ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bachelor Thesis

Tree-based Version Control in Envision

Balz Guenat

Supervisors:
Dimitar Asenov
Prof. Dr. Peter Müller

Chair of Programming Methodology
Department of Computer Science
ETH Zürich

1. August 2015

Abstract

Envision is a language-agnostic open-source IDE that abstracts and visualizes the syntactic structure of software projects. Its goal is to minimize syntactic clutter and convey as much information as possible visually to reduce the time developers spend reading code, making them more productive. In Envision, program code is internally represented and stored as an abstract syntax tree and it features a built-in version control system (VCS) that works on the basis of nodes in the abstract syntax tree instead of text lines like traditional systems.

In this thesis we continue the development of Envision's VCS, extending it with a formal model and proof of correctness for its *diff* algorithm. Furthermore we present a new *merge* algorithm, systematically designed with modularity and verifiability in mind. We introduce the abstract models behind its implementation and provide an argument for the correctness of the algorithm.

The resulting system exhibits a number of advantages over traditional text-based version control systems such as semantic classification of changes in the code, improved conflict detection and the ability to configure and extend how conflicts are resolved automatically. Additionally, the precise descriptions of the algorithms and the arguments for their correctness give confidence to future developers of Envision and will ultimately reassure users operating these tools.

Contents

List of Algorithms

Acknowledgments

I would like to thank Prof. Dr. Peter Müller and the entire Chair of Programming Methodology at ETH Zürich for giving me the opportunity to contribute to the development of Envision for my bachelor thesis. I learned a great deal during this project.

Thanks go to Alex Summers who assisted me in finding a way to structure my proof clearly and effectively. This likely saved me a a lot of frustrating work.

Further, I wish to thank Martin Otth whose input helped tremendously in getting familiar with the existing systems and the code base. He helped me get comfortable in the tricky business that is tree-based version control.

I am hugely thankful for the assistance and guidance I received from Dimitar Asenov with whom I have worked very closely throughout this project. His suggestions and feedback were key to the accomplished design and his support was invaluable for the implementation work.

Finally, I want to thank my friends and family for all their help and support – technical or otherwise.

Chapter 1

Introduction

1.1 Motivation

Envision[1] is an IDE for object-oriented programming languages [1]. It features a graphical code editor that visualizes the syntax structures of software projects with the goal of abstracting from the syntactic rules of a language. Internally, the projects are represented as abstract syntax trees. Envision is an open-source project and is being developed mainly in the context of Dimitar Asenov's PhD work at ETH Zurich.

Traditional code editors and IDEs are text editors at their core. Similarly, existing version control systems like Git[2] and SVN[3] and their employed difference and synchronization algorithms (often just called *diff* and *merge*) work on the basis of text files and lines, ignoring the highly structured nature of program code. This often results in *diff* outputs lacking context, making it harder for the user to make sense of the presented information. More severe issues exist with *merge* algorithms which can introduce bugs by automatically merging pieces of code that are related but not directly adjacent. In cases where conflicts are detected, the resulting output is at times more confusing than helpful.

1.2 Related work

There already exist different approaches to merging documents that are structured as trees. Lindholm [5] proposed a way to merge XML documents modeled as ordered trees of nodes with unique identifiers. Unlike in Envision

[1] `dimitar-asenov.github.io/Envision`
[2] `git-scm.com`
[3] `subversion.apache.org`

however, the relationship of a child to its parent is not labeled, thus the approach cannot guarantee the syntactic validity of the produced AST.

Westfechtel [7] presented a language-agnostic merge algorithm for ASTs that guarantees context-free correctness of the merged tree. This algorithm however does not take move operations into account and focuses on the detection of conflicts, not their resolution.

The development team of *semanticmerge*[4] and *Plastic SCM*[5], a text-based VCS, recently published an article describing some of their efforts to improve their text-based *diff* tool[6]. Similar to Envision's VCS, their tool tries to identify moves of entities and classifies changes as additions, deletions, moves or updates. This shows that the desire for more concise and understandable diffs has been recognized by the industry and solutions similar to that of Envision are being developed.

1.3 Existing Envision VCS and goals

In context of his master thesis [6], Martin Otth designed and implemented the core of a version control system for Envision using Git as a back end. Through development and testing of this system, we gained a lot of understanding about version control of trees and revealed a few correctness issues in the original design. One such issue was that the design relied on an assumption about the encoding of ASTs in relation to `git diff` that turned out to be false. This could result in the *AST Diff* algorithm producing incomplete results. Due to the large scope of Otth's project, time was insufficient to explore these issues in greater depth.

The goal of this work is to investigate two core modules of that system – namely the *AST Diff* and *AST Merge* algorithms – in terms of correctness and to make changes where appropriate. The first step to achieving this goal is to formally define what the requirements for these algorithms are. This includes a formal model for describing the differences between two abstract syntax trees as they are used by Envision. Once we have precisely defined the requirements we systematically check and revise the algorithms until we can show that they produce the expected output.

Once this task is completed, another goal is to find a way to allow for customizable behavior of the *AST Merge* algorithm by plug-in-like components. This would not only make it easier for developers to extend the Envision ver-

[4] `semanticmerge.com`

[5] `plasticscm.com`

[6] `codicesoftware.blogspot.com/2015/07/towards-semantic-version-control.html`

sion control system with new and better default behavior but it would also allow users to adjust these modules to work best for their needs, tailoring them to a particular project or workflow.

1.4 Report structure

The following chapter 2 will introduce some fundamental concepts used throughout this report and define a formal model of what a *diff* of two abstract syntax trees contains. In chapter 3 we present our *AST Diff* algorithm and show that it complies with the formal model. In chapter 4 we present a redesigned *AST Merge* algorithm and two modules that work as plug-in components. We then give an argument for the correctness of the *AST Merge* algorithm and one of the components in chapter 5. An evaluation in chapter 6 highlights the advantages of the presented system over traditional text-based version control systems. Chapter 7 explains some technical aspects of the implementation that are less evident. In chapter 8 we lay out possible work for the future before drawing our conclusions in chapter 9.

Chapter 2

Fundamental Concepts

This chapter serves to introduce some core concepts and terms used throughout this report. Apart from some nomenclature, we present how ASTs and AST nodes are represented and encoded by Envision. These definitions will be the foundation on which we build in the following chapters.

2.1 Nomenclature

Throughout this report, we are often going to use the following terms. These definitions apply unless explicitly stated otherwise.

- *AST* is short for abstract syntax tree. Sometimes just called *tree*.
- An *AST node* or just *node* is a single node of an AST.
- *line of text* or simply *line* refers to a line in a file, delimited by newline characters and identified by its content (not its line number).
- *node line* refers to the concrete, one-line, textual representation of a given AST node in a file.
- *AST Diff* refers to the tree-based diff algorithm presented in chapter 3.
- *AST Merge* refers to the tree-based merge algorithm presented in chapter 4.
- The *encoding* of an AST is the set of node lines that store the nodes of the AST (details in section 2.3).
- The *line diff* of encodings E_A, E_B is the union of the outputs of `git diff` over all files of E_A and E_B. Also denoted lineDiff(E_A, E_B).

2.2 Abstract Syntax Trees (ASTs)

In Envision, ASTs consist of AST nodes which are linked together. An AST node consists of the following:

- *Label*: The label specifies the relationship with the parent node (e.g. then and otherwise for the bodies of an if-statement). Nodes that are elements of lists are labeled with an integer specifying their index.
- *Type*: The type of the node (e.g. Method or Expression).
- *ID*: UUID[1] of the node. The ID defines node identity.
- *ParentId*: UUID of the parent node. This is `{0000...00}` for the root of the AST.
- *Parent*: A pointer to the parent node. This is NULL for the root of the AST.
- *Value*: Only leaf nodes may have a value.

2.2.1 Definition of Valid Form

We consider an AST T to be of *valid form* if and only if it satisfies both of the following properties.

1. Node IDs are unique: $\forall n, n' \in T : n.id = n'.id \rightarrow n = n'$
2. No two children of a node have the same label: $\forall n, n' \in T : n.parent = n'.parent \wedge n.label = n'.label \rightarrow n = n'$

2.3 Encoding of ASTs and AST nodes

We encode an AST node as a node line. A node line is a single line of text containing all members of the node except for the parent pointer. The exact format is shown in figure 2.1. Values are stored in the form `.T_v` where `T` specifies the primitive type of the value and `v` is the value. `T` is `S` for strings, `I` for integers and `F` for floats. Newline characters '`\n`' are escaped in `v`.

The encoding of an AST is the set of node lines encoding its nodes. We divide the AST up into multiple components called *persistent units*. Each

[1]Universally Unique Identifier. See RFC 4122: `www.ietf.org/rfc/rfc4122`

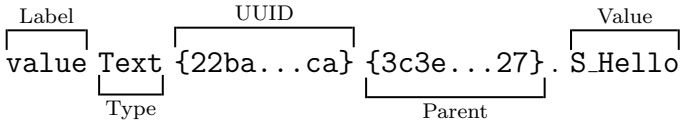


Figure 2.1: Example of Envision's node encoding

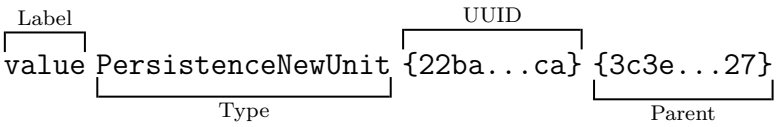


Figure 2.2: Example of persistent unit linking

persistent unit corresponds to one file in which we store the node lines of all nodes belonging to that persistent unit.

To indicate that a particular node is stored in a different persistent unit, instead of a line encoding the node, a line with the special type `PersistenceNewUnit` is stored. The actual encoding of the node can then be found in a file named after the linked node's ID. Figure 2.2 shows an example where the node with the ID `22ba...ca` is stored in a different file named `22ba...ca`. This file would then contain the line from figure 2.1 and all descendants of that node (or further link to the persistent units containing them).

In an encoding of a valid AST all node lines are stored on exactly one line of text and the ID of each node line and the node line itself is unique. The only exception to this are lines used for persistent unit linking i.e. lines with type `PersistenceNewUnit` which have the same ID as the node they refer to. For all considerations in this report, such lines are ignored and treated as if they did not exist.

2.4 Definition of Changes

In this section, we introduce a formal model that defines precisely the set of differences between two ASTs. In practice, when comparing two ASTs most often one of the trees will represent a chronologically later version of the other and the differences will describe changes that have been made by the user. For this reason, because of convention and to help intuition, we shall use terms that only make sense in the context of chronological order while keeping in mind that the principles apply regardless of time or order.

We introduce the notion of a change object that describes how a single

node changes from one version of the AST T_A to another version T_B. A change object is a tuple of the form

$$change :: (id, nodeA, nodeB, kind, labelFlag, typeFlag, valueFlag) \quad (2.1)$$

where id is a node ID, $nodeA$ and $nodeB$ are pointers to nodes (or NULL), $kind \in \{\text{Insertion}, \text{Deletion}, \text{Move}, \text{Stationary}\}$ and $labelFlag, typeFlag, valueFlag \in \{\text{True}, \text{False}\}$. We define the predicate

$$\text{isFake}(change) :\Longleftrightarrow change.kind = \text{Stationary} \wedge$$

$$\neg(change.labelFlag \vee change.typeFlag \vee change.valueFlag)$$

It is possible that a node line exists is both encodings but is still included in the line diff. This leads to the creation of changes that only represent modifications in the encoding but not the AST. The isFake predicate will be used to filter such changes.

In addition to sets of nodes, ASTs can be interpreted as sets of IDs. We use the shorthand

$$id \in T :\Longleftrightarrow \exists node \in T : node.id = id$$

For two ASTs T_A, T_B and $id \in T_A \cup T_B$, we define the function

$$\text{ChangeMap}_{T_A,T_B} : id \mapsto change \text{ such that}$$

- $change.id = id$
- $change.nodeA = node \in T_A : node.id = id$ if such a node exists and NULL otherwise.
- $change.nodeB = node \in T_B : node.id = id$ if such a node exists and NULL otherwise.
- $change.kind = \text{Insertion} \leftrightarrow id \notin T_A \wedge id \in T_B$
- $change.kind = \text{Deletion} \leftrightarrow id \in T_A \wedge id \notin T_B$

This leaves cases where $id \in T_A \wedge id \in T_B$. Let $n_A \in T_A : n_A.id = id$ and let $n_B \in T_B : n_B.id = id$.

- $change.kind = \text{Move} \leftrightarrow id \in T_A \wedge id \in T_B \wedge n_A.parentId \neq n_B.parentId$
- $change.kind = \text{Stationary} \leftrightarrow id \in T_A \wedge id \in T_B \wedge n_A.parentId = n_B.parentId$

- $change.labelFlag = \mathsf{True} \leftrightarrow id \in T_A \wedge id \in T_B \wedge n_A.label \neq n_B.label$
- $change.typeFlag = \mathsf{True} \leftrightarrow id \in T_A \wedge id \in T_B \wedge n_A.type \neq n_B.type$
- $change.valueFlag = \mathsf{True} \leftrightarrow id \in T_A \wedge id \in T_B \wedge n_A.value \neq n_B.value$

From this function, we now define the set

$$\text{Changes}(T_A, T_B) := \{change \in \text{ChangeMap}_{T_A,T_B}(T_A \cup T_B) : \neg\text{isFake}(change)\}$$

By definition, for any ID, this set contains at most one *change* with $change.id = ID$. This set of changes describes the transformation of T_A into T_B. We aim to compute this set in the *AST Diff* algorithm as an intermediate result.

2.4.1 Child structure changes

Additionally to the set of changes between two trees, we want to identify nodes whose child structure changes. This would include nodes that gain or lose one or more children and nodes whose children change their label. To this end we define the set $\text{StructChanges}(T_A, T_B)$ as follows.

$$id \in \text{StructChanges}(T_A, T_B) :\Longleftrightarrow \exists change \in Changes(T_A, T_B) :$$

$$(change.kind \in \{\mathsf{Insertion}, \mathsf{Move}\} \wedge change.nodeB.parentId = id) \vee$$

$$(change.kind \in \{\mathsf{Deletion}, \mathsf{Move}\} \wedge change.nodeA.parentId = id) \vee$$

$$(change.labelFlag \wedge change.nodeA.parentId = id)$$

Again, we aim to compute this set in the *AST Diff* algorithm as an intermediate result.

2.4.2 Extension of change objects

We extend change objects with an additional *structFlag* which will carry the information of the StructChanges set resulting in a more unified representation of differences. Let $(a, \ldots, b) \oplus c = (a, \ldots, b, c)$. We define the set of these extended changes $\text{XChanges}(T_A, T_B)$ as follows.

$$change \in Changes(T_A, T_B) \Longrightarrow change \oplus structFlag \in XChanges(T_A, T_B)$$
$$\text{where } structFlag \equiv change.id \in \text{StructChanges}(T_A, T_B) \tag{2.2}$$

$$
\begin{gathered}
id \in StructChanges(T_A, T_B) \wedge \nexists change \in Changes(T_A, T_B) : change.id = id \\
\Longrightarrow xChange \in XChanges(T_A, T_B) \\
\text{where } xChange.id = id, \\
xChange.nodeA \in T_A \wedge xChange.nodeA.id = id, \\
xChange.nodeB \in T_B \wedge xChange.nodeB.id = id, \\
xChange.kind = \mathsf{Stationary}, xChange.structFlag = \mathsf{True}, \\
xChange.labelFlag = xChange.typeFlag = xChange.valueFlag = \mathsf{False}.
\end{gathered}
\tag{2.3}
$$

There are no other elements in XChanges(T_A, T_B).

We effectively extend the already existing changes with the appropriate flag and add new changes for IDs in StructChanges for which there is not yet a change to be extended. This set XChanges(T_A, T_B) shall be the output of the *AST Diff* algorithm.

Chapter 3

The *AST Diff* Algorithm

In section 2.4 we have precisely defined the set of differences between two ASTs as a set of change objects. In this chapter, we present our *AST Diff* algorithm for computing this set change objects and then provide a proof of its correctness. The presented algorithm is largely based on the one introduced by Otth [6]. It includes slight modifications that help prove its correctness.

When Otth designed and implemented the core of the version control system, Envision did not encode the parent ID in the node line explicitly but implicitly by the order and indentation depth of the node lines. Because of this, moving a node did not directly result in a difference in the respective node line. The assumption was that the node line would change location in the file and thus appear in the line diff. This assumption turned out to be false, as it is possible for a node line to stay at the same location and to keep its indentation depth the same as in the original version.

To correct this, we changed the encoding of nodes to explicitly store the parent. This has the consequence that location and indentation of node lines become irrelevant and moving a node now directly results in a different node line. This will be vital in proving the correctness of the *AST Diff* algorithm.

3.1 Formal interface of the *AST Diff* algorithm

Let T_A and T_B be the two ASTs whose differences shall be computed. The algorithm takes the following inputs

- The encoding E_A of T_A
- The encoding E_B of T_B

- The line diff of E_A and E_B

and returns

- The set XChanges(T_A, T_B)

3.2 Assumptions about the line diff

We assume the line diff of encodings E_A and E_B shows a line of text l as a *removal* if l is present in E_A at some location but not in E_B. Similarly, we assume the line diff shows a line l as an *addition* if l is in E_B but not in E_A. Further, we assume all lines shown as removals are present in E_A and all lines shown as additions are present in E_B. Formally, we assume the line diff gives us two sets lineAdditions and lineRemovals such that

$$E_B \supseteq \text{lineAdditions} \supseteq \{l \in E_B \text{ with } l \notin E_A\} = E_B - E_A \tag{3.1}$$

$$E_A \supseteq \text{lineRemovals} \supseteq \{l \in E_A \text{ with } l \notin E_B\} = E_A - E_B \tag{3.2}$$

Note that the `git diff` does not return these two separate sets but rather a single set of lines, each marked as either an addition or removal. We interpret these as follows.

$$\text{lineAdditions} = \{l.content : l \in \text{GitDiffLines} \wedge l.origin = \mathsf{LineAddition}\} \tag{3.3}$$

$$\text{lineRemovals} = \{l.content : l \in \text{GitDiffLines} \wedge l.origin = \mathsf{LineRemoval}\} \tag{3.4}$$

Furthermore we make the assumption that if a line that is in both encodings is included as an addition, it must also be included as a removal (and vice versa). That is, we assume that

$$line \in E_A \cap E_B : line \in \text{lineAdditions} \iff line \in \text{lineRemovals} \tag{3.5}$$

3.3 Correctness of the *AST Diff* algorithm

We show that our *AST Diff* algorithm indeed produces the output specified in section 3.1.

This section references the algorithms AstDiff (3.1), computeChanges (3.2), createChange (3.3) and computeStructChanges (3.4). Names written in `typewriter font` are referring to the respective entities in these algorithms. We refer to a specific line with L#. We add the line number as subscript when referring to an entity at a specific program state. For example, $\texttt{nodeA}_4$ refers to the variable `nodeA` in the state before execution of L4.

```
1: function astDiff(GitDiffLines)
2:     nodesA ← ∅
3:     nodesB ← ∅
4:     for all lines in GitDiffLines do
5:         node ← Node(line.content)
6:         if line.origin = LineAddition then
7:             nodesB.add(node)
8:         else if line.origin = LineDeletion then
9:             nodesA.add(node)
10:        end
11:    end
                                          ▷ Intermediate state 1
12:    changes ← computeChanges(nodesA, nodesB)
13:    structChanges ← computeStructChanges(changes)
                                          ▷ Intermediate state 3
14:    xChanges ← ∅
15:    for all changes do
16:        xChange ← change ⊕ structChanges.contains(change.id)
17:        xChanges.add(xChange)
18:    end
19:    for all ids in structChanges do
20:        if changes.find(id) = NULL then
21:            xChange.id ← id
22:            xChange.nodeA ← treeA.find(id)
23:            xChange.nodeB ← treeB.find(id)
24:            xChange.kind ← Stationary
25:            xChange.flags ← False
26:            xChange.structFlag ← True
27:            xChanges.add(xChange)
28:        end
29:    end
30:    return xChanges
31: end
```

Algorithm 3.1: The *AST Diff* algorithm

3.3.1 Intermediate state 1

In AstDiff we first create node objects according to the lines in the line diff. Since `git diff` returns a single set of lines instead of the two sets lineAdditions and lineRemovals, we have to decide from the line origin to which set a line belongs.

From examining the AstDiff algorithm (3.1) it is clear that at intermediate state 1, the following holds:

$$\begin{gathered}\forall l \in \text{GitDiffLines}: \\ l.origin = \mathsf{LineAddition} \leftrightarrow \textsc{Node}(l.content) \in \texttt{nodesB} \\ l.origin = \mathsf{LineRemoval} \leftrightarrow \textsc{Node}(l.content) \in \texttt{nodesA}\end{gathered}$$

Or in terms of the mentioned sets:

$$l \in \text{lineAdditions} \iff \textsc{Node}(l) \in \texttt{nodesB} \tag{3.6}$$

$$l \in \text{lineRemovals} \iff \textsc{Node}(l) \in \texttt{nodesA} \tag{3.7}$$

3.3.2 Intermediate state 2

We show that the following holds at intermediate state 2 in computeChanges:

$$\begin{gathered}node_B \in \texttt{onlyInB} \iff \\ node_B \in \texttt{nodesB} \land \nexists node_A \in \texttt{nodesA} : node_A.id = node_B.id\end{gathered} \tag{3.8}$$

Note that `nodesA` and `nodesB` are never modified. We do not need to specify a concrete state when referring to them.

Direction $\Longleftarrow$

$$\mathsf{L3} \implies \texttt{onlyInB}_4 = \texttt{nodesB} \implies node_B \in \texttt{onlyInB}_4$$

We show by contradiction that $node_B$ is never removed from `onlyInB`. We assume that at L9, we call

$$\textsc{remove}(\texttt{nodeB}_9, \texttt{onlyInB}) \text{ with } \texttt{nodeB}_9 = node_B$$

$\texttt{nodeB}_9$ is assigned on L5. This implies that on L5 of the same loop iteration

$$\textsc{find}(\texttt{nodeA.id}_5\texttt{, nodesB}) \text{ returns } node_B$$

$$\implies \texttt{nodeA.id}_5 = node_B.id \lightning \nexists node_A \in \texttt{nodesA} : node_A.id = node_B.id$$

A contradiction. Therefore, our assumption was wrong and $node_B$ is never removed from `onlyInB`.

```
 1: function computeChanges(nodesA, nodesB)
 2:     changes ← ∅
 3:     onlyInB ← nodesB
 4:     for all nodesA do
 5:         nodeB ← nodesB.find(nodeA.id)
 6:         change ← createChange(nodeA, nodeB)
 7:         changes.add(change)
 8:         if nodeB then
 9:             onlyInB.remove(nodeB)
10:         end
11:     end
                                              ▷ Intermediate state 2
12:     for all onlyInB do
13:         change ← createChange(NULL, nodeB)
14:         changes.add(change)
15:     end
16:     for all changes do
17:         if isFake(change) then
18:             changes.remove(change)
19:         end
20:     end
21:     return changes
22: end
```

Algorithm 3.2: computeChanges

Direction $\Longrightarrow$

$$\texttt{L3} \Longrightarrow \texttt{onlyInB}_4 = \texttt{nodesB}$$

No nodes are added to `onlyInB` and none are removed from `nodesB`, hence

$$\texttt{onlyInB} \subseteq \texttt{nodesB}$$

$$\Longrightarrow node_B \in \texttt{nodesB}$$

We now show by contradiction that

$$\nexists node_A \in \texttt{nodesA} : node_A.id = node_B.id$$

We assume

$$\exists node_A \in \texttt{nodesA} : node_A.id = node_B.id$$

The loop at L4 iterates through all nodes in `nodesA`, including $node_A$. We examine the iteration with

$$\texttt{nodeA}_5 = node_A$$

From our assumptions about encodings, we know node IDs are unique within their encoding. From that, it follows that node IDs are also unique within the sets lineAdditions and lineRemovals and further also within `nodesA` and `nodesB`. Specifically, $node_B$ is the only node in `nodesB` with its ID, so at L5

$$\texttt{nodeA.id}_5 = node_A.id = node_B.id$$

$$\Longrightarrow \textsc{find}(\texttt{nodeA.id}_5\texttt{, nodesB}) \text{ returns } node_B$$

$$\Longrightarrow \texttt{nodeB}_6 = node_B \neq \texttt{NULL}$$

Therefore in this iteration the condition at L8 is satisfied and at L9, we call

$$\textsc{remove}(\texttt{nodeB}_9, \texttt{onlyInB}) \text{ with } \texttt{nodeB}_9 = node_B$$

$$\Longrightarrow node_B \notin \texttt{onlyInB}$$

No node is added to `onlyInB` after L3 so at intermediate state 2, we get the contradiction

$$node_B \notin \texttt{onlyInB} \lightning node_B \in \texttt{onlyInB}$$

Therefore, our assumption was wrong and

$$\nexists node_A \in \texttt{nodesA} : node_A.id = node_B.id$$

holds true. □

3.3.3 Intermediate state 3

We show that at intermediate state 3

$$\texttt{changes} = \text{Changes}(T_A, T_B)$$

$$\texttt{structChanges} = \text{StructChanges}(T_A, T_B)$$

Direction $\texttt{changes} \supseteq \text{Changes}(T_A, T_B)$

We first show $\texttt{changes} \supseteq \text{Changes}(T_A, T_B)$. Let $change \in \text{Changes}(T_A, T_B)$ be an arbitrary change object and let $id = change.id$.

$$change \in \text{Changes}(T_A, T_B) \Longrightarrow \text{isFake}(change) = \mathsf{False}$$

We make case distinction on the kind of *change* and show for each case that $change \in \texttt{changes}$.

```
 1: function createChange(nodeA, nodeB)
 2:     if nodeA then
 3:         change.id ← nodeA.id
 4:         if nodeB then
 5:             if nodeA.parentId = nodeB.parentId then
 6:                 change.kind ← Stationary
 7:             else
 8:                 change.kind ← Move
 9:             end
10:         else
11:             change.kind ← Deletion
12:         end
13:     else
14:         change.id ← nodeB.id
15:         change.kind ← Insertion
16:     end
17:     change.flags ← computeFlags(nodeA, nodeB)
18:     change.nodeA ← nodeA
19:     change.nodeB ← nodeB
20:     return change
21: end
```

Comment: computeFlags sets label, type and value flags as specified in section 2.4.

Algorithm 3.3: createChange

Case Insertion:

From $change.kind = \mathsf{Insertion}$ it follows, that all flags are False.

$$change.kind = \mathsf{Insertion} \implies id \notin T_A \land id \in T_B$$

$$id \notin T_A \implies \nexists n \in T_A : n.id = id \implies \nexists l_A \in E_A : l_A.id = id$$

$$id \in T_B \implies \exists n \in T_B : n.id = id \implies \exists l_B \in E_B : l_B.id = id$$

Let $l_B \in E_B : l_B.id = id$ and $n_B := \textsc{Node}(l_B)$.

$$\nexists l_A \in E_A : l_A.id = id \implies l_B \notin E_A$$

$$(3.1) \implies l_B \in \text{lineAdditions}$$

$$(3.6) \implies n_B \in \texttt{nodesB}$$

$$(3.2) \implies \nexists l_A \in \text{lineRemovals} : l_A.id = id$$

$$(3.7) \implies \nexists n_A \in \texttt{nodesA} : n_A.id = id$$

$$(3.8) \implies n_B \in \texttt{onlyInB}$$

When iterating over `onlyInB` at L12, we encounter n_B. In this loop iteration

$$\texttt{change}_{14} = \textsc{createChange}(\mathsf{NULL}, n_B)$$

$$\mathsf{L14} \implies \texttt{change}_{14} \in \texttt{changes}_{16}$$

When we call createChange (3.3), we have

$$\texttt{nodeA} = \mathsf{NULL}, \texttt{nodeB} = n_B$$

We enter the branch at L13.

$$\mathsf{L14} \implies \texttt{change.id} = n_B.id = id$$

$$\mathsf{L15} \implies \texttt{change.kind} = \mathsf{Insertion}$$

At L17, because $\texttt{nodeA} = \mathsf{NULL}$, all flags are set to false.

$$\implies \textsc{createChange}(\mathsf{NULL}, n_B) = change \implies \texttt{change}_{14} = change$$

$$\implies change \in \texttt{changes}_{16}$$

Case Deletion:

From $change.kind = \mathsf{Deletion}$ it follows, that all flags are False. By a similar argument as for the insertion we get

$$n_A \in \texttt{nodesA}$$

$$\nexists n_B \in \texttt{nodesB} : n_B.id = id$$

where $n_A := \textsc{Node}(l_A)$ and $l_A \in E_A : l_A.id = id$. When iterating over `nodesA` at $\mathsf{L4}$, we encounter n_A.

$$\nexists n_B \in \texttt{nodesB} : n_B.id = id$$

$$\implies \textsc{find}(n_A.id, \texttt{nodesB}) \text{ returns } \mathsf{NULL}$$

$$\implies \texttt{nodeB}_6 = \mathsf{NULL}$$

$$\implies \texttt{change}_7 = \textsc{createChange}(n_A, \mathsf{NULL})$$

$$\mathsf{L7} \implies \texttt{change}_7 \in \texttt{changes}_{12}$$

When we call createChange (3.3), we have

$$\texttt{nodeA} = n_A, \texttt{nodeB} = \mathsf{NULL}$$

We enter the branch at $\mathsf{L2}$.

$$\mathsf{L3} \implies \texttt{change.id} = n_A.id = id$$

We then enter the branch at $\mathsf{L10}$.

$$\mathsf{L11} \implies \texttt{change.kind} = \mathsf{Deletion}$$

At $\mathsf{L17}$, all flags are set to false because $\texttt{nodeB} = \mathsf{NULL}$.

$$\implies \textsc{createChange}(n_A, \mathsf{NULL}) = change \implies \texttt{change}_7 = change$$

$$\implies change \in \texttt{changes}_{12}$$

No changes are removed from `changes` in lines $\mathsf{L12}$ - $\mathsf{L16}$ and we get

$$change \in \texttt{changes}_{16}$$

Case Move and Stationary:

We present the argument for the case where $change.kind = \mathsf{Stationary}$. The argument for Move is very similar and is thus omitted.

$$change.kind = \mathsf{Stationary} \implies id \in T_A \land id \in T_B$$

$$id \in T_A \implies \exists n \in T_A : n.id = id \implies \exists l_A \in E_A : l_A.id = id$$

$$id \in T_B \implies \exists n \in T_B : n.id = id \implies \exists l_B \in E_B : l_B.id = id$$

Let $l_A \in E_A : l_A.id = id$ and $n_A := \textsc{Node}(l_A)$.
Let $l_B \in E_B : l_B.id = id$ and $n_B := \textsc{Node}(l_B)$.

$$\text{isFake}(change) = \mathsf{False} \implies \text{ some flag is set}$$

We assume $change.labelFlag = \mathsf{True}$. The argument for the other flags is similar. For Move, instead of having a flag set, the parent IDs are different which results in the same result $l_A \neq l_B$.

$$\implies n_A.label \neq n_B.label \implies l_A \neq l_B$$

Since we assume IDs to be unique, we get

$$l_A \notin E_B \land l_B \notin E_A$$

$$\implies l_A \in \text{lineRemovals} \land l_B \in \text{lineAdditions}$$

$$(3.6) \implies n_B \in \texttt{nodesB}$$

$$(3.7) \implies n_A \in \texttt{nodesA}$$

When iterating over `nodesA` at L4, we encounter n_A.

$$n_A.id = n_B.id \implies \textsc{find}(n_A.id, \texttt{nodesB}) \text{ returns } n_B$$

$$\implies \texttt{nodeB}_6 = n_B$$

$$\mathsf{L6} \implies \texttt{change}_7 = \textsc{createChange}(n_A, n_B)$$

$$\mathsf{L7} \implies \texttt{change}_7 \in \texttt{changes}_{12}$$

When we call createChange (3.3), we have

$$\texttt{nodeA} = n_A, \texttt{nodeB} = n_B$$

We enter the branch at L2

$$\mathsf{L3} \implies \texttt{change.id} = n_A.id = id$$

We then enter the branches at L4 and L5

$$\mathsf{L8} \implies \texttt{change.kind} = \mathsf{Stationary}$$

At L17 we set the flags as specified in section 2.4 to match those of *change*.

$$\implies \textsc{createChange}(n_A, n_B) = change \implies \texttt{change}_7 = change$$

$$\implies change \in \texttt{changes}_{12}$$

No changes are removed from changes in lines L12 - L16 and we get

$$change \in \texttt{changes}_{16}$$

All cases result in $change \in \texttt{changes}_{16}$. From $\text{isFake}(change) = \text{False}$ we know that L20 is never executed with $\texttt{change}_{20} = change$, therefore

$$change \in \texttt{changes}_{21}$$

$$\implies \texttt{changes} \supseteq \text{Changes}(T_A, T_B)$$

Direction $\texttt{changes} \subseteq \text{Changes}(T_A, T_B)$

Let $change \in \texttt{changes}$. We show that $change \in \text{Changes}(T_A, T_B)$.

In computeChanges we have $change \in \texttt{changes}_{21}$

$$\implies change \in \texttt{changes}_{16} \wedge \text{L18 is never executed with } \texttt{change}_{18} = change$$

$$\implies \text{L17 is executed with } \texttt{change}_{17} = change$$

$$\implies \texttt{isFake}(change) = \text{False}$$

$$\implies change.kind \neq \text{Stationary} \vee change.labelFlag$$

$$\vee change.typeFlag \vee change.valueFlag$$

It follows that at least one of the following cases must occur.

Case 1: $change.kind = \text{Insertion}$

In createChange, when creating $change$, let $nodeB = \texttt{nodeB}$.

$$change.kind = \text{Insertion} \implies \text{L15 is executed.}$$

$$\implies change.id = nodeB.id$$

$$\implies \texttt{nodeA} = \text{NULL} \implies change = \text{createChange}(\text{NULL}, nodeB)$$

$$\implies \text{In computeChanges}, change \text{ was created at L13}$$

$$\implies nodeB \in \texttt{onlyInB} \implies nodeB \in \texttt{nodesB}$$

$$\wedge \nexists nodeA \in \texttt{nodesA} : nodeA.id = nodeB.id$$

Let $lineB$ be such that $\text{Node}(lineB.content) = nodeB$.

$$nodeB \in \texttt{nodesB} \implies lineB \in \text{lineAdditions}$$

$$\implies lineB \in E_B \implies nodeB \in T_B$$

The nonexistence of a corresponding node in `nodesA` lets us infer the nonexistence of a corresponding node line in the difference of the encodings.

$$\nexists nodeA \in \texttt{nodesA} : nodeA.id = nodeB.id$$

$$\implies \nexists lineA \in \text{lineRemovals} : \textsc{Node}(lineA.content).id = nodeB.id$$

$$\implies \nexists lineA \in \text{lineRemovals} : lineA.id = nodeB.id$$

$$\implies \nexists lineA \in E_A - E_B : lineA.id = nodeB.id$$

We investigate the two possible cases.

Case 1.1: $\exists lineA \in E_A \cap E_B : lineA.id = nodeB.id$

$$\implies lineA.id = nodeB.id = lineB.id \wedge lineA, lineB \in E_B$$

$$\text{IDs are unique in } E_B \implies lineA = lineB$$

$$lineB \in E_A \cap E_B \wedge lineB \in \text{lineAdditions}$$

$$(3.5) \implies lineB \in \text{lineRemovals} \lightning \nexists lineA \in \text{lineRemovals} : lineA.id = nodeB.id$$

A contradiction to the assumption, therefore this case does not occur.

Case 1.2: $\nexists lineA \in E_A : lineA.id = nodeB.id$

$$\implies \nexists nodeA \in T_A : nodeA.id = nodeB.id \implies nodeB.id \notin T_A$$

$$nodeB.id \notin T_A \wedge nodeB.id \in T_B \wedge change.id = nodeB.id$$

$$\implies \text{ChangeMap}_{T_A,T_B}(change.id).kind = \textsf{Insertion}$$

$$\implies \text{isFake}(\text{ChangeMap}_{T_A,T_B}(change.id)) = \textsf{False}$$

$$\implies \text{ChangeMap}_{T_A,T_B}(change.id) \in \text{Changes}(T_A, T_B)$$

Case 2: $change.kind = \textsf{Deletion}$

In createChange, when creating *change*, let $nodeA = \texttt{nodeA}$.

$$change.kind = \textsf{Deletion} \implies \textsf{L11 is executed.} \implies change.id = nodeA.id$$

$$\implies \texttt{nodeB} = \textsf{NULL} \implies change = \textsc{createChange}(nodeA, \textsf{NULL})$$

$$\implies \text{In } \textsc{computeChanges}, change \text{ was created at } \textsf{L6}$$

$$\implies nodeA \in \texttt{nodesA}$$

$$\texttt{nodeB}_6 = \textsf{NULL} \implies \nexists nodeB \in \texttt{nodesB} : nodeB.id = nodeA.id$$

An argument symmetrical to the above gives

$$\text{ChangeMap}_{T_A,T_B}(change.id).kind = \textsf{Deletion}$$

$$\text{ChangeMap}_{T_A,T_B}(change.id) \in \text{Changes}(T_A, T_B)$$

Case 3: $change.kind = \mathsf{Move}$

In createChange, when creating $change$, let $nodeA = \texttt{nodeA}, nodeB = \texttt{nodeB}$.

$$change.kind = \mathsf{Move} \implies \mathsf{L8} \text{ is executed.}$$

$$\implies nodeA.parentId \neq nodeB.parentId \wedge change.id = nodeA.id$$

$$\texttt{nodeA} \neq \mathsf{NULL} \neq \texttt{nodeB}$$

$$\implies \text{In computeChanges}, change \text{ was created at } \mathsf{L6} \implies nodeA \in \texttt{nodesA}$$

$$\texttt{nodeB}_6 = nodeB \implies \text{at } \mathsf{L5}\ \texttt{nodesB.find}(nodeA.id) \text{ returns } nodeB$$

$$\implies nodeB \in \texttt{nodesB} \wedge nodeB.id = nodeA.id$$

Let $lineA, lineB$ be such that $nodeA = \texttt{Node}(lineA)$ and $nodeB = \texttt{Node}(lineB)$.

$$(3.6), (3.7) \implies lineA \in \text{lineRemovals} \wedge lineB \in \text{lineAdditions}$$

$$(3.1) \implies lineA \in E_A \implies nodeA \in T_A \implies nodeA.id \in T_A$$

$$(3.2) \implies lineB \in E_B \implies nodeB \in T_B \implies nodeB.id \in T_B$$

$$nodeA.id = nodeB.id = change.id$$

$$change.id \in T_A \wedge change.id \in T_B \wedge nodeA.parentId \neq nodeB.parentId$$

$$\implies \text{ChangeMap}_{T_A,T_B}(change.id).kind = \mathsf{Move}$$

$$\implies \text{isFake}(\text{ChangeMap}_{T_A,T_B}(change.id)) = \mathsf{False}$$

$$\implies \text{ChangeMap}_{T_A,T_B}(change.id) \in \text{Changes}(T_A, T_B)$$

Case 4: Some flag is set.

We present the argument for the case where $change.labelFlag = \mathsf{True}$. The argument for the other flags is similar. In createChange, when creating $change$, let $nodeA = \texttt{nodeA}$ and $nodeB = \texttt{nodeB}$.

$$change.labelFlag = \mathsf{True}$$

$$\implies nodeA \neq \mathsf{NULL} \neq nodeB \wedge nodeA.label \neq nodeB.label$$

$$\implies \text{In computeChanges}, change \text{ was created at } \mathsf{L6}$$

From here on, the argument is similar to the Move case. We get

$$\text{ChangeMap}_{T_A,T_B}(change.id).labelFlag = \mathsf{True}$$

$$\implies \text{isFake}(\text{ChangeMap}_{T_A,T_B}(change.id)) = \mathsf{False}$$
$$\implies \text{ChangeMap}_{T_A,T_B}(change.id) \in \text{Changes}(T_A, T_B)$$

To recognize the equality $change = \text{ChangeMap}_{T_A,T_B}(change.id)$, we remind ourselves of the uniqueness of IDs in $\text{Changes}(T_A, T_B)$. That is when considering two changes $change_1, change_2 \in \text{Changes}(T_A, T_B)$ with equal IDs $change_1.id = change_2.id$, then $change_1 = change_2$. One may then consider the components of *change* one at a time to see that kind and all set flags are equal. Equality of unset flags follows by the arguments for the Move and Stationary case in the proof of `changes` $\supseteq \text{Changes}(T_A, T_B)$. With this we have shown

$$change \in \text{Changes}(T_A, T_B)$$
$$\implies \texttt{changes} \subseteq \text{Changes}(T_A, T_B)$$
$$\implies \texttt{changes} = \text{Changes}(T_A, T_B)$$

□

Direction `structChanges` $\supseteq \text{StructChanges}(T_A, T_B)$

Let $id \in \text{StructChanges}(T_A, T_B)$. We show that $id \in$ `structChanges`. By definition, there must be a $change \in \text{Changes}(T_A, T_B)$ such that at least one of the following is true.

- $change.kind \in \{\mathsf{Insertion}, \mathsf{Move}\} \wedge n_B.parentId = id$
- $change.kind \in \{\mathsf{Deletion}, \mathsf{Move}\} \wedge n_A.parentId = id$
- $change.labelFlag \wedge n_A.parentId = id$

where $n_X \in T_X : n_X.id = change.id$. We present the argument for the first case and omit the other cases. Their arguments are very similar.

By the previous result, we know

$$change \in \texttt{changes}$$

In computeStructChanges, when we iterate over `changes` we encounter *change* and in the relevant iteration where `change` = *change*:

$$change.kind \in \{\mathsf{Insertion}, \mathsf{Move}\} \implies \text{the condition at } \mathsf{L4} \text{ is satisfied}$$

$$\mathsf{L5} \implies \texttt{change.nodeB.parentId} \in \texttt{structChanges}$$

We know `change.nodeB` $\neq$ NULL because of the change kind. By definition: `change.nodeB` $= node \in T_B : node.id = change.id$. Because of the uniqueness of IDs, we get

$$\texttt{change.nodeB} = n_B$$

```
 1: function computeStructChanges(changes)
 2:     structChanges ← ∅
 3:     for all changes do
 4:         if change.kind ∈ {Insertion, Move} then
 5:             structChanges.add(change.nodeB.parentId)
 6:         end
 7:         if change.kind ∈ {Deletion, Move} then
 8:             structChanges.add(change.nodeA.parentId)
 9:         end
10:         if change.labelFlag then
11:             structChanges.add(change.nodeA.parentId)
12:         end
13:     end
14:     return structChanges
15: end
```

Comment: On L5, L8 and L11, we know from the kind of the change that the respective node pointer is not NULL and that we can access `parentId`.

Algorithm 3.4: computeStructChanges

$$\implies n_B.parentId \in \texttt{structChanges}$$

$$\implies id \in \texttt{structChanges}$$

$$\implies structChanges \supseteq \text{StructChanges}(T_A, T_B)$$

Direction $\texttt{structChanges} \subseteq \text{StructChanges}(T_A, T_B)$

Let $id \in \texttt{structChanges}$. We show that $id \in \text{StructChanges}(T_A, T_B)$. id must have been added at one of the lines L5, L8 or L11. We present the argument for the case L5 and omit the other cases. Their arguments are very similar.

$$\text{L5} \implies \texttt{change.nodeB.parentId} = id$$

Let $change = \texttt{change}$.

$$\text{L3} \implies change \in \texttt{changes} \implies change \in \text{Changes}(T_A, T_B)$$

$$\text{L4} \implies change.kind \in \{\text{Insertion, Move}\}$$

$$\text{By definition: } \texttt{change.nodeB} = n_B \in T_B : n_B.id = change.id$$

$$\implies n_B.parentId = id$$

$$change.kind \in \{\text{Insertion, Move}\} \land n_B.parentId = id$$

$$\implies id \in \text{StructChanges}(T_A, T_B)$$
$$\implies \texttt{structChanges} \subseteq \text{StructChanges}(T_A, T_B)$$
$$\implies \texttt{structChanges} = \text{StructChanges}(T_A, T_B)$$

□

3.3.4 Final output

We show that the *AST diff* algorithm returns the set $\text{XChanges}(T_A, T_B)$ that is, we show that in ASTDIFF, $\texttt{xChanges}_{30} = \text{XChanges}(T_A, T_B)$. The argument builds on the results from section 3.3.3.

Direction $\texttt{xChanges} \supseteq \text{XChanges}(T_A, T_B)$

Let $xChange \in \text{XChanges}(T_A, T_B)$. We show that $xChange \in \texttt{xChanges}$. Consider these two cases.

Case 1: $\exists change \in \text{Changes}(T_A, T_B) : change.id = xChange.id$
$xChange$ must have been included in XChanges by the first method (2.2), giving us

$$xChange = change \oplus xChange.structFlag$$
$$\texttt{structChanges} = \text{StructChanges}(T_A, T_B) \implies$$
$$\forall id : \texttt{structChanges.contains}(id) \equiv id \in \text{StructChanges}(T_A, T_B)$$

When iterating over `changes` at L15, we encounter *change*.

$$\texttt{structChanges.contains}(change.id) = xChange.structFlag$$
$$\implies \texttt{xChange} = change \oplus xChange.structFlag = xChange$$
$$\text{L17} \implies xChange \in \texttt{xChanges}$$

Case 2: $\nexists change \in \text{Changes}(T_A, T_B) : change.id = xChange.id$
$xChange$ must have been included in XChanges by the second method (2.3), giving us

$$xChange.id \in \text{StructChanges}(T_A, T_B)$$
$$xChange.kind = \text{Stationary}, xChange.structFlag = \text{True}$$
$$xChange.labelFlag = xChange.typeFlag = xChange.valueFlag = \text{False}$$

When iterating over `structChanges` at L19, we encounter $xChange.id$.

$$\nexists change \in \text{Changes}(T_A, T_B) : change.id = xChange.id$$

$$\implies \texttt{changes.find}(xChange.id) = \mathsf{NULL}$$

$$\mathsf{L21} \implies \texttt{xChange.id} = xChange.id$$

$$\mathsf{L22} \implies \texttt{xChange.nodeA} \in T_A \land \texttt{xChange.nodeA.id} = xChange.id$$

$$\mathsf{L22} \implies \texttt{xChange.nodeB} \in T_B \land \texttt{xChange.nodeB.id} = xChange.id$$

$$\mathsf{L24} \implies \texttt{xChange.kind} = \mathsf{Stationary} = xChange.kind$$

$$\mathsf{L25} \implies \texttt{xChange.labelFlag} = \texttt{xChange.typeFlag} =$$
$$\texttt{xChange.valueFlag} = \mathsf{False}$$

$$\mathsf{L26} \implies \texttt{xChange.structFlag} = \mathsf{True} = xChange.structFlag$$

$$\implies \texttt{xChange} = xChange$$

$$\mathsf{L27} \implies xChange \in \texttt{xChanges}$$

Both cases result in $xChange \in \texttt{xChanges}$, therefore

$$\texttt{xChanges} \supseteq \text{XChanges}(T_A, T_B)$$

Direction $\texttt{xChanges} \subseteq \text{XChanges}(T_A, T_B)$

Let $xChange \in \texttt{xChanges}$. We show that $xChange \in \text{XChanges}(T_A, T_B)$. The following two cases can occur.

Case 1: $\exists change \in \texttt{changes} : change.id = xChange.id$

$$\implies change \in \text{Changes}(T_A, T_B)$$

Assume $xChange$ is created at L21.

$$\mathsf{L20} \implies \texttt{changes.find}(xChange.id) = \mathsf{NULL}$$

$$\implies \nexists change \in \texttt{changes} : change.id = xChange.id \; \lightning$$

which is in contradiction to our case assumption therefore $xChange$ is created at L16.

$$\mathsf{L16} \implies xChange = \texttt{change} \oplus \texttt{structChanges.contains}(\texttt{change.id})$$

$$\implies xChange.id = \texttt{change.id} = change.id \implies \texttt{change} = change$$

$$\implies xChange = change \oplus change.id \in \text{StructChanges}(T_A, T_B)$$

$$(2.2) \implies xChange \in \text{XChanges}(T_A, T_B)$$

Case 2: $\nexists change \in \texttt{changes} : change.id = xChange.id$

$$\implies \nexists change \in \text{Changes}(T_A, T_B) : change.id = xChange.id$$

Assume $xChange$ is created at L16.

$$\text{L16} \implies xChange = \texttt{change} \oplus \texttt{structChange.contains}(\texttt{change.id})$$

$$\implies xChange.id = \texttt{change.id}$$

$$\implies \exists change \in \texttt{changes} : change.id = xChange.id \lightning$$

which is in contradiction to our case assumption therefore $xChange$ is created at L21.

$$\text{L21} \implies xChange.id = \texttt{id}$$

$$\text{L22} \implies xChange.nodeA \in T_A \land xChange.nodeA.id = \texttt{id}$$

$$\text{L22} \implies xChange.nodeB \in T_B \land xChange.nodeB.id = \texttt{id}$$

$$\text{L24} \implies xChange.kind = \text{Stationary}$$

$$\text{L25} \implies xChange.labelFlag = xChange.typeFlag =$$

$$xChange.valueFlag = \text{False}$$

$$\text{L26} \implies xChange.structFlag = \text{True}$$

$$\text{L19} \implies \texttt{id} \in \texttt{structChanges}$$

$$\implies xChange.id \in \text{StructChanges}(T_A, T_B)$$

$$(2.3) \implies xChange \in \text{XChanges}(T_A, T_B)$$

Both cases result in $xChange \in \text{XChanges}(T_A, T_B)$, therefore

$$\texttt{xChanges} \subseteq \text{XChanges}(T_A, T_B)$$

$$\implies \texttt{xChanges} = \text{XChanges}(T_A, T_B)$$

□

Chapter 4

The *AST Merge* Algorithm

In this chapter we present the *AST Merge* algorithm which we have developed. It takes a modular approach in that the user can supply plug-in-like components to customize its behavior. We describe the algorithm as well as abstractions, data structures, submodules and interfaces used. We also present two examples of the mentioned components that when employed, result in a nontrivial algorithm for merging ASTs.

The *AST Merge* algorithm takes the following inputs:

- Two ASTs T_A and T_B from two different branches A and B.
- Their closest ancestor T_{base}.
- A pipeline initializer.
- A list of conflict pipeline components.

And it returns the following outputs:

- An AST T_{merged} that is the merged version of T_A and T_B.
- A set of conflict pairs $conflictPairs$ that could not be resolved.

As figure 4.1 illustrates, we first use the introduced *AST Diff* algorithm to compute the sets of changes each of the branches makes relative to T_{base}. From each of these sets we build a change dependency graph, a concept that will be introduced shortly. Next we execute a single pipeline initializer and then a pipeline of independent components that transform these trees and graphs and detect and resolve conflicts between changes. Then we order all non-conflicting changes topologically according to their dependencies and apply them to T_{base}, creating the tree T_{merged}. The last step is to remove any gaps in lists which may have been introduced and then return T_{merged}

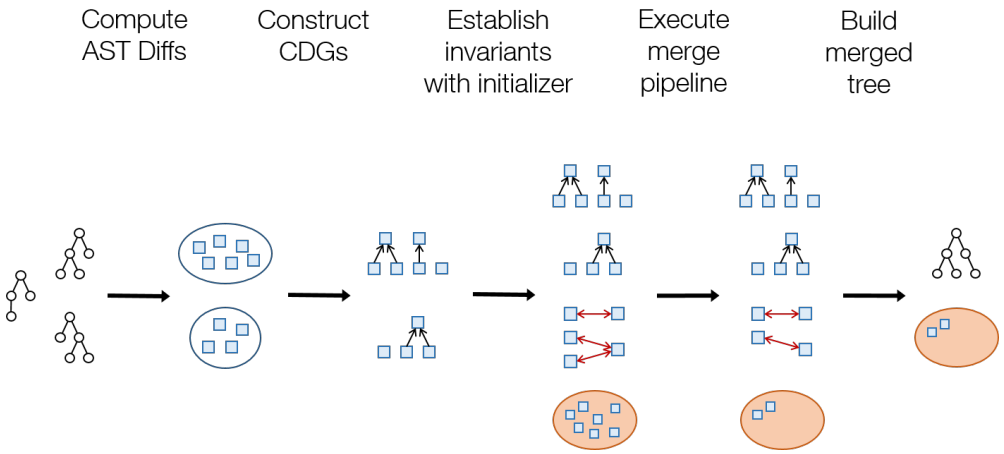


Figure 4.1: Illustration of the *AST Merge* algorithm.

which, together with the set of unresolved conflicts, forms the output of the algorithm.

4.1 Change Dependency Graphs (CDGs)

A node can only be inserted or moved if its new parent exists in the tree. Furthermore, a node can only be deleted if it has no children. For this reason, the application of some changes depends on the earlier application of other changes. For each branch, we express these dependencies as a directed graph with changes as vertices. This graph contains an edge $(change_1, change_2)$ if applying $change_1$ requires $change_2$ to be applied before. Note that these dependencies only make sure that the parent of any node exists in the tree. They do not ensure label-uniqueness under parents or prohibit cyclic relationships of nodes. Also note that the graph is not necessarily connected.

We initialize these graphs as follows. Let *changes* be the set of changes some branch makes. CDG $= (changes, E)$ where $(change_1, change_2) \in E$ if

- $change_1.nodeB.parentId = change_2.nodeB.id \wedge$
  $change_1.kind \in \{\text{Insert, Move}\} \wedge change_2.kind = \text{Insert}$
- $change_1.nodeA.id = change_2.nodeA.parentId \wedge$
  $change_1.kind = \text{Delete} \wedge change_2.kind \in \{\text{Delete, Move}\}$

```
1: function astMerge(treeBase, treeA, treeB, pipelineInitializer, conflict-
   Pipeline)
2:     diffA ← astDiff(treeBase, treeA)
3:     diffB ← astDiff(treeBase, treeB)
4:     cdgA ← buildCdg(diffA)
5:     cdgB ← buildCdg(diffB)
6:     conflictingChanges ← ∅
7:     conflictPairs ← ∅
8:     transition ← pipelineInitializer.run(treeBase, treeA, treeB, cdgA,
   cdgB, conflictingChanges, conflictPairs)
9:     check(transition)
10:    for all components in conflictPipeline do
11:        transition ← component.run(treeBase, treeA, treeB, cdgA, cdgB,
   conflictingChanges, conflictPairs)
12:        check(transition)
13:    end
14:    applicableChanges ← cdgA ∪ cdgB − conflictingChanges
15:    sort(applicableChanges, cdgA, cdgB)
16:    treeMerge ← treeBase
17:    for all applicableChanges do
18:        applyChange(treeMerge, change)
19:    end
20:    for all changed lists do
21:        compactList(list)
22:    end
23:    return [treeMerge, conflictingChanges]
24: end
```

Comment: In `check(transition)` we check the transition returned by the last component with all appropriate component checkers. `sort(applicableChanges, cdgA, cdgB)` sorts the changes topologically according to their dependencies given by the CDGs. `compactList(list)` removes gaps in modified lists so they are continuous.

Algorithm 4.1: The *AST Merge* algorithm

4.1.1 CDGs are acyclic

We show that CDGs contain no circular dependencies by assuming the existence of a cycle and showing that this leads to a contradiction. From the definition we can observe that on any path, once a change with $change.kind \in \{\mathsf{Insert}, \mathsf{Move}\}$ has been reached, the only outgoing edges are to changes of kind Insert hence all following changes in the path are of kind Insert. Therefore any cycle must consist of either only Insert changes or only Delete changes. Consider the case where the cycle contains Insert changes only. Let $(change_1, change_2)$ be any edge in the cycle, let l be the length of the cycle and let $h(node)$ be the depth of $node$ in the tree.

$$change_1.kind = change_2.kind = \mathsf{Insert} \implies$$
$$change_1.nodeB.parentId = change_2.nodeB.id \implies$$
$$change_1.nodeB.parent = change_2.nodeB \implies$$
$$h(change_1.nodeB) - 1 = h(change_2.nodeB)$$

Applying the same argument for the next edge in the cycle $(change_2, change_3)$ gives us

$$h(change_2.nodeB) - 1 = h(change_3.nodeB) \implies$$
$$h(change_1.nodeB) - 2 = h(change_2.nodeB) - 1 = h(change_3.nodeB)$$

Extending this to the whole cycle, we get

$$h(change_1.nodeB) - l = h(change_2.nodeB) - (l-1) = \cdots = h(change_1.nodeB)$$

which is a contradiction therefore no such cycle exists. The argument works symmetrically for the case where the graph consists of Delete changes. □

4.2 Conflict detection and resolution pipeline

Our merge algorithm supports detection and resolution of conflicts by plug-in-like components. These components share a unified interface and work in a pipeline to improve the merged version of the tree by detecting conflicting changes and resolving these conflicts. We call such components *pipeline components*. Before the first component of the pipeline can be executed, the invariants of said interface must be established. For this reason we define a second class of components called *pipeline initializers*. One such pipeline initializer must be supplied to the *AST Merge* algorithm. It is then executed before the first pipeline component to establish the invariants of the pipeline. The following sections will introduce one pipeline initializer and one pipeline component that, when used to configure the generic algorithm, implement a valid and nontrivial merge algorithm.

4.2.1 Interface of pipeline initializers

Pipeline initializers provide a method RUN that takes the following objects as input.

- ASTs T_{base}, T_A, T_B. These are immutable.
- CDGs cdg_A, cdg_B.
- A set $conflictingChanges \subseteq cdg_A.changes \cup cdg_B.changes$.
- A set $conflictPairs \subseteq cdg_A.changes \times cdg_B.changes$.

Additionally, pipeline initializers may rely on the following preconditions.

1. T_{base}, T_A and T_B are of valid form.
2. Applying all and only changes from cdg_A to T_{base} results in T_A.
3. Applying all and only changes from cdg_B to T_{base} results in T_B.
4. Both sets $conflictingChanges$ and $conflictPairs$ are empty.
5. cdg_A and cdg_B are acyclic

Pipeline initializers must establish the pipeline invariants which are described below.

4.2.2 Interface of pipeline components

Like pipeline initializers, pipeline components also provide a method RUN taking the same inputs, now with T_A and T_B being mutable as well. Let $nonconflicting = cdg_A.changes \cup cdg_B.changes - conflictingChanges$ and let T_{merged} be the tree that results from applying all changes in $nonconflicting$ to T_{base}. Following are the pipeline invariants. Each pipeline component may rely on them when it is executed. Each component must have reestablished them when it returns.

The pipeline invariants

1. If sorted topologically by dependencies, all changes in $nonconflicting$ can be applied.
2. T_{merged} is of valid form.
3. Node IDs are unique in T_A and T_B.

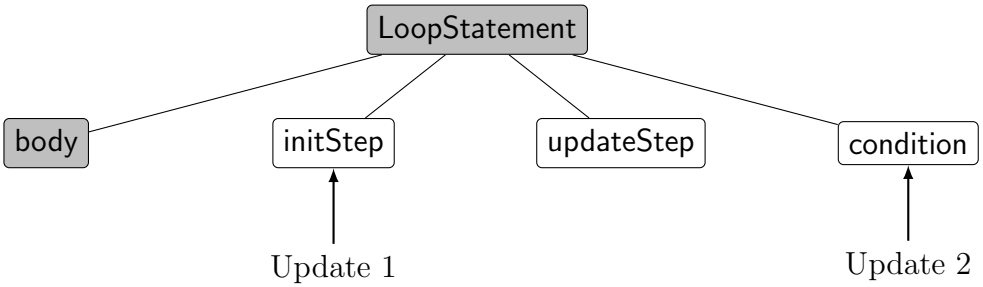


Figure 4.2: Semantic conflict in a loop statement. (adapted from Otth [6])

4. All conflicting changes are either part of a conflict pair or depend on a conflicting change.

5. cdg_A and cdg_B are acyclic.

Mutability and valid from of T_A and T_B

An interesting thing to note is why T_A and T_B are mutable. By allowing each component to make changes to these trees in addition to modifying change objects, it is much simpler for later components in the pipeline to build on the work of earlier components as the trees represent the result of applying all changes in the respective CDG to T_{base}. Since such earlier work only makes sense when considered in the context of a merge, it is not required that the trees T_A and T_B by themselves are of valid form but only that they maintain the ID uniqueness of nodes. It is worth mentioning that these modified trees do not represent any concrete version of the program code and are never encoded and written to disk.

4.2.3 The *Conflict Unit* pipeline initializer

We present a pipeline initializer that detects conflicts based on conflict units, a concept originally introduced by Otth [6]. The following motivational example is taken from Otth's master thesis. Consider a loop statement node and its child nodes describing the loop body, loop variable initialization, increment and loop condition as illustrated in figure 4.2. Now consider the case where branch A changes the initialization node and branch B changes the condition node. While both of these changes could be applied together without creating an ill formed AST it is very likely that this will result in incorrect semantics.

We see that in this situation the changes of both branches are not in a *syntactic* conflict but very likely in a *semantic* conflict. Our goal is not only to detect syntactic but also some semantic conflicts. We do this by defining neighborhoods of nodes which are closely related to each other and only allowing one branch to make changes to any such neighborhood. Getting back to the example with the loop, it would make sense that the initialization, update and condition nodes and the loop statement node itself make up such a neighborhood. Now both branches make changes to that neighborhood and a conflict is detected.

Conflict units are an implementation of this idea of neighborhoods of related nodes where the neighborhoods are connected sections of the AST. To define these sections, we consider a subset of node types as special and split the tree at nodes of such types. See figure 4.3 for an illustrated example. The choice of this subset is artificial and based on a manual assessment of the programming language. In our example, both LoopStatement and body would be one of these special types but in some cases it might be reasonable to regard the loop body as belonging to the same conflict unit as the other nodes. Typically though, this set will include types for classes, methods and statement lists, among others. We will now describe the concept of conflict units in a more formal way.

Definition of conflict units and conflict roots

The *Conflict Unit* pipeline initializer is configured with a set of node types ConflictTypes. We call all nodes with $node.type \in$ ConflictTypes *conflict roots*. We require and assume that the roots of T_A, T_B and T_{base} are such conflict roots. ConflictRoot($node$) is the closest ancestor of $node$ that is a conflict root. If $node$ is itself a conflict root, it is mapped to itself.

$$\text{ConflictUnit}(node) = \{node' : \text{ConflictRoot}(node') = \text{ConflictRoot}(node)\}$$

A set of nodes that have the same conflict root is a *conflict unit*. We identify a conflict unit by the ID of its root node.

A conflict unit is affected by a branch if the branch makes changes to nodes in that conflict unit[1]. A conflict unit is *in conflict* if it is affected by both branches. In such a case, all changes of branch A affecting this conflict unit are considered conflicting with all changes of branch B affecting this conflict unit.

[1]Note that it is possible a change to affect two conflict units at once.

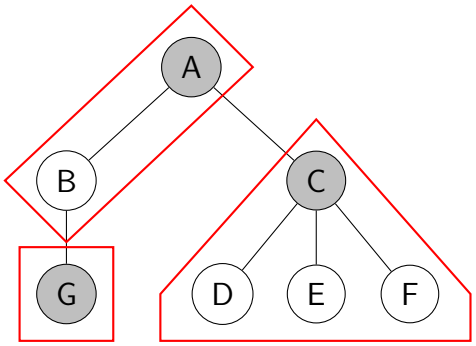


Figure 4.3: Visualization of conflict units. Conflict roots are in gray and the conflict units in red. Nodes A and B belong to conflict unit A. Nodes C, D, E and F belong to conflict unit C. Node G belongs to conflict unit G and is its sole member.

Child-structure-change conflicts

The main purpose of the $structFlag$ flag of change objects is to detect concurrent changes that, if they were applied, would cause the AST to become ill formed. Hence, if we encounter two changes (of different branches) affecting the same node and both having the $structFlag$ set, we mark any changes to the children of this node that would cause the $structFlag$ flag to be set as conflicting. To specify this formally, we define the set of changes of a branch that affect the child structure of a node with a certain ID:

$$\text{StructureChanges}_{cdg}(id) := \{change \in cdg :$$

$$(change.nodeA.parentId = id \vee change.nodeB.parentId = id) \wedge$$

$$(change.kind \neq \mathsf{Stationary} \vee change.labelFlag = \mathsf{True})\}$$

Then for any two changes $changeA \in cdgA, changeB \in cdgB$ with $id := changeA.id = changeB.id \wedge changeA.structFlag \wedge changeB.structFlag$ we add the following set to $conflictPairs$:

$$\text{StructureChanges}_{cdgA}(id) \times \text{StructureChanges}_{cdgB}(id)$$

Label-based dependencies between changes

The dependencies represented by the CDGs are of a very basic nature. They simply ensure that the final tree, every node has a valid parent. The CDGs do not express dependencies that are created by the invariant of valid form and its requirement of label uniqueness under a single parent (see section

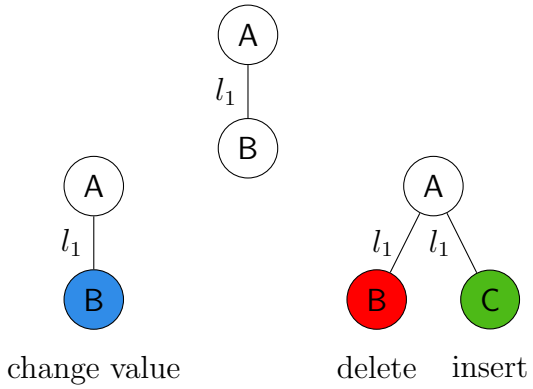


Figure 4.4: A conflicting deletion causing a label collision.

2.2.1). In order to establish the pipeline invariant, the conflict unit pipeline initializer must take these more subtle dependencies into account.

In the example shown in figure 4.4 one branch deletes node B and inserts a new node C at the same position while the other branch changes the value of B. The deletion and the value change are in conflict. Now that the deletion is not applied, the insertion must also not be applied or nodes B and C will have the same label under the same parent.

We define label dependencies as follows. Consider a *change* with

$$change.kind \in \mathsf{Deletion}, \mathsf{Move} \vee change.labelFlag = \mathsf{True}$$

Another change $change'$ from the same CDG depends on *change* if the following holds

$$change.nodeA.parentId = change'.nodeB.parentId$$

$$\wedge change.nodeA.label = change'.nodeB.label$$

Note that unlike the CDGs, these label-based dependencies can contain cycles. Also note that for each *change* there is never more than one such depending change. Consider the case where both $change_1$ and $change_2$ depend on *change*. This would imply

$$change_1.nodeB.parentId = change_2.nodeB.parentId$$

$$\wedge change_1.nodeB.label = change_2.nodeB.label$$

which is in violation of the second requirement of valid form (see section 2.2.1). It follows that this branch's version of the tree is not of valid form which is in contradiction with precondition 1 of pipeline initializers (see section 4.2.1).

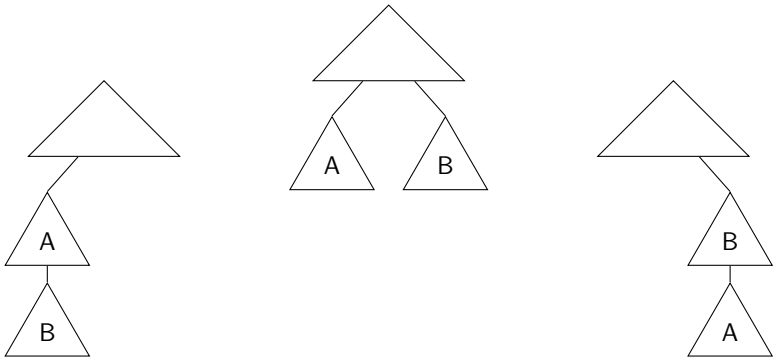


Figure 4.5: The combination of the moves would create a cycle.

Avoiding the creation of cycles in ASTs

Even though all input trees are valid, it is possible that in combination, the changes of both branches creates cycles in the merged AST. Take the example shown in figure 4.5. One branch moves the root of subtree B into the subtree A and the other branch moves the root of subtree A into the subtree B. If both of these changes were applied then a cycle would be created.

To detect such cases, we do additional checks for Move changes. We define a boolean function on node IDs that detects if one node is an ancestor of another:

$$\text{isAncestor}(T, id, ancestorId) \equiv$$

$$\exists node, ancestor \in T : node.id = id \wedge ancestor.id = ancestorId$$

$$\wedge \exists i \geq 0 : node \underbrace{.parent.parent \cdots .parent}_{i} = ancestor$$

Then for every Move change of branch A, we check whether branch B moves an ancestor of the moved node into that node's subtree. That is, for any $changeA \in cdgA : changeA.kind = \mathsf{Move}$, we check the following condition.

$$\forall ancestor \in T_A : \text{isAncestor}(T_A, changeA.id, ancestor.id) \rightarrow$$

$$\forall changeB \in cdgB : (changeB.id = ancestor.id \wedge changeB.kind = \mathsf{Move}) \rightarrow$$

$$\neg\text{isAncestor}(T_B, ancestor.id, changeA.id)$$

If this condition is not satisfied then applying both $changeA$ and the offending $changeB$ will result in a cycle in the merged tree. To avoid this we add $(changeA, changeB)$ to $conflictPairs$. The same is done symmetrically for branch B. For an implementation that checks this condition, refer to algorithm 4.2.

```
 1: function checkMove(changeA, cdgB)
 2:     conflictingMoves ← ∅
 3:     ancestor ← changeA.nodeB.parent
 4:     while ancestor ≠ ROOT do
 5:         changeB ← cdgB.find(ancestor.id)
 6:         if changeB ≠ NULL ∧ changeB.kind = Move then
 7:             if isAncestor(changeB.nodeB, changeA.id) then
 8:                 conflictingMoves.add(changeB)
 9:             end
10:         end
11:         ancestor ← ancestor.parent
12:     end
13:     conflictPairs.addAll(changeA × conflictingMoves)
14:     conflictingChanges.add(changeA)
15:     conflictingChanges.addAll(conflictingMoves)
16: end
```

Comment: When calling `isAncestor(node, ancId)`, `node` specifies both the tree and the starting node: The check is evaluated for the tree that `node` is part of.

Algorithm 4.2: Detecting conflicting move operations.

By this method, we detect cycles that could be introduced by combination of changes of both branches. However, cycles can also be introduced by not applying certain moves because of conflicts. Consider the case shown in figure 4.6 where a branch moves subtree B out of subtree A and then moves subtree A into subtree B. By itself, these are valid changes but the move of the root of B is marked as conflicting with the value change of the other branch. If we were now to only apply the move of A then a cycle would be created. In some sense, the move of A depends on the move of B.

We introduce the notion of move dependencies. We define an auxiliary function moveBound(T_A, T_B, id) to be the ID of the node whose subtree bounds in both trees the move of the node specified by id. Let $P^A = [p_1^A \dots p_n^A]$ and $P^B = [p_1^B \dots p_m^B]$ be the paths resulting in walking up from the node specified by id in the respective trees T_A and T_B such that $p_1^A = p_1^B = id$ and $p_n^A = p_m^B =$ ROOT. moveBound(T_A, T_B, id) is then the first ID of the longest common suffix of P^A and P^B.

For a branch X we say change $ch_1 \in cdgX$ depends on change $ch_2 \in cdgX$ if ch_1 moves a *node* to a parent that is in the subtree of *node* in the base version and ch_2 moves a node that, in the branch version, lies on the path from *node* to moveBound$(T_{base}, T_X, node.id)$. More formally, ch_1 depends on

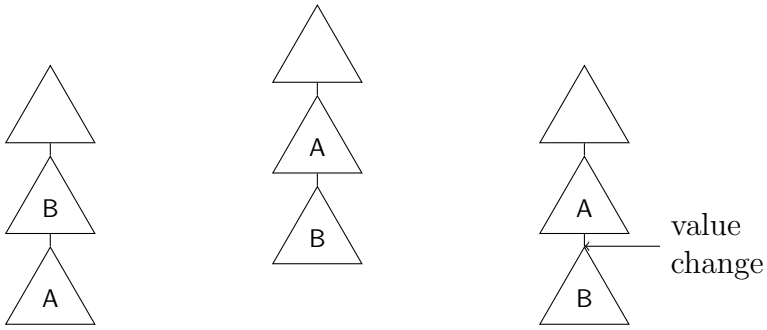


Figure 4.6: Conflicts can result in a cycle.

ch_2 if the following holds:

$$ch_1.kind = ch_2.kind = \mathsf{Move} \wedge$$

$$\text{isAncestor}(T_{base}, ch_1.nodeB.parentId, ch_1.nodeB.id) \wedge$$

$$\text{isAncestor}(T_X, ch_1.id, ch_2.id) \wedge \text{isAncestor}(T_X, ch_2.id, bound)$$

where $bound = \text{moveBound}(T_{base}, T_X, ch_1.id)$.

```
 1: function findMoveDependencies(change, cdg)
 2:     dependencies ← [ ]
 3:     newParentInBase ← treeBase.find(change.nodeB.parentId)
 4:     if isAncestor(newParentInBase, change.id) then
 5:         moveBound ← moveBound(change.nodeA, change.nodeB)
 6:         node ← change.nodeB.parent
 7:         while node.id ≠ moveBound do
 8:             change2 ← cdg.find(node.id)
 9:             if change2 ≠ NULL ∧ change2.kind = Move then
10:                 dependencies.append(change2)
11:             end
12:             node ← node.parent
13:         end
14:     end
15:     return depedencies
16: end
```

Algorithm 4.3: findMoveDependencies

Note that child-structure-change conflicts, label dependencies and move dependencies are not related to conflict units. These concepts complement

```
 1: function moveBound(nodeA, nodeB)
 2:     pathStackA ← [ ]
 3:     pathStackB ← [ ]
 4:     while nodeA ≠ ROOT do
 5:         pathStackA.push(nodeA)
 6:         nodeA ← nodeA.parent
 7:     end
 8:     while nodeB ≠ ROOT do
 9:         pathStackB.push(nodeB)
10:         nodeB ← nodeB.parent
11:     end
12:     moveBound ← ROOT
13:     while pathStackA.peek = pathStackB.peek do
14:         moveBound ← pathStackA.pop
15:         pathStackB.pop
16:     end
17:     return moveBound
18: end
```

Algorithm 4.4: moveBound

conflict-unit-based conflict detection and are necessary to establish the pipeline invariants.

Computing conflicts based on conflict units

To find and mark all such conflicts, we begin by computing the set of affected conflict units for each branch and then consider the intersection of these two sets. For every conflict unit CU in this set, let $\text{Changes}_{A,CU}$ and $\text{Changes}_{B,CU}$ be the sets of changes affecting CU of branches A and B respectively. We add all pairs $\text{Changes}_{A,CU} \times \text{Changes}_{B,CU}$ to $conflictPairs$. We also add all changes $\text{Changes}_{A,CU} \cup \text{Changes}_{B,CU}$ and all changes (transitively) depending on these to $conflictingChanges$. For additional details, see algorithm 4.5.

To compute the set of affected conflict units for a branch, we simply go over all its changes, for each computing the affected conflict unit and adding its conflict root to the result. We find these conflict roots by walking the tree upwards from the affected node until the first conflict root is encountered. For more details, refer to algorithms 4.6 and 4.7.

```
 1: function conflictUnitInitializer(base, treeA, treeB, cdgA, cdgB,
    conflictingChanges, conflictingPairs)
 2:     affectedCUsA ← computeAffectedCUs(cdgA)
 3:     affectedCUsB ← computeAffectedCUs(cdgB)
 4:     for all conflictRoots ∈ affectedCUsA.keys ∩ affectedCUsB.keys do
 5:         changesA ← affectedCUsA.values(conflictRoot)
 6:         changesB ← affectedCUsB.values(conflictRoot)
 7:         conflictingPairs.putAll(changesA × changesB)
 8:         conflictingChanges.putAll(changesA)
 9:         conflictingChanges.putAll(changesB)
10:     end
11:     findStructureConflicts(cdgA,cdgB)
12:     for all Move changes ∈ cdgA do
13:         checkMove(change, cdgB)
14:     end
15:     for all Move changes ∈ cdgB do
16:         checkMove(change, cdgA)
17:     end
18:     for all changes ∈ conflictingChanges do
19:         markDepending(change)
20:     end
21: end
```

Comment: `findStructureConflicts` detects child-structure-change conflicts and marks the appropriate changes as conflicting. `markDepending(change)` recursively marks all changes that depend on `change` as conflicting. It takes into consideration dependencies in the CDGs, label dependencies and move dependencies.

Algorithm 4.5: RUN method of the *Conflict Unit* pipeline initializer

```
1: function computeAffectedCUs(cdg)
2:     for all changes in cdg do
3:         if change.kind ≠ Insertion then
4:             conflictRootA ← findConflictRoot(change.nodeA)
5:             affectedCUs.put(conflictRootA, change)
6:         end
7:         if change.kind ≠ Delete then
8:             conflictRootB ← findConflictRoot(change.nodeB)
9:             affectedCUs.put(conflictRootB, change)
10:        end
11:    end
12:    return affectedCUs
13: end
```

Algorithm 4.6: computeAffectedCUs

```
1: function findConflictRoot(node)
2:     conflictRoot ← node
3:     while conflictRoot.type ∉ ConflictTypes do
4:         conflictRoot = conflictRoot.parent
5:     end
6:     return conflictRoot.id
7: end
```

Algorithm 4.7: findConflictRoot

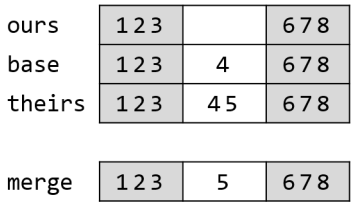


Figure 4.7: A *diff3 parse* and merged version of a list. Stable chunks are marked gray. The unstable chunk has been merged with our list merge algorithm by removing element 4 and inserting element 5.

4.2.4 The *List Merge* pipeline component

In practice, when detecting conflicts using the conflict-unit-based approach, conflicts often occur on lists (e.g. both branches adding methods to a method list). Most of the time resolving such conflicts is trivial and tedious for the user. We try to avoid this by attempting to resolve such conflicts automatically with a pipeline component which we will now present. Note that as this is a pipeline component, its use is optional and the *AST Merge* algorithm will produce a valid tree regardless of whether or not it is used.

One of the most used merging algorithms for ordered lists is *diff3* (or slight variations thereof). Envision gives us the advantage that list elements are uniquely identified by their node ID which allows us to specialize *diff3*. Khanna et al. investigated the behavior of *diff3* formally and introduced some nomenclature we are going to use as well, namely we are also computing a *diff3 parse* which considers two branch versions and a base version of a list to partition it into *stable* and *unstable chunks* [4]. Stable chunks are segments of the list that are identical in all three versions. Unstable chunks are segments where at least one branch deviates from the others. The *diff3 parse* of a list is then the series of these chunks, always alternating from stable to unstable to stable chunks. The goal is to find a merged version of the unstable chunks.

This component is configured with a set ConflictTypes $\subseteq$ NodeTypes. In our implementation, this is the same set also passed to the conflict unit component but in principle, this could be a different set. The configuration also includes two sets ListTypes $\subseteq$ ConflictTypes and UnorderedTypes $\subseteq$ ConflictTypes such that ListTypes $\cap$ UnorderedTypes $= \emptyset$. ListTypes is the set of node types that behave like ordered lists. The types of statement lists and argument lists would typically be such list types. UnorderedTypes is the set of node types that behave like collections or unordered lists. These could

for example be types of class lists or method lists. Ultimately, these sets are determined by the programming language and the choice of the ConflictTypes set. The labels of the children of such nodes are integers.

The algorithm for ordered lists

We select the lists to which to apply the list merge algorithm by finding those list container nodes for which both branches change the child structure indicating that both branches make changes to the list itself. We then check the following conditions and only apply the algorithm if all hold true.

1. The only way the branches change the list container node is changing the child structure. The branches do not move it or make internal changes to it.

2. All list elements are of a type in ConflictTypes.

3. If there is a conflict pair where both changes affect the same list element, one change must only change the label and in all conflict pairs the other change is part of, the paired change must affect a sibling (in either version) of the element. Formally, for every list element $elem$:

$$\forall (ch_A, ch_B) \in conflictPairs : ch_A.id = ch_B.id = elem.id \rightarrow$$

$$(\text{onlyLabel}(ch_A) \land \forall (ch'_A, ch_B) \in conflictPairs :$$

$$\text{affectsParent}(ch'_A, listContainer)$$

$$\lor$$

$$\text{onlyLabel}(ch_B) \land \forall (ch_A, ch'_B) \in conflictPairs :$$

$$\text{affectsParent}(ch'_B, listContainer))$$

   where onlyLabel($change$) is true if and only if $change.kind = \mathsf{Stationary}$ and all flags but the $labelFlag$ are False, $listContainer$ is the node that holds the list and

$$\text{affectsParent}(change, container) \equiv$$

$$change.nodeA.parentId = container.id \lor$$

$$change.nodeB.parentId = container.id$$

For every selected list we compute the *diff3 parse* from the different list versions L_A, L_B and L_{base}. Then we iterate over all unstable chunks of all lists and for each one attempt to build a merged version.

We do so by considering each element that occurs in version A or B of the chunk and trying to insert it into the merged chunk. Such an insertion succeeds if the following two conditions are met:

- A unique best position for the element is found.
- The other branch makes no conflicting change.

Such a unique best position is a position where the *immediate* predecessor and successor of the inserted element are each also a predecessor and successor respectively in the branch version of the chunk. See the algorithms 4.9 and 4.10 for details. If we detect a conflict during the merging of a chunk, we mark the chunk and all depending chunks as conflicting (more on chunk dependencies below).

Once we have computed a merged version for every unstable chunk (or detected a conflict), we use the chunks to assemble merged versions of the lists. We use the merged versions of conflict-free chunks and the base versions of conflicting chunks.

After building these merged versions of the lists, we modify the change objects in the CDGs to represent the transformation of the base versions into the merged versions. We also update the conflictingChanges and conflictPairs sets to represent the resolution of the conflicts.

Note about algorithm 4.8

We do not list implementations for all subprocedures used in the list merge algorithm and instead describe them here. `computeListsToMerge()` selects the lists that should be merged according to the conditions specified in section 4.2.4, page 49. `conflictingDependencies(chunk)` is true if `chunk` depends on some other chunk that is marked as conflicting. `markDepending(chunk)` recursively marks all chunks that depend on `chunk` as conflicting.
`translateIntoChanges(list)` modifies the CDGs, conflict pairs etc. to reflect the computed merged list and the resolution of conflicts.

Dependencies between chunks

Consider a change that moves a list element from one chunk to another chunk of the same or a different list. We can only apply this change if both the origin and the destination chunk are not conflicting. From this, some

dependencies between chunks arise. A conflict in one chunk could be the reason why another chunk must be marked as conflicting as well even if that other chunk would otherwise be conflict-free.

The algorithm for unordered lists

The algorithm for unordered lists differs just slightly. If both branches alter an element, they are only conflicting if they do not agree on the parent of the element. If no unique best position can be found, elements are inserted such that their immediate predecessor in the merged version is also a predecessor in one of the branch versions. If no such predecessor exists, elements are inserted at the beginning of the list.

```
 1: listsToMerge ← computeListsToMerge( )
 2: for all listsToMerge do
 3:     chunks ← computeDiff3Parse(list)
 4: end
 5: conflictingChunks ← ∅
 6: for all unstable chunks do
 7:     for all elem ∈ chunk.A do
 8:         posA ← findPosition(elem, chunk.A, chunk.merged)
 9:         posB ← findPosition(elem, chunk.B, chunk.merged)
10:         action ← computeMergeAction(cdgA, elem, chunk)
11:         if action = Conflict then
12:             conflictingChunks.insert(chunk)
13:         else if action = Insert then
14:             if posA = NULL ∨ conflictingDependencies(chunk) then
15:                 conflictingChunks.insert(chunk)
16:             else
17:                 chunk.merged.insert(elem, posA)
18:             end
19:         else
20:                                                  ▷ action = DoNothing.
21:         end
22:     end
23:     do the same symmetrically for elements in chunk.B
24:     if chunk ∈ conflictingChunks then
25:         markDepending(chunk)
26:     end
27: end
28: for all listsToMerge do
29:     mergedList ← [ ]
30:     for all chunks of list do
31:         if chunk is conflict-free or stable then
32:             mergedList.append(chunk.merged)
33:         else
34:             mergedList.append(chunk.base)
35:         end
36:     end
37:     translateIntoChanges(mergedList)
38: end
```

Comment: For details about used subprocedures with no listed implementation, refer to the note in section 4.2.4, page 50.

Algorithm 4.8: The list merge algorithm

```
 1: function findPosition(element, branchChunk, mergedChunk)
 2:     pred ← branchChunk.pred(element)
 3:     while pred ∉ mergedChunk ∧ pred ≠ branchChunk.before do
 4:         pred ← branchChunk.pred(pred)
 5:     end
 6:     succ ← branchChunk.succ(element)
 7:     while succ∉ mergedChunk ∧ succ ≠ branchChunk.after do
 8:         succ ← branchChunk.succ(succ)
 9:     end
10:     predInMerged ← mergedChunk.find(pred)
11:     if mergedChunk.succ(predInMerged) = succ then
12:         return predInMerged
13:     else
14:         return NULL
15:     end
16: end
```

Comment: `chunk.pred(element)` and `chunk.succ(element)` return the predecessor and successor of `element` in chunk.

Algorithm 4.9: findPosition

```
1: function computeMergeAction(cdgA, element, chunk)
2:     changeA ← cdgA.find(element.id)
3:     if changeA.kind = Insertion then
4:         return Insert
5:     else if element ∈ chunk.merged then
6:         return DoNothing
7:     else if alters(cdgA, element) ∧ alters(cdgB, element) then
8:         if posA = posB then
9:             return Insert
10:        else
11:            return Conflict
12:        end
13:    else if alters(cdgB, element) then
14:        return DoNothing
15:    else                ▷ This branch or neither branch alters the element.
16:        return Insert
17:    end
18: end
```

Comment: `alters(cdg,element)` is true if `cdg` contains a change that deletes the element, moves it to another parent or chunk or if its position relative to any other element in the chunk is different than in the base version.

Algorithm 4.10: computeMergeAction

Chapter 5

Correctness of the *AST Merge* algorithm

5.1 Formal interface of the *AST Merge* algorithm

Let T_A and T_B be the two AST of the respective branches A and B that shall be merged and let T_{base} be the AST in the state of the closest common ancestor commit of A and B. The algorithm takes the trees T_A, T_B and T_{base} as input and produces the following outputs:

- An AST T_{merged} that is the merged version of T_A and T_B.
- A set of changes $conflictingChanges$. These changes are part of conflicts that could not be resolved and were not applied to T_{merged}.

The postcondition of the algorithm is the following:

1. T_{merged} is of valid form.

5.2 Assumptions about inputs and configuration

- T_A, T_B and T_{base} are of valid form.
- The IDs of newly inserted nodes are unique. No node with the same ID exists and no node will be created with the same ID.
- The pipeline initializer and all pipeline components comply with their respective interfaces.

- ListTypes $\cap$ UnorderedTypes $= \emptyset$

5.3 Correctness of the generic algorithm

We remind ourselves of the preconditions of the pipeline initializer which the *AST Merge* algorithm must establish before executing the initializer.

1. T_{base}, T_A and T_B are of valid form.
2. Applying all and only changes from cdg_A to T_{base} results in T_A.
3. Applying all and only changes from cdg_B to T_{base} results in T_B.
4. Both sets $conflictingChanges$ and $conflictPairs$ are empty.
5. cdg_A and cdg_B are acyclic

We show that the *AST Merge* algorithm establishes all of these preconditions. Since the trees are not modified before executing the initializer, precondition 1 directly follows form the assumptions in the previous section. We initially build the CDGs with all changes computed by the *AST Diff* algorithm, hence preconditions 2 and 3 are satisfied. We initialize $conflictingChanges$ and $conflictPairs$ as empty sets, therefore precondition 4 is also satisfied. The CDGs are initialized such that precondition 5 is satisfied (see proof in section 4.1.1).

All preconditions of the pipeline initializer are satisfied so it establishes the pipeline invariants which we have defined as follows where $nonconflicting = cdg_A.changes \cup cdg_B.changes - conflictingChanges$ and T_{merged} is the tree that results from applying all changes in $nonconflicting$ to T_{base}.

1. If sorted topologically by dependencies, all changes in $nonconflicting$ can be applied.
2. T_{merged} is of valid form.
3. Node IDs are unique in T_A and T_B.
4. All conflicting changes are either part of a conflict pair or depend on a conflicting change.
5. cdg_A and cdg_B are acyclic.

With these invariants established, we can execute the pipeline wherein every component establishes the invariants for the next one. After the pipeline has been executed, we build the set $nonconflicting$ and order the changes topologically according to their dependencies. We know such a topological ordering exists from invariant 5. By invariant 1, we can apply all changes in $nonconflicting$ to a copy of T_{base} which, by pipeline invariant 2, results in a tree T_{merged} of valid form. With this, we have shown that the postcondition of the *AST Merge* algorithm is satisfied.

5.4 Extended validation of components

We would like to specify properties that the tree resulting from our merge algorithm has. Because we support generic pipeline components, we would have to specify an interface for them that is strong enough to eventually yield the properties specified for the merge algorithm as a whole. We found that every interface we considered either rendered some hypothetical but desirable and reasonable component impossible or simply failed to provide interesting properties. It seems that a definition that is satisfactory in general, if it exists, is prohibitively complex. This is further complicated by the fact that our merge algorithm is language agnostic.

We decided instead to create a framework that would allow us to define and check properties of individual component executions. Concretely, we define the states before and after the component execution and a transition semantics that relates the two states. A state is described by a partitioning of all changes. We call such partitions *linked changes*. Changes in the same partition are considered linked in the sense that they have some arbitrary relation to each other (e.g. dependency, conflict, etc.). A *state transition* is a mapping of the partitions of the old state to the partitions of the new state. This mapping is surjective i.e. for every partition in the new state, there exists at least one partition in the old state that is mapped to it.

An example of such a transition is shown in figure 5.1. The red squares represent changes, the linked changes are in green and the tall rectangles are the set of all changes in each state. The arrows indicate the mapping of the transition. This transition could be interpreted as follows. Changes 1 and 2 are not modified. Change 3 is replaced with change 7. Change 4 is related to change 5 and/or 6 and together they prompted change 8 to be created.

The idea is that a component outputs such a transition, relating its input state to its output state. One can think of the transition as a record and justification of any creation, deletion or modification of changes. A checker operating on such a transition can then judge if all modifications made by

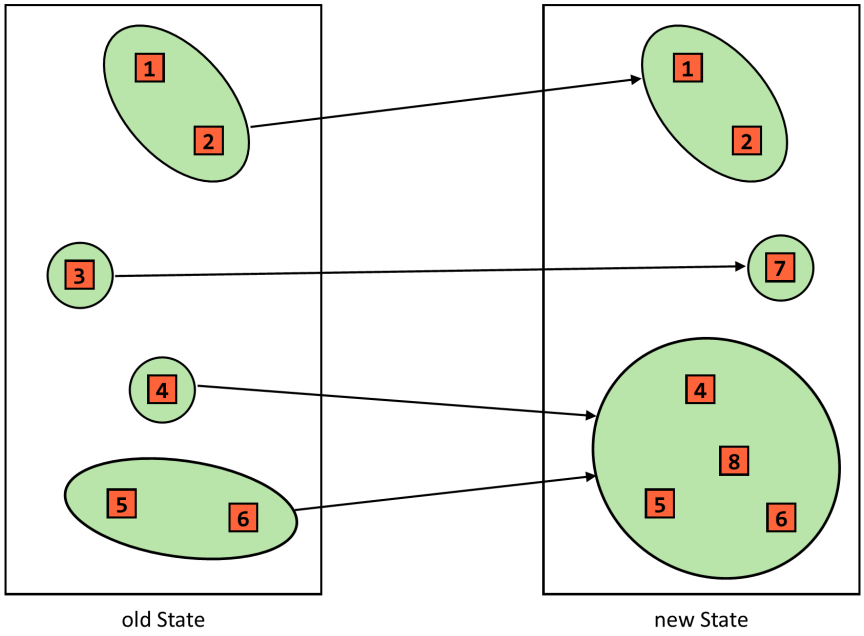


Figure 5.1: Related changes and transition

the component are reasonable. In principle, one is free to apply any number of checkers on any number of transitions so for example one could have one checker that checks all components in the pipeline and another that only checks the transition of one specific component.

We give an example to illustrate how this framework could be used to check a certain property. Assume we would like to restrict one or several components so that they may only modify a previously unchanged node if its parent, a sibling or a child is changed. We would then implement a checker that, for all such new changes, does the following check: Let *newChange* be the change in question and let *newState* and *oldState* be the new and old states. Let *newSet* be the set of linked changes such that $newChange \in newSet \in newState$. Then it must hold that $\exists oldChange \in oldSet \in oldState$ where *oldChange* modifies the parent, a sibling or a child of the node modified by *newChange*.

5.5 Correctness of the conflict unit pipeline initializer

We argue that when the conflict unit pipeline initializer returns, the pipeline invariants are established.

5.5.1 Pipeline invariants 1 and 2

We consider an arbitrary change $changeA$ of branch A that is in $nonconflicting$ (i.e. $changeA \notin conflictingChanges$) when the component returns and show that it can be applied to T_{merge} (which starts out as T_{base}) and that its application does not violate valid form. The argument for a change of branch B is symmetric.

This proof works by induction over the change dependency graph which is acyclic, as we have shown in section 4.1.1. We assume that all changes that $changeA$ depends on (i.e. all changes that can be reached from $changeA$ in the CDG) can be applied and that their application does not violate valid form. For the base case where $changeA$ does not depend on any other changes, it is trivial that this assumption is valid so the form of the proof for the base case is identical. We make a case distinction on the kind of $changeA$.

Case Insertion:

If $changeA$ is of kind Insertion, a parent node $parent \in T_{merge} : parent.id = changeA.nodeB.parentId$ must exist to apply it.

If $parent$ exists in T_{base} then it also exists in T_{merge} at the time we apply $changeA$ unless branch B deletes $parent$ with a $changeB$. $changeA$ is a child structure change of $parent$, therefore there is a change $parentChangeA$ of branch A that affects $parent$ and thus ConflictRoot($parent$). $changeB$ deletes $parent$ and therefore also affects ConflictRoot($parent$) which means $changeB$ gets marked as conflicting so $parent$ is not deleted and $changeA$ can be applied.

If $parent$ does not exist in T_{base} then, from the fact that T_A is of valid form, we know that branch A inserts $parent$ into T_{merge} with a $parentChangeA$ which $changeA$ depends on. By the induction assumption $parentChangeA$ can be applied. Since $parentChangeA$ comes before $changeA$ in a topological ordering, we know that when we apply $changeA$, $parentChangeA$ has already been applied, therefore $parent$ exists in T_{merge} and $changeA$ can be applied.

Because T_A is of valid form and the ID of $changeA.nodeB$ is new and therefore unknown to branch B it follows that the ID of $changeA.nodeB$ is unique in T_{merge}.

It remains to show that applying $changeA$ does not result in $parent$ having two children with the same label.

In the case where $\nexists node \in T_{base} : node.parentId = parent.id \wedge node.label = changeA.nodeB.label$, the only way to get a label collision is if branch B changes a node with a $changeB$ in such a way that $changeB.nodeB.parentId = parent.id \wedge changeB.nodeB.label = changeA.nodeB.label$. Since no such

node existed in T_{base}, it follows that $changeB.kind \neq \mathsf{Stationary} \vee$ $changeB.labelFlag = \mathsf{True}$. Both $changeA$ and $changeB$ are child-structure-changes of $parent$ thus there exist changes $parentChangeA$ of branch A and $parentChangeB$ of branch B that both affect $parent$ and have $structFlag$ set. Then from the definition of child-structure-change conflicts of section 4.2.3 follows $(changeA, changeB) \in conflictPairs$ which is in contradiction to $changeA$ being in $nonconflicting$. Therefore branch B makes no such change and no label collision occurs.

This leaves the case where $\exists node \in T_{base} : node.parentId = parent.id \wedge$ $node.label = changeA.nodeB.label$. Since T_A is of valid form, we know that branch A changes $node$ with a $changeA' : changeA'.nodeA = node$ to change its parent or label. The following holds.

$$changeA'.nodeA.parentId = changeA.nodeB.parentId \wedge$$

$$changeA'.nodeA.label = changeA.nodeB.label$$

From this it follows that $changeA$ depends on $changeA'$ in the sense of label-dependencies explained in section 4.2.3. Assume $changeA' \notin nonconflicting$. Therefore $changeA' \in conflictingChanges$ which means that MARKDEPENDING($changeA'$) was executed and all changes depending on $changeA'$ in the sense of label-dependencies have been added to $conflictingChanges$. This includes $changeA$ thus $changeA \notin conflictingChanges$ which leads to a contradiction and therefore the assumption $changeA' \notin nonconflicting$ was wrong. Both $changeA$ and $changeA'$ are in $nonconflicting$ and are applied thus a label collision in T_{merge} is averted.

In all other cases $changeA.nodeA$ must exists in T_{merge} in order to apply $changeA$.

Case Deletion:

$changeA.nodeA$ must exists in T_{merge} in order to apply $changeA$. $changeA.nodeA$ exists unless branch B deletes it with a $changeB$. In this case both changes affect ConflictRoot($changeA.nodeA$) which leads to a contradiction, therefore $changeA.nodeA$ exists. If $changeA$ is of kind Deletion then to apply it, $changeA.nodeA$ must not have any children. If $changeA.nodeA$ has any children in T_{base} then, from the fact that T_A is of valid form, we know that branch A deletes or moves away all children of $changeA.nodeA$. $changeA$ is dependent on all these deletions and moves therefore these will all have been applied when $changeA$ is applied. Therefore $changeA.nodeA$ does not have any children unless branch B inserts or moves a child under $changeA.nodeA$ with a $changeB$. $changeB$ is a

child structure change of $changeA.nodeA$ and therefore branch B also affects ConflictRoot($changeA.nodeA$) which leads to a contradiction.

Case Stationary:

By the same argument as in the Deletion case, $changeA.nodeA$ exists in T_{merge} therefore $changeA$ can be applied. For the case where $changeA.labelFlag$ is set, we have to show that $changeA$ does not result in a label collision. The argument for this is analogous to the Insertion case.

Case Move:

If $changeA$ is of kind Move, in addition to the conditions described for the Stationary case (for which the proof is identical), the new parent of the moved node must exist. The arguments why this parent must exist and why no label collisions occur are analogous to the Insertion case.

With this, we have shown that pipeline invariants 1 and 2 are established by the conflict unit component.

5.5.2 Pipeline invariants 3, 4 and 5

By assumption, T_A and T_B are of valid form before the component is executed. Valid form implies uniqueness of IDs and since we do not modify T_A or T_B it follows that invariant 3 holds when the component returns. To see that invariant 4 is established we note that whenever we add a change to $conflictingChanges$, we either also add a pair containing that change to $conflictPairs$ or the change depends on another change in $conflictingChanges$. By only adding changes in this manner, we never violate invariant 4 and it still holds when the component returns. According to precondition 5 of the pipeline initializer, invariant 5 holds just before the component is executed. Since we do not modify the CDGs, it follows that invariant 5 also holds when the component returns. □

Chapter 6

Evaluation

In this chapter we highlight the advantages of some of the features of our algorithms when compared to a traditional line-based version control system like Git. We also revisit a case for which the previous implementation did not produce the correct output and illustrate how the new implementation addresses the issue.

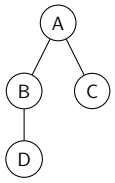
6.1 Corrected *AST Diff* algorithm

This is an example that illustrates the issues briefly described at the beginning of the *AST Diff* chapter (ch.3) which caused incomplete results in the old implementation. Consider the situation illustrated in figure 6.1 where the labels of nodes B and C are switched and node D is moved from B to C. Notice how the node line of D is not modified and even stays at the exact same position. When calling `git diff` on these two versions, this node line will be regarded as unchanged and will not be part of the line diff. This has the result that the fact that node D is moved to a new parent is not detected by the old *AST Diff* algorithm.

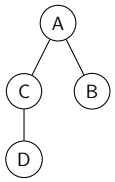
Compare now the same situation with the new encoding and algorithm as seen in figure 6.2. All lines except the root have changed and are therefore included in the line diff. The move of node D is detected.

6.2 Comparing the *AST Merge* algorithm to a text-based approach

This section explores different merge scenarios and examines how a traditional text-based VCS compares to Envision's solution. Concretely, we com-




```
root ListOfFoo {0ec91...be77e}
   0 Foo {f4d3b...dbf9b}
      value Name {fab00...4b6e3}. S_D
   1 Foo {6eeed...13b49}
```




```
root ListOfFoo {0ec91...be77e}
   0 Foo {6eeed...13b49}
      value Name {fab00...4b6e3}. S_D
   1 Foo {f4d3b...dbf9b}
```

Figure 6.1: An AST evolution that lead to a incomplete *diff* output.

```
root ListOfFoo {0ec91...be77e} {00000...00000}
   0 Foo {f4d3b...dbf9b} {0ec91...be77e}
      value Name {fab00...4b6e3} {f4d3b...dbf9b}. S_D
   1 Foo {6eeed...13b49} {0ec91...be77e}
```

```
root ListOfFoo {0ec91...be77e} {00000...00000}
   0 Foo {6eeed...13b49} {0ec91...be77e}
      value Name {fab00...4b6e3} {6eeed...13b49}. S_D
   1 Foo {f4d3b...dbf9b} {0ec91...be77e}
```

Figure 6.2: The same AST evolution with the new encoding.

pare our *AST Merge* algorithm in the configuration of using both the presented pipeline initializer and list merge component with Git[1], one of the most popular version control systems.

6.2.1 Relocating a class

Consider a class `ClassBar` with a few fields and methods that is written in a file `myClasses` containing other classes. Now consider the situation where branch A moves `ClassBar` from its original file into its own file `classBar`. At the same time, branch B makes a few changes to a method of `ClassBar`.

Git regards the changes of branch A as the deletion of many lines in `myClasses` and a creation of a new file `classBar` and the changes of branch B as a few modifications in `myClasses`. The changes of branch B affect some of the same lines that are deleted by branch A and so a conflict is detected in the file `myClasses`. The user has to manually apply the changes of branch B to the code in file `classBar`.

Envision's VCS handles the same situation differently. First of all, projects are split across multiple files implicitly and transparently. It is likely that a change like the one performed by branch A will never be necessary in the first place. Only if branch A makes an actual semantic change like moving `ClassBar` into a different package will this be a change for Envision. In this case, the change of branch A is simply a Move operation on the node of `ClassBar`. Assuming method nodes are conflict roots, this change is not in conflict with the changes of branch B and the changes of both branches are applied in the final merged version without additional input from the user.

6.2.2 Conflict-unit-based conflict detection

Consider the situation illustrated by listings 6.1 through 6.3 where both branches fix a buggy increment of the loop variable. Git resolves this conflict by applying both changes, resulting in the loop variable getting updated twice as seen in listing 6.4. The automatic merging algorithm has created a bug that is not detected at compile time but only by testing or manual inspection.

Now consider the same situation in Envision where the AST node representing the while-loop and the statements themselves are conflict roots. Both branches move the increment and thus make changes that affect the child structure of the loop body so during execution of the conflict unit component, these changes are detected as conflicting. The list merge algorithm

[1] `git-scm.com`

detects that both branches moved the element into different positions and does not resolve the conflict and so the user is notified who then merges the fixes manually without causing a bug.

Listing 6.1: Base version

```
while (i < n) {
        i++;
        sum += arr[i];
        prod *= arr[j];
        j++;
}
```

Listing 6.2: Branch A

```
while (i < n) {
    sum += arr[i];
    prod *= arr[j];
    j++;
    i++;
}
```

Listing 6.3: Branch B

```
while (i < n) {
    sum += arr[i];
    prod *= arr[j];
    i++;
    j++;
}
```

Listing 6.4: Git merge result

```
while (i < n) {
    sum += arr[i];
    prod *= arr[j];
    i++;
    j++;
    i++;
}
```

6.2.3 Concurrent modification of unordered lists

Consider a language like Java where the order of method definitions does not matter. We examine the scenario where both branches add a method to the same class and at the same position. When merging, Git detects a conflict and the user has to manually merge the file and make sure that both methods get included correctly.

Envision regards the same scenario as two insertions of independent nodes into the same unordered list. Assuming the methods are conflict roots, the

only conflict detected by Envision are the two changes of the child structure of the method list. This conflict can be resolved by the list merge component so that both new methods are inserted correctly in the final merged tree.

6.2.4 Resolving conflicts in ordered lists

In the scenario illustrated by listings 6.5 through 6.7, branch A removes the line `foo(bar)` while branch B removes the same line, moves the line `foobar(bar)` to that location and adds a new line `baz()`. When attempting to merge with Git, a conflict is detected, even though all changes of branch A are also made by branch B.

In Envision, the move of the `foobar(bar)` statement is detected as such and during execution of the list merge component these conflicts are resolved. Both branches agree on the deletion of the `foo(bar)` statement and for both statements `foobar(bar)` and `baz()` a unique best position can be found. The merge result of Envision is equivalent to the version of branch B.

Listing 6.5: Base version

```
/*
 * some code
 */
foo(bar);
/*
 * more code
 */
foobar(bar);
/*
 * even more code
 */
```

Listing 6.6: Branch A

```
/*
 * some code
 */
/*
 * more code
 */
foobar(bar);
/*
 * even more code
 */
```

Listing 6.7: Branch B

```
/*
 * some code
 */
foobar(bar);
baz();
/*
 * more code
 */
/*
 * even more code
 */
```

Chapter 7

Implementation Details

This chapter aims to explain some of the aspects of the implementation that might not be obvious on first sight. We try to shed light on certain implementation details and explain some of the decisions made. Developers extending, debugging or refactoring the code should read this section and have a thorough understanding of the discussed systems before making changes.

7.1 Lazy-loading nodes in change objects

We generally want to avoid loading nodes if not necessary. This is why we do not load the whole trees during the diff but work only from the node lines supplied by Git. It is possible however that we want to create a change object to mark a child structure change for a node whose node line is not modified and therefore never passed by Git. Only the ID of the affected node is known. We could just always use a `git grep` call to load the node but this is not cheap and it is possible that the loaded node is not even used after. Where possible, we avoid loading the node immediately by storing a pointer to the child node (whose change has caused the child structure change in the first place) instead. When the affected node is actually needed at some point in time, it can be loaded by getting the parent of the linked child.

7.2 Maintaining tree invariants

`GenericTree` objects have some invariants and its worth mentioning when and where some of them are established and checked. Consider the following invariants. (1) In a tree that has a `quickLookupHash`, every loaded node is contained therein. (2) Within a tree, node IDs are unique. (3) Every loaded

node is linked (i.e. has valid pointers in its `parent` `children` members) to all appropriate loaded nodes.

These invariants are not established by the `GenericTree` alone but also by anyone creating new nodes in the tree. Whenever an entity creates a new node, it is responsible to call `linkNode` on that node, which then establishes the mentioned invariants. Removing nodes should always be done by calling `GenericTree::remove` and not `GenericNode::remove` as only the former will maintain invariant (1) from above.

The reason for this is that currently the only way to get the id of a node line is to create the node and then read from it. This requires the ability to create a node and discard it again if its ID already exists in the tree. This also allows to more easily ignore these invariants in cases where the `GenericTree` is purely used to allocate node objects without unique IDs and no links between them.

7.3 Details of the list merge component

7.3.1 Propagation of conflicts for chunks

Consider a pair of chunks that depend on each other. When computing the first chunk, we do not yet know the conflict state of the other. We take an optimistic approach and assume the other chunk resolves without conflicts but we record the fact that the current chunk being conflict-free depends on the other chunk being conflict-free. We do so with the mapping `chunkDependencies` which maps a given chunk to the set of chunks that have to be marked as conflicting should the given chunk get marked as conflicting. This need to mark chunks as conflicting retroactively is the reason why we first compute all chunks of all lists before beginning to merge them.

7.3.2 Translating ID lists into changes and marking conflicts as resolved

Once we have computed all merged lists, we need to translate these results back into the context of changes, CDGs and conflict pairs. To do so we iterate over the merged lists and, if one exists, modify an existing change of one of the branches (and mark its conflicts as resolved) or create a new change. This works well for all changes except for deletions. Intuitively, one would think that we could simply check what elements exist in the original lists but not in the merged lists. This however does not take into account that merged lists also do not include elements of conflicting chunks. We

have to be able to distinguish whether an element is not present in a list because it was deleted or because its chunks had a conflict. For this reason, we handle deletions already when creating the merged lists from the chunks. Only deletions in conflict-free chunks are marked as resolved.

7.4 Related changes and transitions

Transitions relate an earlier state to a later state. Because `ChangeDescription` objects are mutable it is necessary to always create a copy of the current state before executing a component. This not only includes copying the `ChangeDescription` objects but also the `GenericNode` objects it points to. A copy of these requires a `GenericTree` as an allocation manager. To avoid creating multiple such trees for each transition, we only create one but ignore the invariants that are usually desired for the tree. This allows us to allocate all necessary nodes in the same tree while also not requiring the contained nodes to be linked. This is an important consideration when implementing a component checker.

Chapter 8

Future Work

8.1 Partially completed work

The original plan was to produce a proof of correctness for the *AST Merge* algorithm and its components in the same style as the one we presented for the *AST Diff* algorithm. Unfortunately, we could not get as formal as we wished. We believe however that the presented argument for correctness, in addition to its inherent value, provides some intuition as to how a more formal proof would work.

We also wish we had more time to further evaluate the designed system. A more in-depth evaluation could reveal less obvious advantages to traditional systems or highlight where there is room for improvement.

The conceptual work on extended component validation was finished relatively late into the project. For this reason there was insufficient time left to implement linked changes and transitions for the list merge component.

8.2 Applying the presented principles outside of Envision

We would like to make it possible to apply the principles of Envision's VCS, especially the merge algorithm, to projects not written in Envision. Envision already features an import feature for certain languages that translates textual source code into the tree-based representation our VCS works with. Unfortunately, this feature cannot be used directly to import revision histories from traditional VCSs because during the import all AST nodes are created with unique IDs that have no relation to other revisions in the history.

A more advanced translation functionality would have to be developed

that matches up nodes from the ASTs of the different versions and assigns matched nodes the same ID. Both Falleri [2] and Hashimoto [3] have proposed good algorithms to find such a matching between trees. Once such a translation functionality has been implemented, all the presented principles can be used and integrated with traditional VCSs.

8.3 Visualization of merges

Often users want to review the result of a merge. To help the user understand the changes made by both branches and how they were merged, most traditional version control systems resort to showing the diffs comparing the merged version to its ancestors, possibly relating these diffs to each other to further help the user. A great addition to Envision's VCS would be a user interface serving a similar purpose. Thanks to the semantic information in change objects and the visualizations available in Envision, it should be possible to convey more valuable information more clearly. A good user interface would help a great deal in improving the usability of Envision's VCS.

8.4 User interfaces for manual conflict resolution

Even the best merge algorithm cannot resolve all conflicts. This is when the user has to be asked for input. Ideally this input only consists of a choice between multiple good solutions and an optional set of corrections. To give this input, the user has to understand the context of the choices. Future work could explore how to present existing information to the user and how the user makes their choice to create an efficient work flow. We believe this work will be closely related to how merges are visualized.

These explorations would also reveal clearly how the *AST Merge* algorithm should proceed in cases where not all conflicts could be resolved. Currently the conflicting changes are returned without context which is most likely not satisfactory. Especially interesting would be to see if the issue of manual conflict resolution can be solved only with pipeline components.

8.5 Additional Components

With our work we have created a solid foundation on which future contributors to Envision can build on. We have some ideas for future components

and would like to list a few here.

- Cases where both branches make the same change like deleting the same node are seen as conflicting in the current implementation. Such conflicts could be resolved by a component that finds such cases and simply removes one of the changes.

- Renaming entities can cause problems after merging. If a branch renames an entity and is later merged with a branch that also makes changes involving this entity, then the code of this latter branch might refer to the entity by its old, now non-existing name. Such conflicts are not detected by traditional version control systems and often only surface at compile time. Since in Envision, entities are not only referenced by name but also include the ID of the AST node that declares the referenced entity, it is possible to detect an action of renaming as such. This allows the detection and subsequent resolution of conflicts caused by renaming.

- Consider the situation where one branch makes a small change in a method and the other branch deletes the whole method. A reasonable way to resolve this conflict would be to rate the small change inside the method as insignificant compared to the deletion of the entire method. It would be interesting to see a component that implements such a policy that resolves conflicts by ranking some change as more important and discarding conflicting changes.

- One could implement a simpler pipeline initializer that does not make use of conflict units and is agnostic of the language model. This would then make it possible to use the *AST Merge* algorithm even if no ConflictTypes set has been specified.

- There is no reason why components should be completely autonomous. Considering conflicts that can be automatically resolved in multiple ways for which the component cannot decide which solution should be realized, getting user input during component execution could prove very useful.

8.6 Component Validation

We would like to see some work that extends or makes use of the validation framework we have built for components. Even if it is just a proof-of-concept, we would like to see a working component checker. Building such a checker

could reveal strengths and weaknesses of the validation framework. We have proposed such a checker in section 5.4.

Chapter 9

Conclusion

Martin Otth designed the core of a version control system for Envision [6]. We have built on his work to improve some key aspects. The here presented *AST Diff* algorithm is an only slightly modified version of the original proposal. We have precisely defined what the differences between two ASTs are and have shown that our algorithm conforms to this definition and produces the expected output. This proof of correctness gives us great confidence in our model.

Doing a similar investigation of the merge algorithm proposed by Otth proved to be much more involved. It became clear that in order to arrive at an algorithm we could reason about we would have to completely overhaul the merge algorithm. This gave us the opportunity to create a design in which plug-in behavior is a core feature. The *AST Merge* algorithm we have presented provides a modular approach to merging that can be configured for any number of possible behaviors and policies. We have defined an interface for merge components that, if observed, provides meaningful and useful guarantees for the resulting merged AST.

Going beyond these general properties of validity, with linked changes and transitions we have proposed a framework enabling the validation of components. Much in the same spirit as the *AST Merge* algorithm itself, this framework focuses on modularity and extensibility. This also averts the issue of imposing an interface on the components that is too strict which would prevent the implementation useful components.

We have presented a few examples of merge scenarios that highlight several advantages of our system over a traditional, text-based solution. Envision's version control system detects more real conflicts and avoids some false positives. Additionally, our algorithm for resolving conflicts in lists further reduces the manual work required from the user.

Bibliography

[1] D. Asenov and P. Müller. Envision: A fast and flexible visual code editor with fluid interactions (overview). In *Visual Languages and Human-Centric Computing (VL/HCC)*, pages 9–12, 2014.

[2] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Montperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 313–324. ACM, 2014.

[3] Masatomo Hashimoto and Akira Mori. Diff/ts: A tool for fine-grained structural change analysis. In *Reverse Engineering, 2008. WCRE'08. 15th Working Conference on*, pages 279–288. IEEE, 2008.

[4] Sanjeev Khanna, Keshav Kunal, and Benjamin C Pierce. A formal investigation of diff3. In *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science*, pages 485–496. Springer, 2007.

[5] Tancred Lindholm. A three-way merge for xml documents. In *Proceedings of the 2004 ACM symposium on Document engineering*, pages 1–10. ACM, 2004.

[6] Martin Otth. Fine-grained software version control based on a program's abstract syntax tree. Master thesis, ETH Zurich, 2014.

[7] Bernhard Westfechtel. Structure-oriented merging of revisions of software documents. In *Proceedings of the 3rd international workshop on Software configuration management*, pages 68–79. ACM, 1991.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

TREE-BASED VERSION CONTROL IN ENVISION

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
GUENAT	BALZ

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, 16.7.15	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.