[
\begin{align*}
\text{int } A(a,c) &= 1; \\
\text{int } b &= 2; \\
\text{int } x &= A(a,c) + b;
\end{align*}
\]

Label \(A\) is called a *dimension*. Choices labeled with the same dimension are synchronized. By definition, a choice is a set of alternative expressions. In the example, the choice is \(A(a,c)\), while the alternative expressions are \(a\) and \(c\). The following two patches of codes show the result of selecting the left and right alternative for \(A\). Note that for binary choices, we use selectors \(A_l\) and \(A_r\) to represent the selection of the left and right alternative, respectively.

<table>
<thead>
<tr>
<th>(A_l)</th>
<th>(A_r)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int (a = 1;)</td>
<td>int (c = 1;)</td>
</tr>
<tr>
<td>int (b = 2;)</td>
<td>int (b = 2;)</td>
</tr>
<tr>
<td>int (x = a + b;)</td>
<td>int (x = c + b;)</td>
</tr>
</tbody>
</table>

By selecting alternatives, choices are eliminated. If we assign the second choice in the example a different name, such as \(B(a,c)\), the two choices in the example are independent of each other. To get a non-variational program, both of those two choices need to be selected. For example, we can use selectors of \(A_lB_r\) and selectors of \(A_rB_l\) to produce the following programs that are not available if using only one dimension for choice.
between a and c.

<table>
<thead>
<tr>
<th>$A_iB_r$</th>
<th>$A_iB_l$</th>
</tr>
</thead>
<tbody>
<tr>
<td>int a = 1;</td>
<td>int c = 1;</td>
</tr>
<tr>
<td>int b = 2;</td>
<td>int b = 2;</td>
</tr>
<tr>
<td>int x = c + b;</td>
<td>int x = a + b;</td>
</tr>
</tbody>
</table>

Including choices in one program actually integrates many versions of this program. By selecting all choices, a variant of a program is produced, which is a non-variational expression. In general, $n$ independent choices can encode $2^n$ different programs [16].

As the choice calculus can provide a combined form of variations, it also supports the representation of variations in different ways. This highly flexible tool is helpful for users to understand and reason about variational structures.

For this thesis, the choice calculus plays a significant role in translating feature models into Choice Dependency Graphs, which will be discussed in Section 3. The foundation of that application is that feature models can be represented as choice calculus expressions. Features in a feature model are equivalent to choices. For example, the camera feature in Figure 2.1 can be represented as follows.

\[ \text{Camera}(\epsilon, \text{Camera}) \]

The dimension name of this choice is the feature name. The left alternative is $\epsilon$, which means the camera feature is not included in the product. The right alternative means the camera feature is selected. More mappings of feature relationships to choice calculus expressions can be found in Chapter 5.
To represent feature models, the syntax of choice calculus expressions is as follows. Note: the original syntax of the choice calculus can be found in [17].

\[ e ::= \epsilon \mid D \mid D\langle e, e \rangle \mid e \ldots e \]

The syntax says a feature model can be empty (\( \epsilon \)); it can include only one feature (\( D \)); it can be a choice of two feature models (\( D\langle e, e \rangle \)), and a concatenation of other models (\( e \ldots e \)). We can represent a simple feature model in Figure 2.2 by using a choice calculus expression. In this feature model, the feature  \emph{MobilePhone} has an optional relationship with its child feature  \emph{GPS}. The choice calculus expression for this feature model is

\[ \text{MobilePhone}\langle \epsilon, \text{MobilePhone GPS}\langle \epsilon, \text{GPS} \rangle \rangle \]

Therefore a product which includes the feature  \emph{MobilePhone} selects the right alternative of choice  \emph{MobilePhone}, and it may include the feature  \emph{GPS} by selecting the right alternative of  \emph{GPS}. If we select the left alternative of  \emph{GPS}, the product does not include the feature  \emph{GPS}. Here  \emph{MobilePhone} is the root of the feature model, therefore products which don’t include  \emph{MobilePhone} is not provided by this feature model.

![Figure 2.2: A simple feature model in which feature \textit{MobilePhone} has an optional child feature \textit{GPS.}](image-url)
2.3 The Choice Edit Model

The choice edit model (CEM) can be considered as an application of the choice calculus for representing program edits. The basic idea of the CEM is to represent a program edit that changes a program $P$ to $Q$ by a choice $A(P, Q)$ [16]. The selector $A_r$ represents the application of the program edit $A$, whereas not applying the edit $A$ is represented as $A_l$. $P$ is the old variant, while $Q$ is the new variant. An editing operation is considered as the introduction of $Q$ into the program $P$. Because choices only include the “old” and “new” variants, the choice in the CEM has only two alternatives.

The key property of the CEM is that it is a non-linear edit model, which supports selective undo. For example, the program shown in Section 2.2 is a variational program which is labeled as $V_A$:

$$\text{int } A\langle a, c \rangle = 1;$$

$V_A$

$$\text{int } b = 2;$$

$$\text{int } x = A\langle a, c \rangle + b;$$

In the context of program editing, edit $A$ introduces a variant of the variable’s name. The old name $a$ is changed to the new name $c$. Then we add one more change, which changes both $c$ and $b$ to $d$ in the program. This program edit is represented as a new dimension $B$, and the new variational program is $V_{AB}$:

$$\text{int } A\langle a, B\langle c, d \rangle \rangle = 1;$$

$V_{AB}$

$$\text{int } B\langle b, d \rangle = 2;$$

$$\text{int } x = A\langle a, B\langle c, d \rangle \rangle + B\langle b, d \rangle;$$
By applying both edit $A$ and edit $B$, the choices are eliminated, which yields the plain program $P_{bd}$ shown in table 2.2. The user can also undo the most recent edit, which is $B$ in this example, which yields the program $P_{ac}$.

<table>
<thead>
<tr>
<th>$P_a$</th>
<th>$P_{ac}$</th>
<th>$P_{bd}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>int a = 1;</td>
<td>int c = 1;</td>
<td>int d = 1;</td>
</tr>
<tr>
<td>int b = 2;</td>
<td>int b = 2;</td>
<td>int d = 2;</td>
</tr>
<tr>
<td>int x = a + b;</td>
<td>int x = c + b;</td>
<td>int x = d + d;</td>
</tr>
</tbody>
</table>

Table 2.2: 3 plain programs produced by edit $A$ and $B$ in the linear edit model.

However, there is a hidden version that is not reachable by a linear edit model: the version $P_{cb}$, which is obtained by applying edit $B$, but not applying edit $A$.

$$P_{cb} \quad \text{int a = 1;}
\quad \text{int d = 2;}
\quad \text{int x = a + d;}$$

From the variational representation $V_{AB}$ we know that part of edit $B$ is nested in edit $A$. In other words, $B$ partially depends on $A$. $P_{cb}$ can be interpreted as a partial selective undo. Specifically, it means reversing the previously applied edit $A$, while keeping the edit $B$. Because only part of $B$ depends on $A$, the independent part of $B$ is not affected by $A$. Undoing $A$ only reverses the dependent part of edit $B$.

As a special application of the choice calculus, the CEM has the syntax:

$$e ::= \varepsilon \mid p \mid D(e,e) \mid e...e$$

Compared with the syntax of choice calculus expressions in section 2.2, the CEM has a new case $p$, which represents a plain program without choices. It also omits the case $D$. 
which is only applicable to feature models. We can represent \( \text{int} \ A\langle a, B\langle c, d \rangle \rangle = 1 \); as 
\( e_1 A\langle e_2, e_3 \rangle e_4, \) where:

\[
\begin{align*}
e_1 &= \text{int} \\
e_2 &= a \\
e_3 &= B\langle c, d \rangle \\
e_4 &= =1;
\end{align*}
\]

In this expression, \( e_3 \) is a choice, and \( e_1, e_2, \) and \( e_4 \) are plain program text. The whole expression is a concatenation of program texts and choices.

According to the syntax of the CEM, it is possible that a choice is contained in the left alternative of another choice. For example, \( A\langle B\langle a, b \rangle, c \rangle \) represents an edit which changes \( a \) to \( c \), and alternatively to \( b \). Therefore \( a \) can be changed in two directions. These two edits are branches of each other. Considering the semantics of the CEM, once \( a \) changes to \( c \), edit \( B \) has no effect, because \( a \) no longer exists.

Until now I have introduced that a choice is nested in one alternative or a choice, \( A\langle a, B\langle c, d \rangle \rangle \) or \( A\langle B\langle a, b \rangle, c \rangle \). Besides that, there exists another situation that a choice appears in two alternatives of a choice, for example, \( A\langle B\langle a, b \rangle, B\langle c, d \rangle \rangle \). Edit \( B \) changes \( a \) and \( c \) to \( b \) and \( d \). But \( a \) and \( c \) are not compatible, and it is impossible to change them together. Therefore edit \( B \) is not valid and the whole expression corresponds to no real program edit.

In Section 4, I will present how to use the CEM to represent edits in version control systems, and by analyzing specific patterns in the CEM to find edits without semantical relevance.
2.4 Version Control Systems

As I mentioned in Section 2.2, choice calculus expressions support the management of software variations in multiple dimensions. The version control systems discussed in this section are managing program variations over time. Version control systems can be divided into two groups: centralized version control systems (CVCS) and distributed version control systems (DVCS).

Two most popular version control systems in the early days were Revision Control Systems (RCS) [22] and Source Code Control Systems (SCCS) [21]. After those, many new version control systems were developed and became popular. A well-known example is Concurrent Versions Systems (CVS) [12]. RCS and SCCS were designed to manage small-scale local projects, while CVS is well-suited for large internet-based open source software projects [13].

The systems mentioned above are all CVCSs, as they have a single canonical source repository. All developers work against this repository through a checkout taken from the repository, essentially a snapshot at the specific moment [15]. Every developer synchronizes and checks into the master branch. A commit of a change to the master branch can be seen by all the other programmers.

Unlike in CVCSs where a central server stores all versions of the program, a DVCS lets every developer clones a copy of the repository and have the full history of the project on their own hard drive [19]. Since everyone has a local copy, developers can make changes and roll back on their local machines. Most work is done offline, and developers connect to the server only when they decide to share changes. One of two
significant advantages of a DVCS over a CVCS is that a DVCS is much faster, because
developers only need to access the hard drive and not a remote server. The other advan-
tage is that branching and merging is much easier. Since every developer has their own
branch, it is easy to automatically combine changes [3]. In this section, I will introduce
Git [4], a DVCS that I will analyze in Chapter 4.

Git is aimed at non-linear workflow. It supports rapid branching and merging. A
typical workflow of Git is shown in Figure 2.3. Each commit in Git corresponds to a
change (i.e. an edit) applied to a file in a Git project. Every time a user pushes a commit,
the system takes a snapshot of all files of the user at that moment and stores a reference
to that snapshot.

Figure 2.3: A typical branching and merging workflow in Git [4].

I will describe the scenario of Figure 2.3 in this paragraph. Several developers, in-
cluding Joe, are working on a project, which creates a recipe of making lemon cake.
After commits C0, C1 and C2, the recipe has Flour, Sugar and Lemon. Those three
commits are all applied to the master branch, shown as orange in the figure. Then Joe
creates a new branch, whose commits are green in the figure. Creating a new branch
means Joe takes a copy of the recipe file in the master branch to his own branch. Joe then pushes the commit $C_5$ to his branch in order to add *Cream* to the recipe. Meanwhile, another developer who works on the master branch adds *Butter* (via $C_4$) in the recipe. Then Joe merges his branch into the master branch. Now Git does a three-way merge of two snapshots pointed to by the branch tips ($C_4$ and $C_5$), and the common ancestor of the two ($C_2$). If there is no conflicts, Git creates a new snapshot of the result of the merging, and it corresponds to a new commit.

However, this 3-way merging is not always executed smoothly. If users change the same part of the same file in two different branches, Git is not able to merge branches automatically. In the scenario in Figure 2.2, Joe’s branch and the master branch work on the same file (the recipe), and it changes the same lines in this recipe. On the master branch, the developer adds *Butter* after *Lemon*, while on Joe’s branch, he adds *Cream*. In this case, the system pauses to wait for the user to resolve the conflict.

If there is no $C_4$, the tip of master is $C_2$, which is the ancestor of Joe’s branch. In this case, merging Joe’s branch causes no conflicts. Because there is no change in the recipe on the master branch, Joe can merge his file directly and the new recipe is the same as Joe’s. This is called a “fast-forward” merging, which is not a true merge operation. No merge commits are created for this operation. More information about merge operations in Git can be found in [5].

Occasionally, conflicts occurred in merging are not real conflicts. For example, in Figure 2.3 if Joe reorders the ingredients on the recipe without adding *Cream*, it causes a conflict as shown in Figure 2.4. This conflict must be resolved manually to implement the merging. For the result of this merge commit, all features in these two branches
should be included. Specifically, the merged recipe should have a *Butter* element and reorder the elements. However, the order of items in the recipe actually has no effects on the semantics of the recipe. The merged recipe is semantically equivalent to the recipe after commit $C_4$. The conflicts occurred in this case are semantical irrelevant or “non-conflicting” conflicts. Manually resolving those conflicts is time-consuming and has no significant impact on the recipe. An automatic detection of such conflicts can improve the efficiency of merging branches in Git.

![Figure 2.4: Merge ordered recipe into master branch.](image)

In Chapter 4, I will present how to identify semantically irrelevant conflicts in Git by using the CEM to represent the merged program.
Chapter 3: Edit Representations

This chapter presents two edit representations: Edit Sequence (ES) and Choice Dependency Graphs (CDG). I first introduce ES in Section 3.1, and in Section 3.2 I present the relationship between ES and the Choice Edit Model (CEM). ES formalizes editing operations applied to a program and records the editing process of the user. When the user makes a change to a program, a new variant of this program is produced. The CEM is used to represents all variants of this program, and it is based on the ES of the program. In Chapter 4, I will discuss how to apply ES and CEM to identify conflicts in merged program.

In Section 3.3, I describe CDGs, and discuss the transformation from Choice Calculus Expressions (CCE) to CDG. In Chapter 5, I will use CCE to represent the logical relationships between features in feature models. The CDG derived from a CCE will be applied in several analysis operations.

3.1 Edit Sequences

An Edit Sequence (ES) is a sequence of edits, which represent the changes made to a text. Each edit is labeled by a dimension, and edits with the same dimension are synchronized. Besides a dimension, an edit also indicates the specific operation executed in this edit. The operations in ES are insert, delete, change, undo, and redo. The
insert, delete, and change operations also indicate the position in the text where this edit is applied.

The following is the syntax of ES. $D$ is the dimension of an edit and $op$ refers to editing operations. Each edit is labeled by a dimension name to identify it. Edits with the same dimension are applied synchronously. $pos$ indicates the position where to apply editing operations. For insert and change operations, $lit$ after $op$ refers to the plain string that the editing operations introduce. $D$, $lit$ and $pos$ are all nonterminals.

$$es := edit^*$$

$$edit := \text{fordim } D \ op$$

$$op := \text{insert } pos \ lit$$

$$\mid \text{delete } pos$$

$$\mid \text{change } pos \ lit$$

$$\mid \text{undo}$$

$$\mid \text{redo}$$

For a string $s$, its positions are as shown Figure 3.1. An edit $\text{fordim } A \ \text{insert } 0 \ x$ has dimension $A$, and adds $x$ before $a$. The new string $t$ has updated positions, and the new position 0 is the gap before $x$. The delete and change operations delete or change the character after $pos$. An edit $\text{fordim } B \ \text{delete } 1$ deletes the second character and changes the string to $u$.

An undo operation reverses an edit, and a redo operation re-applies edits. For example, an edit $\text{fordim } B \ \text{undo}$ changes the string $u$ back to $t$. An edit $\text{fordim } B \ \text{redo}$ changes $t$ to $u$ again. The undo operation can only reverse edits that have been
applied, and **redo** operations re-apply the edits that have been reversed.

3.2 Transforming Edit Sequences to the Choice Edit Model

ES is a formalization of edits that occur in a program, but it is hard to find specific edit patterns in a program. As discussed in Section 2.4, the CEM represents edits in terms of choices and reveals special edits. For example, in Chapter 4 I will use the CEM to represent changes in commits of Git, and a special edit called move is identified. ES is a bridge which connects the real changes in a program to choice-structured edits.

The complete rules to translate ES to CEM are presented in the Appendix. In this section, I introduce several important transformation rules to illustrate the basic. First, I define the judgement forms used in the rules.

**Judgement:**

A judgement states that one or more objects have some properties or are in some relation to each other. The syntax of the judgements in rules that transform ES to CEM is:
es ⊢ (vs, Γ) → (vs, Γ)

es has type Edit Sequence. Γ is an environment of selections which have been made before the ES applies. Γ is a set of pairs \((Dimension, Bool)\). If the boolean value is True, the right alternatives of choices with this dimension are selected. In another words, the right alternatives are included in the variant produced by those selections. If the boolean value is False, the left alternatives of choices with the dimension are selected. vs has type Variational String (VS). It is a string which may contain choices. The VS has the following grammar:

\[
e ::= s \mid D\langle C, C \rangle \mid e...e
\]

where \(s\) is a plain string. For example, the VS \(A\langle b, c\rangle\) at represents the two strings bat and cat. By selecting the left alternative of choice \(A\), it is bat, and it is cat if the right alternative is selected.

The judgement states that an ES applies to a VS with selections leads to another VS with updated selections.

Axioms:

In a proof system, an axiom is taken to be true and serves as a premise for further transformations. For the rules to transform ES to CEM, axioms describe one edit applied to one character.

\[
[\text{fordim } A \text{ insert } 0 b] \vdash (a, \delta) \rightarrow (A\langle \varepsilon, b \rangle a, \{(A, True)\} \cup \delta)
\]

\[
[\text{fordim } A \text{ delete } 0 b] \vdash (a, \delta) \rightarrow (A\langle a, \varepsilon \rangle, \{(A, True)\} \cup \delta)
\]
There are two more axioms about *undo* and *redo* edits, and they are discussed in Section 3.3. For the rules above, the first one describes inserting $b$ before $a$. The edit sequence in this rule has only one edit, and it applies to $a$, with an environment of selections $\delta$. Here $a$ is a plain character and after the insertion operation, the empty string before $a$ changes to $b$. This operation forms a choice $A(\epsilon, b)$ before $a$. The choice and $a$ are joined to form a new VS: $A(\epsilon, b)a$. Because the edit $A$ is applied, the new environment includes the pair $(A, True)$. The second rule for the delete operation can be considered as an inverse operation of insertion.

**Inference Rules:**

An inference rule derives logical conclusions from premises known to be true. In this section, I introduce 3 important inference rules:

\[
\begin{align*}
\text{[fordim A op]} & \vdash (vs1 (vs2 vs3), \delta) \rightarrow (vs', \delta') \\
\text{[fordim A op]} & \vdash ((vs1 vs2) vs3, \delta) \rightarrow (vs', \delta') \\
\text{es1} & \vdash (vs1, \delta) \rightarrow (vs2, \delta') \\
\text{es2} & \vdash (vs2, \delta') \rightarrow (vs3, \delta'') \\
\text{(es1 + +es2)} & \vdash (vs1, \delta) \rightarrow (vs3, \delta'')
\end{align*}
\]

\[
\text{[fordim A op]} \vdash (vs1, \delta) \rightarrow (vs1', \delta')
\]

\[
\text{[fordim A op]} \vdash (D(vs1, vs2), \{(D, False)\} \cup \delta) \rightarrow (D(vs1', vs2), \{(D, False)\} \cup \delta')
\]

The first rule describes the associative properties of joined VSs. No matter the order in which VSs are joined, for the same edit, the result VS $(vs')$ and the updated environment of selections $(\delta')$ are the same.

The second rule presents that two edit sequences could be concatenated as long as
the result of the first sequence is the VS and the selection environment for the second sequence to apply. The ++ symbol in this rule means concatenation.

The third rule explains that when a VS is changed by an edit sequence, by putting this VS in a choice, the same edit sequence makes the same effect on the VS in the choice. An edit sequence changes vs1 to vs1' and the environment of selections δ to δ'. For a choice which has vs1 as the left alternative, the edit sequence also changes vs1 to vs1' in the choice. To limit the object to apply the edits, the environment δ must have a pair (D, False) to select vs1. In the choice edit model, the left alternative corresponds to the False value in (Dimension, Bool), and the right alternative refers to True. Besides this pair, the environment δ changes to δ'. Here exist a constraint that the initial environment for vs1 changing to vs1' in the predicates is the same as the one for the choice containing vs1. There also exits a symmetric rule for changing the right alternative in a choice. This rule is shown in Appendix.

3.3 Properties of the Transformation

In this section, I introduce two invariants for the CEM.

**Undo and redo edits have no effect on the CEM representation** An undo operation reverses an edit and takes the program back to the previous version. However, it introduces no new variants with a choice. The alternatives of the choice are the same as before applying the undo. The same also holds for the redo operation. The axioms for undo and redo are shown below.
undo means to unselect a choice, while redo implies selection of a choice.

Order-independence of edits  An example of this property is changing a string $ac$ to $bd$. This change is implemented by an edit sequence including two edits: edit with dimension $A$ changes $a$ to $b$, and edit with dimension $B$ changes $c$ to $d$.

In this case, if we reorder edit $A$ and $B$, the result is:
The result is completely different with the previous order. Therefore, only the Edit Sequences which form independent CEMs can be reordered.

Theorem 1 For all dimension $D$, variational string $vs$ and environment of selections $\delta$,

\[
[\text{fordim } D \text{ undo}] \vdash (vs, (A, True) \cup \delta) \rightarrow (vs, (A, False) \cup \delta)
\]

\[
[\text{fordim } D \text{ redo}] \vdash (vs, (A, False) \cup \delta) \rightarrow (vs, (A, True) \cup \delta)
\]

The symbol $\vdash$ refers applying edit sequences, and $\rightarrow$ leads a result. This theorem is machine-checked by the Coq program in the Appendix B. The proofs of theorem undo-gen and redo-gen in the Coq program prove that undo or redo an edit has no effect on the result choice edit model expression. It only changes the environment of selections.

The second invariant is proved by proving theorem reorder1 and reorder2 in the Coq program. Basically it describes that:

For all dimension $X,Y$, plain character $a,b,c,d$, environment of selections $\delta$:

\[
[\text{fordim } X \text{ change } 0 c,\text{fordim } Y \text{ change } 1 d] \vdash (ab, \delta) =
\]

\[
[\text{fordim } Y \text{ change } 1 d,\text{fordim } X \text{ change } 0 c] \vdash (ab, \delta)
\]

Edit $X$ and $Y$ are independent, because the parts they changed have no overlap. ($X$
changes $a$, and $Y$ changes $b$). Theorem reorder shows that reordering those two edits produces the same result. This invariant can be generalized to indexes other than 0 and 1. The proof of this case will be included in the future work.

3.4 Choice Dependency Graphs

A choice dependency graph (CDG) is a graph that represents the choices in terms of their dependency relationship. A CDG is not a DAG, because it is possible to have cycles. An example of a CDG with cycles is shown in Figure 5.3 (c). The vertices correspond to the dimensions of choices while each edge corresponds to a dependency relationship between two choices. The syntax of a CDG is

\[
cdg ::= D^* \text{edge}^*
\]

\[
\text{edge} ::= \text{chain} D_1 D_2
\]

\[
\quad \mid \text{branch} D_1 D_2
\]

$D_1$ and $D_2$ are the dimensions of choices, while chain and branch describe the dependency of two choices. chain means a choice is the right alternative of another choice. We also say it is nested in another choice. In contrast, branch indicates a choice in the left alternative of another choice. The expression $A\langle B\langle a, b \rangle, C\langle c, d \rangle \rangle$ is represented as $[A, B, C][\text{branch} A B, \text{chain} A C]$.

An example of a CDG is shown in Figure 3.2. In a CDG, a line (usually pointing to the right) represents a chain edge, and a dashed line (pointing to the left) corresponds to a branch edge. In this CDG, the vertical $DE$ connects to $C$ via a dash. This means
choice $D$ has branch relationship with $C$, choice $E$ has chain relationship with $D$, and $D$ and $E$ are independent. The independent choices with the same relationship to another choice are concatenated to form one vertex. $C$ and $F$ are also two choices with chain relationship with $A$, but because $F$ has dependency with $C$, they cannot be concatenated.

![CDG Diagram](image)

Figure 3.2: An example of a CDG

The expression for this CDG is $A\langle B(\_,\_), C\langle D(\_,\_), E\langle\_,\_), F\langle\_,\rangle\rangle F\langle\_,\rangle\rangle$. The symbol $\_$ means the alternatives are not fixed, because the CDG only shows relationship of choices, the content in this choice is not represented in the graph. This expression shows that $D$ and $E$ are independent and $F$ is partially dependent with $C$.

Therefore the dimensions has a *many to one* relationship with vertices. Dimensions can be combined to form one vertex, but there is no multiple vertices representing one dimension appear in a CDG.
3.5 Translating Choice Calculus Expressions into Choice Dependency Graphs

In this section, I introduce several important inference rules for translating a CCE into a CDG.

**Axioms**

\[
\begin{align*}
\varepsilon & \rightarrow [ ] [ ] \\
D(a, b) & \rightarrow [D] [ ] \\
D(a, E(b, c)) & \rightarrow [D, E] [\text{chain } D E] \\
D(E(a, b), c) & \rightarrow [D, E] [\text{branch } D E]
\end{align*}
\]

The first rule describes that an empty CCE (\(\varepsilon\)) constructs an empty CDG. In the second rule, a CCE with dimension \(D\) has no choices in the alternatives, and it constructs a CDG with only one vertex. The third rule describes that if one choice \(E\) is nested in choice \(D\), they form a *chain* edge with two dimensions. The last rule shows if one choice \(E\) is the left alternative of choice \(D\), it has *branch* edge with \(D\).

**Inference rules**

\[
\frac{cce_1 \rightarrow cdg_1 \quad cce_2 \rightarrow cdg_2}{cce_1 cce_2 \rightarrow cdg_1 \cup cdg_2}
\]

The \(cdg_1 \cup cdg_2\) means the union of \(cdg_1\) and \(cdg_2\). The union of a CDG with vertex set \(V\) and edge set \(E\) and a CDG with vertex set \(V'\) and edge set \(E'\) is a new CDG. This CDG has vertices that are the union of \(V\) and \(V'\) and edges that are the union of \(E\) and \(E'\).
\((V,E) \cup (V',E') = (V \cup V', E \cup E')\) This inference rule explains for joined CCEs, the result CDG can be considered as the union graph of the CDGs transformed from each those CCE.

A CDG is valid if the CCE which is represented by this CDG is valid. A CDG is invalid if there exist two paths with common start and end point such that one path contains a branch edge and the other path contains a chain edge.

An example of this kind of CDG is shown in following figure. This CDG represents an expression: \(A\langle B\langle C\langle \_\rangle \rangle, D\langle \_\rangle \rangle\) In this expression a C choice appears in the left and right alternatives of A. This violates a constraint of the CEM.
Chapter 4: Analysis of Merge Conflicts

In this chapter, the editing of a program in Git is translated to Edit Sequence, and the merged program is transformed to a Choice Edit Model. These two representations are used to identify the conflicts occur in merging two branches in Git, especially for the conflicts without semantic relevance. In the Section 4.1, I introduce the algorithm of finding conflicts in merged program. In Section 4.2, I describe the method of identifying non-conflicting conflicts. In Section 4.3, I describe an experiment of 50 well-known repositories in Git to answer the following questions:

1. How many merge commits have conflicts.
2. In these conflicts, how many of them can be treated as “non-conflicting”.

4.1 Identifying Conflicts in Merge Commits of Git

As a version control system, Git supports branching and merging the work of developers. Every developer can create her/his own work copy, i.e. create a branch, and then the developer can merge it to another branch. A typical workflow is that the repository has a master branch which maintain the work has been compiled and tested successfully. Developers work on different part of the repository and merge their work back to the master branch.

However, if developers change the same part of the same file in different branches,
merging these two branches leads to conflicts. The merging process pauses in this case and waits for the developer to resolve the conflicts. Therefore we can define that conflicts are the parts of a file which are different in different branches. The developer must resolve these different parts to finish the merge commit.

To resolve merge conflicts, the developer can open the file that has conflicts using an editor. Figure 4.1 shows a file which has a merge conflict. "<<<<<<<" is the conflict marker, which indicates that the following content is merge conflict. The change from HEAD is after the line "<<<<<<<HEAD". HEAD is a reference to the currently checked out commit. It is actually a reference to the branch the developer have checked out. The next line is "=======", which divides the change from HEAD to the other branch, i.e. branch1. The change from the branch1 is followed by ">>>>>>>branch1" [6].

The developer compares the content from different branches and then decides to keep one of them or create a brand new change, which may incorporate from both branches. The developer deletes the markers and makes changes she/he wants in the final merge. In this example, the developer decides to maintain both the changes and revises the file to what is shown in Figure 4.2. Then the developer can commit the change and merge these two branches.
If you have questions, please email me via wangx4@oregonstate.edu or find me in my office KEC3048.

Figure 4.2: The file after the merge conflict is resolved.

To get the changes after the merge commit, we can use the command line git-diff to show the difference between the merge commit and its parent commits. The default format when showing merges with git-diff is a combined diff. It is produced by the command git-diff with the “--cc”. As the document of Git shows, --cc controls the format of diff output [7].

A combined diff format of the example above is shown in Figure 4.3.

```diff
diff --cc Test1
index 73b280d,f9d9a01..1943447
--- a/Test1
+++ b/Test1
@@ -1,1 -1,1 +1,1 @@
If you have questions, please
-email me: wangx4@oregonstate.edu.
-find me in my office: KEC3048.
++If you have questions, please email me via wangx4@oregonstate.edu or find me in my office KEC3048.
```

Figure 4.3: A combined diff format of the file.

I will introduce all parts of this format. The combine diff format preceded with a “git diff” header, use

```diff
diff --cc Test1
```

The header shows the option is --cc. It is followed by one or more extended header lines. In this example, it has the index line,

```index 73b280d,f9d9a01..1943447```
which indicates the file that stores information to prepare this merge commit. It is also
called the “staging area” between the working directory and Git directory. (This is not
the focus of this thesis, and more information can be find in [8].) Next is a two-line
from-file/to-file header. For the merge commit, the files to be compared are the same
(*Test1* in this case). The next is the chunk header. The number of @ characters equals
the number of parents + 1. In the example, there are two parents, so 3 @ lead the chunk
header. Between the @ characters the ranges of lines are shown, which are changed in
merging process.

```
@@@ -1,1 -1,1 +1,1 @@ @@@
```

The next part is the most significant part, which shows the difference between the merge
result and its parents.

A - character means that the line appears in one of the parent files, but it does not
appear in the result. A + character means that the line appears in the result, and one of
the parents does not have that line. In the example, in each parent one line is removed.
These two lines are labeled with -. Two + symbols (++) lebles a line which was added
in the result and does not appear in both parents.

The --cc option is used to produce combined diff format also compresses the output
by omitting uninteresting differences. These uninteresting differences appear when
the contents in the parents have two variants, and the merge result picks one of them
without modification. In the example, if the developer decides to merge the branches by
overwriting the other branch, the result is the same has HEAD. This kind of merge is
not included in this thesis.
4.2 Identifying Conflicts Without Semantic Relevance

Figure 4.4 is a diff file of a merge commit.

```
"gulp-eslint": "~0.9.0",
"gulp-inject": "~1.0.2",
"gulp-jsonlint": "1.1.0",
+ "gulp-less": "3.0.3",
+ "gulp-minify-css": "~0.5.1",
"gulp-nodemon": "2.0.3",
"gulp-notify": "2.2.0",
"gulp-plumber": "1.0.1",
- "gulp-less": "3.0.3",
- "gulp-minify-css": "~0.5.1",
"gulp-reduce-file": "0.0.1",
"gulp-rev": "6.0.1",
"gulp-rev-replace": "0.4.2",
```

Figure 4.4: A diff file shows reordering lines alphabetically.

The lines begin with - are lines deleted after the merge commit, and lines begin with + are the lines added. The lines without these symbols are the context of those changed lines. We call this kind of lines context lines.

Because this merge commit has a diff file, it caused conflicts. The user resolved the conflicts and committed the merge successfully. In order to resolve conflicts, the developer removed two lines and in another position added these two lines. This editing actually can be considered as a move operation that moves the two lines. The new file arranges lines alphabetically. However, the order of the lines actually does not affect the semantics of this file. This program can be compiled successfully without this reordering. This kind of conflicts which has no semantical relevance is called "non-
conflicting” conflicts. Observing the file above, we can find that it has a special editing pattern: the same line removed and added again in another place.

To identify a move operation, we can translate the diff file to variational strings (VS) that show the changes introduced by the merge commit. The context lines with removed lines form the old variant of this program, and the context lines with added lines are the new variant of this program after the merge. As I discussed in the Chapter 3, the Choice Edit Model is transformed from Edit Sequences (ES). So the first step is representing the diff file using ES.

ES indicates the editing operations and the positions to apply the edits. For example, the merge commit added a line "gulp-less": "3.0.3". This editing can be translated as an edit: **fordim A insert** 0 "gulp-less": "3.0.3". For the deleting operation the edit is **fordim B delete** 0 "gulp-less": "3.0.3". The symbols - and + are removed, because they are just used to label the lines.

```
"gulp-eslint": "~0.9.0",
"gulp-inject": "~1.0.2",
"gulp-jsonlint": "1.1.0",
A⟨ε,"gulp-less": "3.0.3"⟩
B⟨ε,"gulp-minify-css": "~0.5.1"⟩
   "gulp-nodemon": "2.0.3",
   "gulp-notify": "2.2.0",
   "gulp-plumber": "1.0.1",
C⟨"gulp-less": "3.0.3",ε⟩
D⟨"gulp-minify-css": "~0.5.1",ε⟩
   "gulp-reduce-file": "0.0.1",
   "gulp-rev": "6.0.1",
   "gulp-rev-replace": "0.4.2",
```

Figure 4.5: The diff file which encodes choices.

Applying the inference rules to transform ES to CEM, the diff file above can be trans-
lated into a VS, which is shown in Figure 4.5. Edit A and B change empty to 2 lines of codes, while the edit C and D change two lines of codes to empty. Edit A and C move the line "gulp-less": "3.0.3", while edit B and D move the line "gulp-minify-css": "~0.5.1". Observing edit A and C, B and D we can find the left alternative of A is the right alternative of C and the right alternative of A is the left alternative of C. Therefore we can summarize that two edits in which one edit swaps the alternatives of another edit as forming a move operation. The conflicts caused by the move operation can be considered as “non-conflicting”.

4.3 Experiment

In this section, I introduce the experiment which analyzes merge conflicts of 50 well-known repositories\(^1\) in Git. (The list of these repositories can be found in Appendix C.) This result of this experiment shows that about 27% of merge commits have conflicts, and among these conflicts, about 3% are caused by the move operation.\(^2\) In Section 4.3.1, I present how to perform the experiment. In Section 4.3.2 and Section 4.3.3 I describe the data and discuss the result.

\(^1\)These are the 50 most starred repositories on Github in October 2015.
\(^2\)All data is based on the repositories on October 25th, 2015.
4.3.1 Analyzing Merge Conflicts in Repositories

As the previous sections show, the diff files of merge commits provide information about conflicts and move operation. The command line to get all diff files of merge commits is

```
git log --merges --cc
```

The first part `git log` shows a record of commits. The option `--merges` filter commits to print only merge commits. The `--cc` option is the same as used in section 4.1. It omits uninteresting hunks whose content in the parents have only two variants and the merge result picks one of them without modification [7]. The output of this command line consists of logs for each merge commit. Figure 4.6 shows the log for one commit. The red line number is not in the log file, I use them to label each line.

```
commit 1ed9b9e2e2b62f6ef89558f98d786e9fcd649152
Merge: 35dac50 fefb779
Author: Carl Meyer <carl@oddbird.net>
Date:   Sat Nov 15 18:56:52 2014 +0100
Merge pull request #3536 from Zweedeend/ticket_23837
Fixes #23837: Replace list with deque in migration-planner for improved performance.
diff --cc django/db/migrations/graph.py
index b454d17,dd3106a..a2c14f2
--- a/django/db/migrations/graph.py
+++ b/django/db/migrations/graph.py
@@ -1,4 -1,5 +1,5 @@@
 from __future__ import unicode_literals
+ from collections import deque
 from django.db.migrations.state import ProjectState
 from django.utils.datastructures import OrderedSet
```

Figure 4.6: The output of command line `git log --merges --cc` for one commit

The 1st line shows the commit ID. It is forty hexadecimal characters that specify a 160-bit hash value. The 2nd line indicates that this is a merge commit, and the IDs of the commits it merges are shown after `Merge`. The 3rd and 4th lines show the author and the
date of this commit. The first 4 lines can be considered as the header of this commit log. Lines 5 ~ 9 are for the comments of this commit. It describes what is merged and why the repository needs this merge. Lines 10 sim 19 show the combined diff format of this merge commit, which I have discussed in Section 4.1.

To perform this experiment, I have created a Haskell program to scan the log file. First, the program calculates how many merge commits have a combined diff format. Some merge commits have only header and comments, and these commits merge two branches without conflicts. Second, the program divides the log file to logs for each commit. Because one commit changes several files, the log for one commit is divided to logs for each file. The log of one commit for one file is analyzed and identifies the move operations. This kind of log can be represented as log\textsubscript{mn}, which means it is the log for the \textit{m}th commit, and the \textit{n}th file in this commit.

The program then analyzes the diff file of log\textsubscript{mn} and transforms the diff file to variational strings. The algorithm has been discussed in Section 4.2. The next step is scanning all choices to find choice pairs which have swapped alternatives. Any such choice pair forms a move operation.

### 4.3.2 Data Description

The experiment basically produces 4 kinds of data.

1. \(M\): the number of merge commits in this repository.

2. \(M_{\text{Conflicts}}\): the number of merge commits with conflicts.
3. $C$: the number of conflicts in all merge commits.

4. $C_{\text{Moves}}$: the number of conflicts which are caused by move operations.

Then we can define the following two measures.

- $P_{\text{Conflicts}} = \frac{M_{\text{Conflicts}}}{M}$
  
  It shows what percentage of merge commits have conflicts.

- $P_{\text{Moves}} = \frac{C_{\text{Moves}}}{C}$
  
  It shows what percentage of conflicts are caused by move operations.

To get a review of all 50 repositories, I averaged $P_{\text{Conflicts}}$ and $P_{\text{Moves}}$ to get the result for these 50 repositories.

4.3.3 Results

The averaged result of 50 repositories shows that about 27% of merge commits have conflicts, and in these conflicts about 3% are caused by move operations. The evaluation has shown that this analysis does not seem to be widely applicable. Therefore, it probably should not be built into a tool, even though it might be quite useful in individual cases. For example, the repository vhf/free-programming-books has 10% conflicts are caused by the move operations.
Chapter 5: Supporting Operations on Feature Models by CDGs

In this chapter, I present how to translate feature models (FMs) to Choice Dependency Graphs (CDGs). The reason of using a CDG to represent a FM is that the CDG simplifies analysis operations of a FM. FMs are first translated to Choice Calculus Expressions (CCEs) based on propositional logic. The CCEs are then transformed to CDGs based on the rules discussed in Chapter 3. Four analysis operations are discussed in Section 5.2. The result shows manipulating translated CDGs simplify the analysis operations on feature models.

5.1 Translating a Feature Model to a Choice Dependency Graph

As discussed in Section 2.1, there exists a mapping of relationships in feature models to propositional logic. We can build CCEs based on the propositional logic representation of feature models. Table 5.1 extends Table 2.1 with one more column which shows the corresponding choice calculus expressions.

For simplicity, I show only the case for two child features in Or and Alternative relationships. The meaning of all CCE mappings are explained in the following paragraphs.

- Mandatory: As the PL mapping shows, features $P$ and $C$ are both either selected or not. Therefore, these two features can be grouped to form a “grouped feature”.
A product can have feature $P$ and feature $C$ or neither of them.

- **Optional**: If feature $P$ is selected, feature $C$ may be selected in which case the result is $PC$. If $C$ is not selected, we get only $P$.

- **Or**: After including $P$, the product must select at least one of the features $C_1$ and $C_2$. By selecting one of the three alternatives of $X$, it covers all possible selections. $C_1C_2$ means selecting $C_1$ and $C_2$ together. Here $X$ is a virtual dimension. It represents no feature, and it is only used to limit the relationship of features.

- **Alternative**: This relationship is similar to **Or**. Feature $P$ with only one of $C_1$ and $C_2$ can be selected.

- **Requires**: Because $A$ is in the right alternative of $B$, selecting $A$ implies $B$ has been selected.

- **Excludes**: A virtual dimension $X$ lets one of $A$ and $B$ be included in the product.

In Section 3.5, I present several rules to translate CCEs to CDGs. Table 5.2 shows CDG mappings for feature relationships. However, observing the tables we can find that in

<table>
<thead>
<tr>
<th>Feature Relationship</th>
<th>PL Mapping</th>
<th>CCE Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mandatory</strong></td>
<td>$P \leftrightarrow C$</td>
<td>$P,C,\epsilon, [P, C]$</td>
</tr>
<tr>
<td><strong>Optional</strong></td>
<td>$C \rightarrow P$</td>
<td>$P,\epsilon, PC, [P, C]$</td>
</tr>
<tr>
<td><strong>Or</strong></td>
<td>$P \leftrightarrow (C_1 \lor C_2)$</td>
<td>$P,\epsilon, PX, (C_1, C_2, C_1C_2)$</td>
</tr>
<tr>
<td><strong>Alternative</strong></td>
<td>$(C_1 \leftrightarrow (\neg C_2 \land P)) \land (C_2 \leftrightarrow (\neg C_1 \land P))$</td>
<td>$P,\epsilon, PX, (C_1, C_2)$</td>
</tr>
<tr>
<td><strong>Requires</strong></td>
<td>$A \rightarrow B$</td>
<td>$B,\epsilon, BA, (\epsilon, A)$</td>
</tr>
<tr>
<td><strong>Excludes</strong></td>
<td>$\neg (A \land B)$</td>
<td>$X, (A, B)$</td>
</tr>
</tbody>
</table>

Table 5.1: The mapping of feature models to choice calculus expressions
CDGs for Alternative, Or and Excludes relationships, features \( C_1, C_2, C_1C_2, A \) and \( B \) are missing. The reason of this situation is that they are all alternatives of choices, and not the dimensions. As the definition of a CDG, it only shows the dimensions. Therefore, some features are not represented in CDGs. This loss of information has some consequences. First, some relationships of features are not shown in CDGs. Second, branch edges of CDGs are not used, and CDGs are more like lists, not graphs. In order to solve this problem. I adjust the construction rule of CDGs to make it applicable to feature
models.

Here I propose a phrase “pseudo choice” to represent a choice in which the right alternative must be selected. For instance, in *Excludes* relationship, the left alternative $A$ can be translated to a pseudo choice: $A(ε, A)$. The right alternative of this choice must selected, which is $A$. Now feature $A$ can be represented in the CDG, and it connects to $X$ via a branch edge. The case is similar for $B, C_1, C_2$ and $C_1C_2$. CDGs mappings for *Or, Alternative, and Excludes* are shown in Table 5.3.

<table>
<thead>
<tr>
<th>FR</th>
<th>CDG Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Or</strong></td>
<td><img src="image" alt="Or CDG" /></td>
</tr>
<tr>
<td><strong>Alternative</strong></td>
<td><img src="image" alt="Alternative CDG" /></td>
</tr>
<tr>
<td><strong>Excludes</strong></td>
<td><img src="image" alt="Excludes CDG" /></td>
</tr>
</tbody>
</table>

Table 5.3: The mapping of *Or, Alternative* and *Excludes* relationship to Choice Dependency Graphs

In the *Or* relationship, a new virtual dimension $Y$ appears. The reason for adding this dimension is that CDGs are only used to represent binary choices. The CCE mapping for *Or* is $P(ε, PX(A, C_1C_2))$, which has 3 alternatives in $X$. Because $A(a, b, c)$ is equivalent to $A(a, B(b, c))$, that expression is equivalent to $P(ε, PX(A, C_1, Y(A, C_1C_2)))$. 
Based on the CDG mappings, a feature model can be translated to a CDG. Figure 5.1 shows a feature model of a Home Integration System (HIS) [23]. The translated CDG for this feature model is shown in Figure 5.2.

There are 3 kinds of features in the Figure 5.2:

- The root mandatory feature: The vertex with a * symbol represents a group of root mandatory features. It is different from the root feature in a feature model. For example, in Figure 5.1, feature HIS is the root feature. However, because HIS has mandatory relationship with several features, and these features also have mandatory features, they form a group feature HIS, supervision systems, control, light control, temperature. All features in this group must be included in all products belong to this feature model.
Figure 5.2: The CDG of HIS feature model

- The virtual feature: Features that start with Virtual are virtual features. The numbers following Virtual are used to distinguish different virtual features.

- The regular Feature: The remaining features are all regular features.
5.2 Using Choice Dependency Graphs to Support Analysis Operations for Feature Models

In this section, I present how to manipulate translated CDGs to support 4 analysis operations of feature models.

5.2.1 Valid Product

This operation takes a feature model and a product as input and returns whether this product belongs to this feature model. A valid product should include the root feature of the feature model. Besides that, features which have mandatory relationship with the root feature must be included also. The root feature with its mandatory features exactly form the root mandatory feature, which is labeled with * in CDGs.

Therefore, if the features of a product include all features in the root mandatory feature, this product is valid. For instance, we have known in the HIS feature model, the root mandatory feature is HIS, supervision systems, control, light control, temperature. Consider the products $P_1$ and $P_2$:

$P_1 = \{\text{HIS, supervision systems, control, light control, temperature, appliances control}\}$

$P_2 = \{\text{HIS, supervision systems, control, temperature, ADSL, video on demand}\}$

Product $P_2$ is not valid, because it does not include the mandatory feature light control.
5.2.2 Dead Features

A feature is dead if it cannot appear in any product of a feature model. Figure 5.3 shows two cases of dead features and the corresponding CDGs.

For the case (a) and (b), since the root mandatory feature \(A,B\) and \(A,B,C\) must be selected, any features which appear oppositely to the root mandatory feature is dead. “Appear oppositely” means this feature and the root mandatory feature are two alternatives in a choice. For (a), \(D\) is the left alternative of virtual dimension \(Virtual\ 2\), and \(A,B,C\) is the right alternative of \(Virtual\ 2\). Therefore, \(D\) is dead. For case (b), \(C\) is dead.

In Figure 5.2, \(Power\ Line\) appears oppositely to the root mandatory feature, so it is a dead feature. For a big scale feature model, the situation is complicated. Therefore, we have the following criterion to determine whether a feature is dead:

\[
\text{Let } cdg \text{ be a Choice Dependency Graph translated from a feature model } fm. \text{ Let } r \text{ be the root of } cdg. \text{ If there is another feature } f, \text{ which connects to } r \text{ via multiple paths, and two edges of those paths form a “divergent” structure, then } f \text{ is dead.}
\]

A “divergent structure” means one branch edge and one chain edge from one vertex. For instance, the root of CDG in case (a) is \(Virtual\ 2\), and \(D\) connects to \(Virtual\ 2\) via two paths. Two edges in those paths are the branch and chain edges of \(Virtual\ 2\). Therefore, \(D\) is dead.

Case (c) is a special case, because the CDG has no root. The CCE mapping of this feature model is: \(D(\epsilon, D[A, B, C] \langle \epsilon, [A, B, C] Virtual 1 \langle D, E \rangle \rangle)\). In this case, \(A, B, C\) (shown in bold) is the root mandatory feature, and it must be selected. In order to select it, the right alternative of choice \(D\) (the first \(D\) in the expression) is selected, which is feature
D. Because D and E are alternatives of Virtual 1, selection of D means unselecting of E. Therefore E is a dead feature.

5.2.3 Void Feature Model

This operation takes a feature model as input and returns a value determining whether such feature model is void or not. An example of a void feature model is shown in Figure 5.4. There are two criteria to determine that this is a void feature model.

1. From the CDG we can find that the root mandatory feature A,B,C is a dead feature. We know that the root mandatory feature must be included in all products belonging to this feature model. A feature model whose root mandatory feature is dead is void.

2. The translated CDG actually is not valid. As mentioned in Section 3.5, in a valid
CDG there is no vertex connecting to another one by both a branch and a chain edge. Therefore if a CDG translated from a feature model is not valid, this feature model is void.

5.2.4 Redundancy

A feature model contains redundancies when some semantic information is modeled in multiple ways [25]. Figure 5.5 shows 3 examples of redundancies, and their corresponding CDGs. For the CDG of (b), it is clear that there is two edges connecting A, C and B.

One is actually from A, and the other is from C. A and C form a group feature, and it only needs one edge to B. The two edges indicate the redundancy. So the edge from C to B is redundant. In the feature model of (b), the Requires relationship between B and C is redundant.

However, in case (a), there aren’t multiple edges connecting two features. Here it is the
fact that the subgraph $B \leftarrow \text{Virtual 2} \rightarrow C$ is isomorphic to $B \leftarrow \text{Virtual 1} \rightarrow C$ that reveals the redundancy.
Chapter 6: Conclusion

In this thesis, I have proposed two representations for variational software to improve the reasoning about merge conflicts and feature model properties. I use the choice edit model to represent changes introduced by merge commits to files. A merge commit merges two parent commit, and the merge may cause conflicts. Analyzing the choice edit model representations of merge commits, conflicts caused by move operations are detected. An experiment of 50 repositories in Git shows that about 27% merge commits have conflicts, and among these conflicts, about 3% are conflicts caused by move operation.

The thesis has further presented translating feature models into choice calculus expressions and Choice Dependency Graphs. Analyzing the translated choice dependency graphs, dead features and redundancies in feature models can be detected. Moreover, it is also used to determine whether feature models are void, and whether products belongs to feature models. Choice Dependency Graph provides much more obvious structure than feature model diagrams. It supports the analysis operations of feature models more effectively.
Bibliography


[23] Pablo Trinidad, David Benavides, and Antonio Ruiz-corts. Isolated features detection in feature models.


[27] Hai Wang, Yuan Fang Li, Jing Sun, Hongyu Zhang, and Jeff Pan. A semantic web approach to feature modeling and verification. In *In Workshop on Semantic Web Enabled Software Engineering (SWESE’05, 2005.*/

Appendix A: Inference Rules to translate Edit Sequence to Choice Edit Model

The name in the parenthesis after each rule corresponds to the rule’s name in the Coq program.

\[
\begin{align*}
\text{[fordim A undo]} & \vdash (\mathit{vs}, (A, \text{True}) \cup \delta) \rightarrow (\mathit{vs}, (A, \text{False}) \cup \delta) \quad \text{(aundo)} \\
\text{[fordim A redo]} & \vdash (\mathit{vs}, (A, \text{False}) \cup \delta) \rightarrow (\mathit{vs}, (A, \text{True}) \cup \delta) \quad \text{(aredo)} \\
\text{[fordim A insert 0 \, b]} & \vdash (a, \delta) \rightarrow (A\langle\varepsilon, b\rangle a, \{(A, \text{True})\} \cup \delta) \quad \text{(inlit0)} \\
\text{[fordim A insert 1 \, b]} & \vdash (a, \delta) \rightarrow (aA\langle\varepsilon, b\rangle, \{(A, \text{True})\} \cup \delta) \quad \text{(inlit1)} \\
\text{[fordim A delete 0 \, b]} & \vdash (a, \delta) \rightarrow (A\langle a, \varepsilon \rangle, \{(A, \text{True})\} \cup \delta) \quad \text{(delit0)} \\
\text{[fordim A change 0 \, b]} & \vdash (a, \delta) \rightarrow (A\langle a, b \rangle, \{(A, \text{True})\} \cup \delta) \quad \text{(chlit0)}
\end{align*}
\]

\[
\begin{align*}
\text{[fordim A insert n \, a]} & \vdash (\mathit{vs}, \delta) \rightarrow (\mathit{vs}', \delta') \\
\text{[fordim A insert (succn) \, a]} & \vdash (\mathit{xvs}, \delta) \rightarrow (\mathit{xvs}', \delta') \\
\text{[fordim A change n \, a]} & \vdash (\mathit{vs}, \delta) \rightarrow (\mathit{vs}', \delta') \\
\text{[fordim A change (succn) \, a]} & \vdash (\mathit{xvs}, \delta) \rightarrow (\mathit{xvs}', \delta') 
\end{align*}
\]
\[
\text{[fordim A delete n]} \vdash (vs, \delta) \rightarrow (vs', \delta')
\] (jolitde)

\[
\text{[fordim A delete (succn)]} \vdash (xvs, \delta) \rightarrow (xvs', \delta')
\]

\[
\text{[fordim A op]} \vdash (vs1, \delta1 \cup \delta12) \rightarrow (vs1', \delta1' \cup \delta12')
\] (joinle)

\[
\text{[fordim A op]} \vdash (vs1vs2, \delta1 \cup \delta12 \cup \delta2) \rightarrow (vs1'vs2, \delta1' \cup \delta12' \cup \delta2)
\]

\[
\text{[fordim A insert n a]} \vdash (vs, \delta vs \cup \delta vsC) \rightarrow (vs', \delta vs' \cup \delta vsC')
\] (jorgin)

\[
\text{[fordim A insert (succn) a]} \vdash (Cvs, \delta vs \cup \delta vsC \cup \delta C) \rightarrow (Cvs', \delta vs' \cup \delta vsC' \cup \delta C')
\]

\[
\text{[fordim A change n a]} \vdash (vs, \delta vs \cup \delta vsC) \rightarrow (vs', \delta vs' \cup \delta vsC')
\] (jorgch)

\[
\text{[fordim A change (succn) a]} \vdash (Cvs, \delta vs \cup \delta vsC \cup \delta C) \rightarrow (Cvs', \delta vs' \cup \delta vsC' \cup \delta C')
\]

\[
\text{[fordim A delete n]} \vdash (vs, \delta vs \cup \delta vsC) \rightarrow (vs', \delta vs' \cup \delta vsC')
\] (jorgde)

\[
\text{[fordim A delete (succn)]} \vdash (Cvs, \delta vs \cup \delta vsC \cup \delta C) \rightarrow (Cvs', \delta vs' \cup \delta vsC' \cup \delta C')
\]

\[
\text{[fordim A op]} \vdash (vs1 \ (vs2 \ vs3), \delta) \rightarrow (vs', \delta')
\] (join3)

\[
\text{[fordim A op]} \vdash ((vs1 \ vs2) \ vs3, \delta) \rightarrow (vs', \delta')
\]
\[
es_1 \vdash (vs_1, \delta) \rightarrow (vs_2, \delta') \quad es_2 \vdash (vs_2, \delta') \rightarrow (vs_3, \delta'') \quad (\text{cones})
\]

\[
(es_1 + es_2) \vdash (vs_1, \delta) \rightarrow (vs_3, \delta'') \quad (\text{cholef})
\]

\[
\begin{align*}
\text{[fordim } A \ op]\vdash (vs_1, \delta) & \rightarrow (vs_1', \delta') \\
\text{[fordim } A \ op]\vdash (D\langle vs_1, vs_2 \rangle, \{(D, False)\} \cup \delta) & \rightarrow (D\langle vs_1', vs_2 \rangle, \{(D, False)\} \cup \delta')
\end{align*} \quad (\text{chorig})
\]

The rules \textit{joinlitin joinlitch} and \textit{joinlitde} describes that an edit applies to a variational string \( vs' \) to \( vs' \). If there exist one plain character \( x \) before \( vs \), then the same edit moves to the right by one position to change \( vs \) to \( vs' \). The result is \( vs' \) following \( x \).

The rules \textit{jorgin}, \textit{jorgch} and \textit{jorgde} describe that edits change \( vs \) following a choice \( C \), not a plain character. \( C \) has its own selections, and this environment may have intersection with \( vs \). Suppose the intersection is \( \delta_{vsC} \), and \( \delta C \) is the environment for \( C \) excluding \( \delta_{vsC} \). Then \( \delta C \cup \delta_{vsC} \) is the environment of \( C \). The situation is similar to \( vs \). For the whole string \( C vs \), it has environment \( \delta_{vs} \cup \delta_{vsC} \cup \delta C \).

The rule \textit{joinlef} describes that an edit changes \( vs_1 \) to \( vs_1' \), the another variational string \( vs_2 \) is added after \( vs_1 \), the edit changes this concatenated string to \( vs_1' vs_2 \). Because \( vs_2 \) may have environment of selection, which may have intersection with \( vs_1 \). It applies the similar idea to represent environment as rules \textit{jorgin}, \textit{jorgch} and \textit{jorgde}.

From the rule we can observe that \( \delta 2 \) doesn’t change, because \( vs_2 \) is independent with \( vs_1 \). An edit to \( vs_1 \) doesn’t affect the selections of \( vs_2 \).
Appendix B: Source code of the Coq Program to translate Edit Sequences to Choice Edit Model

B.1 Define types

(*The first 5 lines imports libraries for integer, boolean value, and list. The scopes of integer and boolean value are added.*)

Require Import ZArith.
Require Import Bool.
Require Import List.
Open Scope Z.
Open Scope bool.

(*Define the data type of a list.*)
Inductive list (T: Type) : Type :=
| nil : list T
| cons : T -> list T -> list T
| conc : list T -> list T -> list T.

(*Define a pair of two variables.*)
Inductive prod (A B : Type) : Type :=
pair : A -> B -> prod A B.

(*Define data type of dimension in choices.*)
Inductive Dim : Type :=
| A : Dim
| B : Dim
| C : Dim.

(*Define the environment of selections. One selection is a pair of dimension
and boolean value. The environment of selections is Dec, which means ‘Decision’.*)
Definition Sel : Type := prod Dim bool.
Definition Dec : Type := list (Sel).

(*Define the data type of position, and content an edit changes with the position.*)
Definition Pos := Z.

Inductive Tup (T: Type) : Type :=
| tup : Pos -> T -> (insert).

(*Define the data type of operations in edits*)
Inductive Op (T: Type) : Type :=
| insert : (Tup T) -> Op T
| delete : Pos -> Op T
\begin{verbatim}
(\textbf{Define the data type of an edit, and an edit sequence})

\textbf{Inductive Edit (T: Type)} : Type :=
| fordim : Dim \to (Op T) \to (Edit T).

\textbf{Definition ES (T: Type)} : Type := list (Edit T).

(\textbf{Define the choice edit model. It can be empty, a single plain character, a choice, and concatenation of multiple choice edit models *})

\textbf{Inductive CET (T: Type)} : Type :=
| lit : T \to (CET T)
| choice : Dim \to (CET T) \to (CET T) \to (CET T)
| join : (CET T) \to (CET T) \to (CET T)
| empty : CET T.

(\textbf{Define a choice edit model with its environment of selections*}) \textbf{Definition}
\textbf{CER (T: Type)} : Type := prod (CET T) Dec.
\end{verbatim}
B.2 Rules to translate edit sequences to the choice edit model

Inductive EStoCER (T: Type) : (CER T) -> (ES T) -> (CER T) -> Prop := 
  | aundo : forall (A:Dim) cet dec, EStoCER T 
    (pair (CET T) Dec cet (cons Sel (pair Dim bool A true) dec)) 
    (cons (Edit T) (fordim T A (undo T))(nil (Edit T))) 
    (pair (CET T) Dec cet (cons Sel (pair Dim bool A false) dec))

I will give a detailed explanation of the `undo` rule, then notations in other rules are similar.

EStoCER is a proposition, which has type Prop. It describes that an original choice edit model, which has type CER T, is applied with an edit sequence (ES T), and changes to a new choice edit model.

This rule says, for all dimension A, the original choice edit model with its selections is ((pair (CET T) Dec cet (cons Sel (pair Dim bool A true) dec)). pair is a pair of type CET T and Dec. cet is the original choice edit model. (cons Sel (pair Dim bool A true) dec)) describes a list which has type Sel, and this list adds an pair of dimension and boolean value (pair Dim bool A true) which represents (A,True) to another list dec.

The original choice edit model is applied with an edit sequence (cons (Edit T) (fordim T A (undo T))(nil (Edit T))) which represents a list which has one edit `fordim A undo`. (nil (Edit T)) represents an empty list of edits. The result choice
edit model is (pair (CET T) Dec cet (cons Sel (pair Dim bool A false) dec)). Comparing it with the original choice edit model, the cet are the same, but (A, True) in selections changes to (A, False).

In this rule, every data constructor needs type declaration, which makes it hard to read. So in the following rules, I removed all type declarations to improve the readability of this program. In each rule, there are 3 lines after EStoCER, the first line is the original choice edit model with its selection, the second line is a edit sequence, and the third line is the result choice edit model.

| aredo : forall (A:Dim) cet dec, EStoCER  |
| (pair cet (cons (A false) dec))         |
| (cons (fordim A (redo))nil)             |
| (pair cet (cons (A true) dec))          |

| inlit0 : forall (a:T) (b:T) (A:Dim) dec, EStoCER |
| (pair (lit a) dec)                               |
| (cons (fordim A(insert (tup 0 b)))(nil))         |
| (pair (join (choice A empty (lit b)) (lit a)) (cons (pair A true)dec)) |

Here exits a new notations: tup 0 b, which means at position 0, the edit sequence adds a character b.

| inlit1 : forall (a:T) (b:T) (A:Dim) dec, EStoCER |
| (pair (lit a) dec)                               |
| (cons (fordim A(insert (tup 1 b)))(nil))         |
| (pair (join (lit a) (choice A empty (lit b))) (cons (pair A true)dec)) |

| delit0 : forall (a:T) (b:T) (A:Dim) dec, EStoCER |

(pair (lit a) dec)
(cons (fordim A(delete 0)) (nil))
(pair (choice A (lit a) empty) (cons (pair A true) dec))

<table>
<thead>
<tr>
<th>chlit0 : forall (a:T) (b:T) (A:Dim) dec, EStoCER</th>
</tr>
</thead>
<tbody>
<tr>
<td>(pair (lit a) dec)</td>
</tr>
<tr>
<td>(cons (fordim A(insert (tup 0 b))) (nil))</td>
</tr>
<tr>
<td>(pair (choice A (lit a) (lit b)) (cons (pair A true) dec))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>jolitin : forall (a:T) (A:Dim) cet cet' dec dec' (n:Pos) (x:T), EStoCER</th>
</tr>
</thead>
<tbody>
<tr>
<td>(pair cet dec)</td>
</tr>
<tr>
<td>(cons (fordim A(insert (tup n a))) (nil))</td>
</tr>
<tr>
<td>(pair cet' dec') -&gt; EStoCER</td>
</tr>
<tr>
<td>(pair (join (lit x) cet) dec)</td>
</tr>
<tr>
<td>(cons (fordim A(insert (tup (n+1) a))) (nil))</td>
</tr>
<tr>
<td>(pair (join (lit x) cet') dec')</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>jolitch : forall (a:T) (A:Dim) cet cet' dec dec' (n:Pos) (x:T), EStoCER</th>
</tr>
</thead>
<tbody>
<tr>
<td>(pair cet dec)</td>
</tr>
<tr>
<td>(cons (fordim A(insert (tup n a))) (nil))</td>
</tr>
<tr>
<td>(pair cet' dec') -&gt; EStoCER</td>
</tr>
<tr>
<td>(pair (join (lit x) cet) dec)</td>
</tr>
<tr>
<td>(cons (fordim A(insert (tup (n+1) a))) (nil))</td>
</tr>
<tr>
<td>(pair (join (lit x) cet') dec')</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>jolitde : forall (a:T) (A:Dim) cet cet' dec dec' (n:Pos) (x:T), EStoCER</th>
</tr>
</thead>
<tbody>
<tr>
<td>(pair cet dec)</td>
</tr>
</tbody>
</table>
This rule is a special case of rule joinlef. When cet1 is a plain character in joinlef, it is equivalent to jochle. Because a plain character has no environment, we don’t need to consider the intersections. This simplify the proof process for this special case.
Here exits a new notation $\text{conc}$, which means concatenating two selections.

\[
\text{\| jorgin : forall } (a:T) (A:\text{Dim}) \text{ cet1 cet2 cet2'} \text{ dec2 dec2'} \text{ dec12 dec12'} \text{ dec1} (n:\text{Pos}), \text{ EStoCER}
\]

\[
\text{(pair cet2 (conc dec2 dec12))}
\]

\[
\text{(cons (fordim A (insert (tup n a))) (nil))}
\]

\[
\text{(pair cet2' (conc dec2' dec12')) \rightarrow EStoCER}
\]

\[
\text{(pair (join cet1 cet2) (conc dec2 (conc dec12 dec1)))}
\]

\[
\text{(cons (fordim A (insert (tup (n+1) a))) (nil))}
\]

\[
\text{(pair (join cet1 cet2') (conc dec2' (conc dec12' dec1)))}
\]

\[
\text{\| jorgch : forall } (a:T) (A:\text{Dim}) \text{ cet1 cet2 cet2'} \text{ dec2 dec2'} \text{ dec12 dec12'} \text{ dec1} (n:\text{Pos}), \text{ EStoCER}
\]

\[
\text{(pair cet2 (conc dec2 dec12))}
\]

\[
\text{(cons (fordim A (insert (tup n a))) (nil))}
\]

\[
\text{(pair cet2' (conc dec2' dec12')) \rightarrow EStoCER}
\]

\[
\text{(pair (join cet1 cet2) (conc dec2 (conc dec12 dec1)))}
\]

\[
\text{(cons (fordim A (insert (fordim (n+1) a))) (nil))}
\]

\[
\text{(pair (join cet1 cet2') (conc dec2' (conc dec12' dec1)))}
\]

\[
\text{\| jorgde : forall } (a:T) (A:\text{Dim}) \text{ cet1 cet2 cet2'} \text{ dec2 dec2'} \text{ dec12 dec12'} \text{ dec1} (n:\text{Pos}), \text{ EStoCER}
\]

\[
\text{(pair cet2 (conc dec2 dec12))}
\]

\[
\text{(cons (fordim A (delete T n )) (nil))}
\]

\[
\text{(pair cet2' (conc dec2' dec12')) \rightarrow EStoCER}
\]
(pair (join cet1 cet2) (conc dec2 (conc dec12 dec1)))
(cons (fordim A (delete T (n+1))) (nil))
(pair (join cet1 cet2') (conc dec2' (conc dec12' dec1)))

| jrglitch :forall (A:Dim) cet cetx dec dec' (x:T) (a:T), EStoCER
  (pair (lit x) dec)
  (cons (fordim A (change (tup 0 a)) nil) (nil))
  (pair cetx dec') → EStoCER
  (pair (join cet (lit x)) dec)
  (cons (fordim A (change (tup 1 a)) nil) (nil))
  (pair (join cet cetx) dec')

This rule is a special case of rule jorgch. When cet2 is a plain character in jorgch, it is equivalent to jochle. A plain character has no selections, and proof for this special case is simple.

| join3 :forall (A:Dim) cet1 cet2 cet3 cet dec dec' op, EStoCER
  (pair (join cet1 (join cet2 cet3)) dec)
  (cons (fordim A op) (nil))
  (pair cet dec') → EStoCER
  (pair (join (join cet1 cet2) cet3) dec)
  (cons (fordim A op) (nil))
  (pair cet dec')

| cones : forall (e: Edit T) (es: ES T) cet1 cet2 cet3 dec1 dec2 dec3, EStoCER
  (pair cet1 dec1)
(cons e (nil))
(pair cet2 dec2) -> EStoCER
(pair cet2 dec2)
es
(pair cet3 dec3) -> EStoCER
(pair cet1 dec1)
(cons e es)
(pair cet3 dec3)
| cholef : forall (D:Dim) (A: Dim) cet1 cet1' cet2 dec dec' op, EStoCER
  (pair cet1 dec)
  (cons (fordim A op) (nil))
  (pair cet1' dec') -> EStoCER
  (pair (choice D cet1 cet2) (cons (pair D false) dec))
  (cons (fordim A op) (nil))
  (pair (choice D cet1' cet2) (cons (pair D false) dec'))

| chorig : forall (D:Dim) (A: Dim) cet1 cet2 cet2' dec dec' op, EStoCER
  (pair cet2 dec)
  (cons (fordim A op) (nil (Edit T)))
  (pair cet2' dec) -> EStoCER
  (pair (choice D cet1 cet2) (cons (pair D true) dec))
  (cons (fordim A op) (nil (Edit T)))
  (pair (choice cet1 cet2') (cons (pair D true) dec')).
B.3 Proof of theorems

The first theorem says for all dimension $X$, a choice edit model $\text{cet1}$ with its selections $(A, True) \cup dec1$ is applied by an undo edit. This edit does not change the choice edit model, but change $(A, True)$ to $(A, False)$ in selections.

The source code of this theorem is:

```coq
Theorem undogen : forall (X : Dim) (dec1 : Dec) (T: Type) (cet1 : CET T),
EStoCER T
(pair (CET T) Dec cet1 (cons Sel (pair Dim bool X true) dec1))
(cons (Edit T) (fordim T X (undo T)) (nil (Edit T)))
(pair (CET T) Dec (cet1) (cons Sel (pair Dim bool X false) dec1)).
```

To make it easy to read, I remove the type declarations:

```coq
Theorem undogen : forall (X : Dim) (dec1 : Dec) (T: Type) (cet1 : CET T),
EStoCER T
(pair cet1 (cons (pair X true) dec1))
(cons (fordim X undo) nil)
(pair (cet1) (cons (pair X false) dec1)).
```

The proof is simple for this theorem, because it is the same as rule $\text{undo}$:

```coq
Proof.
intros.
apply undo with (A := X) (cet := cet1) (dec := dec1).
Qed.
```

The second theorem is: for all dimension $X$, a choice edit model $\text{cet1}$ with its selec-
tions \((A, False) \cup dec1\) is applied by an redo edit. This edit does not change the choice edit model, but change \((A, False)\) to \((A, True)\) in selections. The Coq program (without type declaration) is:

\[
\text{Theorem redogen} : \forall (X : \text{Dim}) (dec1 : \text{Dec}) (T : \text{Type}) (cet1 : \text{CET T}), \\
\text{EStoCER T} \\
\quad \left(\text{pair cet1 (cons (pair X false) dec1)}\right) \\
\quad \left(\text{cons (fordim X undo) nil}\right) \\
\quad \left(\text{pair (cet1) (cons (pair X true) dec1)}\right).
\]

Proof.

intros.

apply aredo with \((A := X) (cet := cet1) (dec := dec1)\).

Qed.

The third theorem presents for all dimension \(X, Y\), plain character \(x, y, u, v\), and environment of selections \(\delta\):

\[
[\text{fordim} X \text{ change } 0 u, \text{fordim} Y \text{ change } 1 v] \vdash (xy, \delta) = \\
[\text{fordim} Y \text{ change } 1 v, \text{fordim} X \text{ change } 0 u] \vdash (xy, \delta)
\]

Edit \(X\) and \(Y\) are independent, because the parts they changed have no overlap. \((X\) changes \(x\), and \(Y\) changes \(y\)).

This theorem are proved by two steps. First, I prove that the sequence \(X,Y\) changes \(ab\) into \(C_1\), and \(Y,X\) changes \(ab\) into \(C_2\). If \(C_1\) and \(C_2\) are equivalent, then the theorem is proved. Theorem \textit{reorder1} proves that \([\text{fordim} X \text{ change } 0 u, \text{fordim} Y \text{ change } 1 v] \vdash (xy, \delta) = C_1\)
Lemma reorder1 :forall (X: Dim) (Y:Dim)(T: Type) (x y u v : T), EStoCER T
(pair (join (lit x) (lit y))nil)
(cons (fordim X (change (tup 0 u))) (cons(fordim Y (change (tup 1 v)))) nil))
((pair (join (choice X (lit x) (lit u)) (choice Y (lit y) (lit v)))) (cons (pair Y true) (cons (pair X true) nil)))).
The first line is an original string xy. It is edited by edits X and Y. The result is a choice expression: \(X(x,u)Y(y,v)\). The proof is:
Proof.
intros.
apply cones with (e := fordim X (change (tup 0 u)))
( (es := (cons (fordim Y (change (tup 1 v)))) (nil)))
(cet1 := join (lit x) (lit y))
(cet2 := join (choice X (lit x) (lit u)) (lit y))
(cet3 := join (choice X (lit x) (lit u)) (choice Y (lit y) (lit v))))
(dec1 := nil) (dec2 := cons (pair X true) nil)
(dec3 := cons (pair Y true) (cons (pair X true) nil))).
Rule cones takes the first edit (e) in the edit sequence. This edit changes xy to \(X(x,u)Y(y,v)\).
The remaining edit sequence (es) changes \(X(x,u)Y\) to \(X(x,u)Y(y,v)\).
apply jochle with (A := X) (cet := (lit y))
( (cetx := choice X (lit x) (lit u))
(dec := nil) (dec' := (cons (pair X true) nil)))
(op := change (tup 0 u)).

Rule jochle separates $X \langle x, u \rangle y$ to $X \langle x, u \rangle$ and $y$. If an edit can change $x$ to $u$, then it can change $xy$ to $uy$.

apply chlit0 with (a := x) (b := u) (A := X) (dec := nil).

Rule chlit0 proves that edit $X$ changes $x$ to $X \langle x, u \rangle$.

apply jrglitch with (A := Y) (a:=v) (cet := (choice X (lit x) (lit u)))
(cetx := (choice Y (lit y) (lit v)))
(dec := (cons (pair X true) nil))
(dec' := (cons (pair Y true) (cons (pair X true) nil))).

This rule proves that if an edit $Y$ can change $y$ to $v$, then it can changes $X \langle x, u \rangle y$ to $X \langle x, u \rangle Y \langle y, v \rangle$.

apply chlit0 with (a := y) (b := v) (A := Y) (dec := (cons (pair X true) nil)).

Qed.

The last rule proves that edit $Y$ changes $y$ to $Y \langle y, v \rangle$.

The theorem reorder2 is:

Fact reorder2 : forall (X: Dim) (Y:Dim) (T: Type) (x y u v : T), EStoCER T
(pair (join (lit x) (lit y))nil)
(cons (fordim Y (change (tup 1 v))) (cons(fordim X (change (tup 0 u)))) nil))

It applies rules cones, jolitch, chlit0, jochle and chlit0.

Comparing the result CEM in reorder1 and reorder2 we can find they are the same.
Appendix C: 50 most well-known repositories in Git

apple/swift; FreeCodeCamp/FreeCodeCamp; python/cpython; twbs/bootstrap;
torvalds/linux; vhf/free-programming-books; angular/angualr.js; nodejs/node;
mbostock/d3; jquery/jquery; FortAwesome/Font-Awesome; h5bp/html5-boilerplate;
rails/rails; impress/impress.js; meteor/meteor; github/gitignore;
robbyrussell/on0my-zsh; Homebrew/homebrew; adobe/bracket; jashkenas/backbone;
moment/moment; zurb/foundation-sites; hakimel/reveal.js; docker/docker;
jeekyll/jekyll; mrdoob/three.js; facebook/react; AFNetworking/ AFNetworking;
socketio/socket.io; resume/resume.github.com; tiimgreen/github-cheat-sheet;
gitlabhq/gitlabhq; laravel/laravel; Modernizr/Modernizr; kennethreitz/requests;
TryGhost/Ghost; driftyco/inoic; jashkenas/underscore; discourse/discourse;
nnnick/Chart.js; ariya/phantomjs; django/django; emberjs/ember.js;
vinta/awesome-python; mitsuhiko/flask; atom/atom; papers-we-love/papers-we-love;
antires/redis; gulpjs/gulp;