Diploma Thesis

June 30, 2008

Improving ChangeDistiller

Improving Abstract Syntax Tree based Source Code Change Detection

Michael Würsch

of Zürich, Switzerland (01-701-754)

supervised by

Prof. Dr. Harald C. Gall

Beat Fluri; Christoph Kiefer

University of Zurich

Department of Informatics

s.e.a.l.

software evolution & architecture lab

Diploma Thesis

Improving ChangeDistiller

Improving Abstract Syntax Tree based Source Code Change Detection

Michael Würsch

University of Zurich
Department of Informatics

s.e.a.l.
software evolution & architecture lab

Diploma Thesis

Author: Michael Würsch, wuersch@ifi.unizh.ch

Project period: April 3, 2006 - October 3, 2006

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

First of all, I would like to thank the S.E.A.L.-team at the Department of Informatics at the University of Zürich, especially Prof. H. Gall for giving me the opportunity of writing this thesis, Beat Fluri as my supervising assistant, Martin Pinzger and Patrick Knab for their support and advice. They did not get tired of promptly answering the many questions that have arisen over the past six months of work on my thesis and showed me that work can also be a lot of fun. Furthermore, I thank Beat and Patrick for proofreading my thesis extensively and giving valuable feedback.

Next, I would like to thank my two colleagues, Andreas Jetter and Roman Flückiger. A problem shared is a problem halved!

Special thanks to my parents, Jutta and Arnold Würsch, for their support and for showing me that dinner can be quite healthy and relaxing after a hard day of work.

Last but definitely not least, I would like to thank my girlfriend Claudine for her patience and her love during the last six months.

Abstract

Changes are a crucial part of the life-cycle of modern software systems. Common versioning systems such as CVS store version histories of source code. Usually, they are not capable of tracking changes on a more sophisticated level. They provide lexical but not syntactical change analysis.

The existing Eclipse-plug-in ChangeDistiller bridges this gap by providing a sophisticated analysis of structural source code changes. It uses an abstract syntax tree (AST) representation of subsequent revisions of source code files and compares the trees by using a change detection algorithm for hierarchically structured information. The outcome is an edit script describing the operations that are necessary to transform the original version of the tree into the modified one.

We aim at improving the sub-algorithm responsible for matching trees. It yields insufficiencies in terms of matching leaves in general, it often produces sub-optimal results for small subtrees, and it is not able to handle large number of changes adequately. To overcome this issues, we propose customized similarity measures and a similarity ranking algorithm for leaves, as well as dynamic modulation of the tree similarity thresholds whenever small tree structures are encountered.

To prove our claims, we establish an extensive benchmark for investigating runtime performance and accuracy. The benchmark is based on the JUnit regression testing framework and relies on artificial source code examples, as well as on examples taken from a medium-sized real project.

Zusammenfassung

Änderungen sind ein wichtiger Bestandteil im Lebenszyklus eines jeden modernen Software Systems. Versionierungssysteme, wie zum Beispiel CVS, verwalten die Änderungen von Programmcode über Zeit. Werkzeuge zum Vergleich zweier Programmversionen basieren üblicherweise auf lexikalischen, also text-basierten, Methoden und sind nicht in der Lage strukturelle Änderungen zu erfassen.

ChangeDistiller, ein existierendes Plug-In für Eclipse, geht einen Schritt weiter, indem es eine umfassende Analyse von Änderungen in der Programmstruktur bereitstellt. Um dies zu bewerkstelligen, generiert das Werkzeug eine auf abstrakten Syntaxbäumen basierende Repräsentation von Programmcode. Die Bäume werden mittels eines auf Änderungen in hierarchisch-strukturierten Informationen spezialisierten Algorithmus verglichen. Das Resultat ist eine Menge von Änderungsoperationen, die die ursprügliche Programmversion in die modifizierte Fassung überführen.

Diese Diplomarbeit bezweckt eine Verbesserung des Teilalgorithmus, welcher Übereinstimmungen zwischen Bäumen lokalisiert. Der Algorithmus birgt einige Schwächen. Ähnlichkeiten zwischen Bättern der verglichenen Bäume werden nicht immer erkannt. Desweiteren werden meist suboptimale Resultate erzielt, wenn kleine Subbäume analysiert werden oder zu umfangreiche Änderungen zwischen zwei aufeinanderfolgenden Versionen von statten gegangen sind.

Um diesen Einschränkungen Herr zu werden, schlagen wir nun folgende Massnahmen vor: Zum Einen empfehlen wir neue, auf Programmcode spezialisierte, Metriken zur Ähnlichkeitsbestimmung. Zum Anderen haben wir den Algorithmus dahingehend angepasst, dass er eine auf Ähnlichkeiten basierende Rangfolge zwischen möglichen Paaren von Blättern berechnet. Um das Problem kleiner Subbäume zu mildern, verwenden wir eine dynamische Anpassung der Grenzwerte, welche darüber entscheiden, ob zwei Teilbäume in ausreichendem Masse Ähnlichkeiten aufweisen, um als Übereinstimmung gewertet zu werden.

Um die Auswirkungen unserer Massnahmen zu untersuchen, stellen wir umfangreiche Untersuchungen an. Wir testen das Laufzeitverhalten, sowie die Präzision unseres Algorithmus mittels dem JUnit-Framework für Regressionstests. Als Testgrundlage wählen wir eine Kombination aus konstruierten Testfällen und realen Beispielen, die einem Projekt mittlerer Grösse entnommen wurden.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1 Motivation

Since Lehman stated his *Laws of Program Evolution* in the 1980's [Leh80], in particular the *Law of Continuing Change*, it is well understood that software has to be adapted to changing requirements and environments or becomes progressively less useful. Hence, changes are broadly accepted as a crucial part of a software's life-cycle.

Admitting such an importance to changes and putting a focus on them during the software development process allows to manage software aging adequately. Nevertheless, many challenges remain: The above-mentioned awareness is only a sufficient precondition for mastering large-scale software engineering; The developer needs further assistance in applying and tracking changes in a complex software system consisting of *e.g.*, several millions lines of code.

If a software engineer modifies a certain source code entity, it is desirable to give him some further information about his work, for example whether the modification was functionality-modifying or functionality-preserving and to perform an impact analysis to tell if or even how other source code entities are affected.

A tool capable of delivering such insights can also be extended to guide programmers along related changes by telling them "Programmers who changed these functions also changed..." as mentioned in [ZWDZ04] and thus *suggest and predict likely changes*, *prevent errors due to incomplete changes* or *detect couplings undetectable by program analysis*. Furthermore an overall view of the changes, applied to a software system, can be used for quality assurance concerns.

To gain a deeper insight on source code changes, it is self-evident to take a closer look at version histories usually stored in common versioning systems such as CVS,[1] SUBVERSION,[2] or RATIONAL CLEARCASE.[3] Unfortunately, these systems do not track structural changes of source code, but rather rely on computing textual differences and are therefore only able to tell that a certain line was added, deleted, or—at best—moved between two versions of a file.

Since these systems are not natively capable of delivering us the information we are interested in, we are in need of algorithms to reconstruct the missing parts using common infrastructure. The differencing algorithm has to perform an comparison of two versions of a program, an *original* version and a *modified* version, to find atomic changes between these two versions. We want to detect the location of the changes within the source code entities as well as provide a mapping from the original version to the modified counterpart. Then, we classify the changes to tell that *e.g.*, a statement *s* has been added to—or removed from—a method *m* or that the identifier of a

[1] Concurrent Versions System - http://www.nongnu.org/cvs/
[2] http://http://subversion.tigris.org/
[3] http://www.ibm.com/software/awdtools/clearcase/

field *f* has been renamed from *a* to *b*. The results shall be used to assign a certain significance level to each change (for example: "This particular change has a very high significance level, since it effects the public interface of a class" or even "The method `A.foo()` experienced a series of changes with very high significance level between revisions 1.3 to 1.7") or to reveal code smells and change couplings.

CHANGEDISTILLER [FG06], our source code extraction plug-in for ECLIPSE,[4] already implements some of the above-mentioned functionality. It uses a tree differencing algorithm, developed by Chawathe *et al.* [CRGMW96] and finds tree edit operations between two intermediate abstract syntax trees (AST) based on a matching set of tree nodes. The outcome of the algorithm is an edit-script transforming the first into the second tree. The set of edit operations, which compose the edit script, reflects the changes that a developer has applied to transform the original program into the modified one.

The approach taken by Chawathe *et al.* yields some insufficiencies in terms of source code. They use a heuristic based on the assumption that changes between the two input trees are small. This is valid for source code in most cases, since changes between subsequent versions of a class are usually small. In return, this leads to non-minimal edit scripts if the amount of changes between the two versions of the Java source file exceeds a certain size. Matching on the leaf-level, *i.e.*, matching of single statements, is imprecise because of the implemented string similarity measure. Furthermore, the matching algorithm often produces suboptimal results for small subtree structures due to the characteristics of the similarity measure that is used to decide whether two subtrees do match or do not match. The result in this case is—again—a non-minimal edit script.

Having a non-minimal edit script is unfavorable, since the edit operations are used to classify the significance levels of the changes. Under these circumstances, we are therefore not able to assess the impact of changes on other source code entities precisely, *i.e.*, we cannot reliably decide whether a change was relevant or irrelevant to the structure of a program.

1.2 Envisioned Outcome of the Thesis

This thesis aims at improving the matching algorithm presented by Chawathe *et al.* to make its edit script calculation more flexible. We focus on enhancements to the tree matching algorithm for reaching a more accurate and comprehensible outcome. To realize them, we investigate the suitability of different similarity measures or, to be exact, of combinations of them. In particular, we want to establish more intuitive matching between leaves, including a similarity-ranking, and a heuristic to handle small trees more accurately.

Furthermore, we aim to develop a benchmark to investigate accuracy (precision and recall) and efficiency (runtime performance) of the improved algorithm in comparison with the actual approach. The benchmark will also serve as an extensible and important tool for the evaluation of future work. We use artificial test cases as well as data taken from the medium-sized project ARGOUML[5] to cover common program structures.

[4] http://www.eclipse.org
[5] http://argouml.tigris.org

1.3 Structure of Thesis

The remainder of the thesis is structured as follows: Chapter 2 presents work related to source code change detection. We discuss the advantages as well as the disadvantages in comparison with our approach.

In Chapter 3, we deliver backround information on tree-like data-structures and point out the characteristics of source code in contrast to other structured documents. Next, we discuss the approach taken by Chawathe *et al.*, before we propose step-by-step customizations for improving change detection in terms of source code.

Chapter 4 establishes a benchmark that puts our improvements to their paces. We investigate whether our customizations are able to outperform the work done by Chawathe *et al.* during change detection. For this, we use artificial source code examples and real data taken from the ARGOUML-repository.

Conclusions are drawn in Chapter 5, where we also suggest perspectives of future work. Finally, Appendix A lists additional benchmarking results.

Chapter 2

Related Work

This chapter presents and discusses the related work in the field of change detection and summarizes the shortcomings as well as the advantages in contrast to our approach.

2.1 Change Detection Based on Lexical Differencing

Techniques and tools for computing textual differences between documents are well-known and approved. However, existing tools such as *GNU diff* deal with flat, rather than with hierarchical information. They usually calculate a list of lines that must be changed, inserted, or deleted to make a first document match a second one. Listing 2.1 shows two subsequent revisions of a Java class file and Listing 2.2 the corresponding output of *GNU diff* under Apple's OSX. Lines from the first file are prefixed by $<$ and lines from the second are prefixed by $>$. In other words, to transform the original version into the new version, you have to delete all lines prefixed by $<$ and append those lines prefixed by $>$.

Apparently, the major shortcoming of *GNU diff* is that we are able to tell that *e.g.*, line 2 has changed, but we are not able to decide whether the change was relevant to a program's structure or to its functionality. We cannot, for example, distinguish between changes concerning license information or documentation and changes affecting classes or methods. Furthermore, large, but syntactically irrelevant changes, for instance indentation or reformatting, easily lead to a mismatch between two versions of a file. Regarding our example in Listing 2.2, *GNU diff* cannot detect, that the class has been renamed, that an attribute of the class was deleted and that a method invocation has been added between the old and the modified version.

Our approach, using abstract syntax trees for representing the hierarchical relationships between entities of a document, allows to detect a lot more changes more precisely and we are able to assign a particular change to a concrete source code entity (such as the declaration or body part of a method), rather than to a line number. This establishes additional possibilities, *e.g.*, fine-grained distinction between source code changes according to tree edit operations or classification of the significance level of changes, depending on their type and on the source code entity that they were applied to. Looking at the example above, we are now not only able to detect that a developer applied a class renaming, an attribute deletion, and that he inserted a method invocation statement into the `main(String[] args)`-method, as well as that the class renaming will have a strong impact on other source code entities (since a part of the public interface of the class has changed), and that the attribute deletion or insertion of the method invocation statement respectively, will not involve other changes.

```
//original version:
public class HelloWorld {
   private HelloWorld theInstance = new HelloWorld();
   public static void main(String[] args){
   }
}

//modified version:
public class NewHelloWorld {
   public static void main(String[] args){
      System.out.println("Hello World");
   }
}
```

Listing 2.1: Two subsequent versions of a .java-file.

```
unix-machine:~/$ diff -b HelloWorld.java NewHelloWorld.java
1,2c1
< public class HelloWorld {
< private HelloWorld theInstance = new HelloWorld();
---
> public class NewHelloWorld {
3a3
> System.out.println("Hello World");
```

Listing 2.2: The output of GNU diff for the two files shown in Listing 2.1.

2.2 Change Detection Based on Syntactic Differencing

We have implemented and modified the change detection algorithm for hierarchically structured data that has been presented by Chawathe *et al.* [CRGMW96]. Their algorithm addresses the problem of detecting and representing changes to hierarchically structured information. They cover structure in *ordered trees* and describe the deltas between two versions of hierarchical data using the notion of a *minimum edit script*, defined by *node insert*, *node delete*, *node update* and *subtree move* as basic editing operations. Although the approach applies well to the problem of AST-based source code change detection, it has been validated by calculating and representing changes in structured documents—LaTeX documents, in particular—and thus had to be further adapted to satisfy our special needs (see Section 3.2.3 for more details on the *acyclic label condition*, which often is a characteristic of structured text, but not of source code in particular).

Xing and Stroulia presented *an algorithm for tracking structural changes between designs of subsequent versions of object-oriented software* in [XS05]. They perform reverse engineering on two source code versions to recover the corresponding UML class models and use them as input for their tool UMLDIFF. The outcome is an edit script, similar to the one calculated in our approach, consisting of *additions*, *removals*, *moves*, *renamings* of different entities, *changes to their attributes* and *changes of the dependencies* amongst them. Unlike our tool CHANGEDISTILLER, UMLDIFF is also able to track moves between different classes. However, UMLDIFF focuses on *recovering higher-level design knowledge evolution*, *i.e.*, changes on the interface level, whereas our work additionally allows fine-grained differencing on the implementation-level, *i.e.*, changes on single statements inside of method-bodies.

2.3 Change Detection Based on Semantic Differencing

Apiwattanapong *et al.* describe a completely different approach in [AOH04]. They define a new representation, based on control-flow graphs, for modeling the behavior of object-oriented programs. Enhanced[1] control-flow graphs of two subsequent program versions are used as input for their algorithm CALCDIFF, which identifies modifications using graph isomorphism. Doing so, Apiwattanapong *et al.* are able to detect behavioral changes between two versions of a method.

We discuss this difference by taking a closer look at Figure 2.1, showing an example for two enhanced control-flow graphs, each representing a version of a method. The source code statement corresponding to node 7 throws the exception `E3`. Node 20 explicitly catches the exception `E2`, whereas node 22 explicitly catches exception `E1`. Assume now, that the supertype of the exception `E3` has changed from `E2` to `E1` between the original (represented by graph (a)) and the modified (represented by graph (b)) version of the method. In the original program, the exception `E3` was caught by node 20, since it is a subclass of `E2`. Due to the change to the exception hierarchy in the modified version, it is now a subclass of `E1` and therefore it is caught instead by node 22. Although the modification took place at a single line of the program—the class declaration part of `E3`—the control-flow between the statements, *i.e.*, the behavior of the program, has changed completely. Our algorithm detects a *Parent Class Update* and rates the impact of the change on other source code entities as *crucial*, but misses the change in behavior. We claim that both approaches, the one presented by Apiwattanapong *et al.* and our work, are complementary and that semantic differencing can be used to extend and refine our classification.

[1] *Enhanced*, in a way that traditional control-flow graphs are enriched by information covering object-oriented features of the programming language, *e.g.*, *dynamic binding* or *exceptions*.

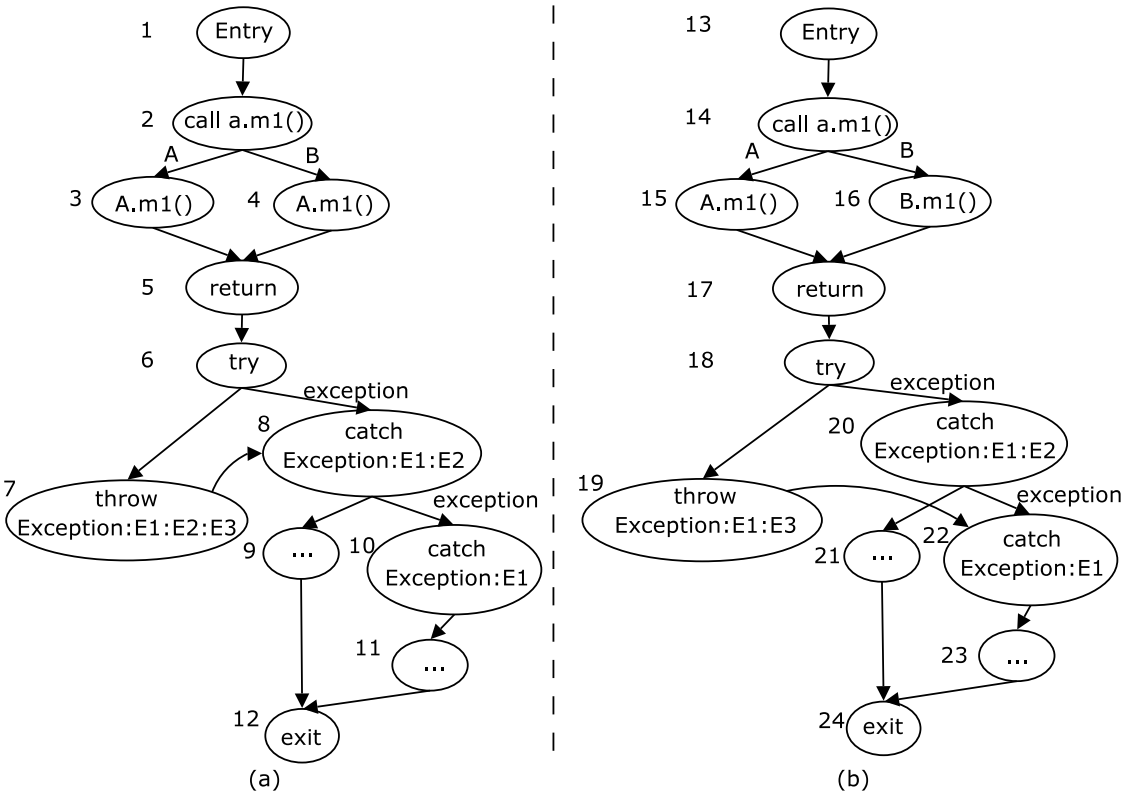

Figure 2.1: Enhanced control-flow graphs representing two subsequent versions of a method (source: [AOH04]).

2.4 Code Clone Detection

The field of code clone detection research is closely related to change detection; To identify the parts of a program that have changed, it is necessary to find those parts first that have experienced only small modifications or that have not changed at all. These parts can be considered as *near miss clones* and *clones* (between files representing subsequent versions of a source code entity) respectively, if spoken in terms of code clone detection. Nevertheless, there are noteworthy differences between the requirements for a good code clone detection algorithm and a good change detection algorithm that have to be taken into consideration while evaluating the use of the former for change detection:

- **Edit script generation:** In terms of changes, it is desirable to have a set of basic instructions as output of the algorithm, explaining how to transform the original version of a program into the modified one. Code clone detectors usually do not provide anything similar, although they sometimes calculate macros or patches that refactor the investigated source code in a way that duplicated parts can automatically be removed.

- **Taking program structure into consideration:** The reason for using abstract syntax tree comparisons in code clone detections is to achieve better accuracy. Locating the hierarchical position of duplicates is not as important, as it is for change detection to know where a particular change took place. Hence, token-based approaches as presented for example in [KKI02] are perfectly suited for code clone detection, but are less or not at all applicable to

change detection.

- **Restrictions concerning comparisons:** Code clones can occur between classes that have otherwise nothing in common. Thus, comparisons between each source code entity contained by a software system can become necessary. We, in contrast, only have to detect changes between subsequent versions of a file. If we presume that the versioning system was used in a disciplined manner, then we can exploit this knowledge by applying several heuristics, such as that changes between subsequent versions are usually quite small. This allows us to search more specifically for particular changes.

There are a number of algorithms applicable to detecting code duplicates, so our discussion will be limited to those approaches that are most related to our work:

Sager *et al.* [SBPK06] used several tree matching algorithms for detecting similar Java classes. First, they convert the abstract syntax tree as generated by ECLIPSE into the intermediary model FAMIX [DTS99]. In a second step, they transform the model into a generic tree format. The generic tree representations of all classes of a software system are then matched against each other in order to find duplicated code. Sager *et al.* evaluated three different tree similarity algorithms for this purpose, derived from a *bottom-up maximum common subtree isomorphism*, a *top-down maximum common subtree isomorphism* and an *edit distance of two given trees*, all three originally presented in [Val02]. These algorithms can be used to replace the tree similarity measure calculated in our tool CHANGEDISTILLER.

Baxter *et al.* describe another tool for code clone detection in [BYM+98]. Their approach also relies on abstract syntax trees, but categorizes subtrees by hashing. This reduces the number of comparisons needed significantly, since only subtrees with the same hash values have to be compared. Classification using hash values works well for exact duplicates, but fails for locating near-miss clones, *i.e.*, code duplicates that are very similar, but experienced for example parameter renaming. Baxter *et al.* are able to overcome this shortcoming by choosing an artificial bad hash function, *i.e.*, a function that ignores identifier names.

Chapter 3

Improving ChangeDistiller

This chapter covers the first part of the main contributions of this thesis. First, we give a brief introduction to CHANGEDISTILLER. Next, we set the stage for our improvements by providing some background information on tree data structures and similarity analysis of structured documents in general. Furthermore, we carefully point out the characteristics of documents containing source codes and their differences in contrast to other structured text such as LaTeX. Eventually, we discuss the work by Chawathe *et al.*, before we present and evaluate some options. Finally, we draw conclusions and suggest an improved algorithm for finding changes between two versions of a source code entitity. An empiric evaluation can be found in Chapter 4, where we apply a benchmark to put our efforts through their paces.

3.1 ChangeDistiller - A Tool for Classifying Change Types

Fluri and Gall presented their tool CHANGEDISTILLER in [FG06]. It implements a source code change extraction algorithm. This thesis focusses on finding improvements to this algorithm, so that it can handle changes more adequately. Before we dive into the main subject, we give an overview on the architecture of CHANGEDISTILLER and how it is embedded into the infrastructure.

The tool integrates into the reengineering workflow, due to the fact that it is built as a plug-in for the ECLIPSE-platform. See Figure 3.1 for an overview on the integration of CHANGEDISTILLER into ECLIPSE. Focal to all efforts lies the *Release History Database* (RHDB), originally presented in [FPG03]. The RHDB-approach accounts the fact that versioning systems *provide only insufficient support for a detailed analysis of software evolution aspects*. CHANGEDISTILLER is a logical next step towards understanding software evolution, *i.e.*, the way how software changes over time. It extends the insufficient capabilities of modern versioning systems for tracking changes by providing structural- instead of lexical change analysis between subsequent revisions of file containing source code.

The link to the versioning system is established by EVOLIZERBASE, a plug-in that relies on `org.eclipse.team` for checking out multiple revisions of a file from a versioning system such as CVS. Evolizer also provides an object-oriented interface to the RHDB by using the O/R-mapper Hibernate.[1] CHANGEDISTILLER exploits the compare plug-in of Eclipse to extract the declaration and body changes between the subsequent revisions of the Java source code files that were checked out before and uses the API of `org.eclipse.jdt` to retrieve their abstract syntax

[1] http://www.hibernate.org

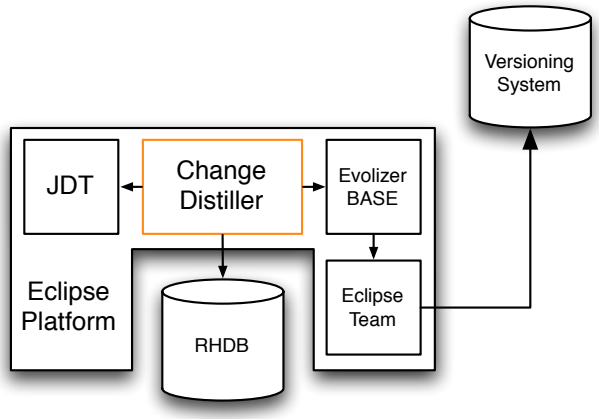

Figure 3.1: Change Distiller is a plug-in for the Eclipse Platform.

tree. Next, it transforms the AST into an intermediate tree-representation, where leaves are statements valued with their string representation. The intermediate ASTs are used as input for the source code change extraction algorithm. The output is a set of basic tree edit operations, transforming the tree of the original revision into the modified one. CHANGEDISTILLER stores the edit operations via EVOLIZERBASE into the RHDB and classifies them further. It also provides visualization and additional analysis integrated to an Eclipse-perspective. For details, please refer to [FG06].

The approach yields promising, but improvable results, under the precondition that changes between subsequent versions of a source code file are relatively small. This is usually guaranteed whenever the versioning system was used in a disciplined manner, *i.e.*, when commits happen on a regular basis. The generated edit script (and consequently the classification of the changes) will always be correct—even if these circumstances are not given—but accuracy will suffer in terms of a non-minimal number of classified source code changes. In other words, the algorithm detects unnecessary changes, but applying them will transform the original program version correctly into the modified one.

In the remainder of this chapter, we will point out the issues of the process described above and present our efforts on improving CHANGEDISTILLER.

3.2 Background Information

The following sections introduce the concept of tree-like data-structures and define some important vocabulary. We render the utility of abstract syntax trees as a representation for documents containing source code and talk about the nature of source code as a subset of structured documents. This knowledge will prove itself useful later on, when we present the approach by Chawathe *et al.* and how we tempt to customize it for source code change detection.

3.2.1 Tree-like Data-structures in General

In computer sience, a tree is a commonly used data-structure. Speaking in terms of graph theory, a tree is a directed acyclic graph consisting of nodes interconnected by edges representing a parent-child relationship. If a node n has a child m, then n is called *parent node of* m, which we denote $n = p(m)$ according to the notation used by Chawathe *et al.* in [CRGMW96]. Nodes along the path to the top of the tree are called *ancestors of* m. In return, m is called their *descendant*. The top-level node, *i.e.*, the only node in the tree that has no parent, is called *root node* or simply *root*. The bottom-level nodes, *i.e.*, the nodes that have no children, are called *leaf nodes*. Nodes in-between are *inner nodes*. Whenever the distinction between *root*, *inner node* and *leaf* doesn't add to our discussion, we will talk about *nodes* in general. A node n typically has a value denoted $v(n)$ and can have an optional label denoted $l(n)$, storing *e.g.*, a type or name value for the node. Figure 3.2 illustrates the vocabulary that we have discussed. The colored leaf on the bottom right shows how we illustrate labels and values from now on: The leaf is labeled as A and has the value *val*.

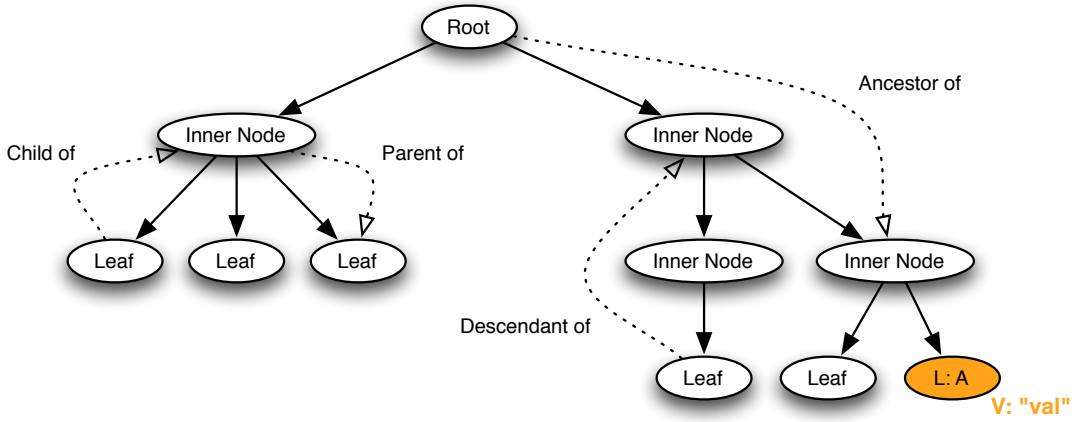

Figure 3.2: A generic tree structure and the relationships between some of the nodes. The right-most leaf shows how we annotate labels and values of nodes.

3.2.2 Abstract Syntax Trees (AST)

Tree-based data-structures became very popular in the field of compiler development: Whenever a file containing source code is sent to the compiler, several steps are performed before machine-instructions can be generated. The code has to be tokenized by a *Lexer* first: It separates the input stream into individual tokens and passes them to the *Parser*, which uses a context-free grammar of the programming language to build an intermediate code representation, a so-called *Parse Tree*. Each token found by the lexer is represented by a node in the parse tree. Not every token/node has a semantic value: Some tokens, for instance parentheses and semicolons, are purely syntactic sugar and can be therefore omitted. The resulting data-structure is called an *Abstract Syntax Tree (AST)*. Now that the source code is represented as a tree, it can be analyzed in a more sophisticated manner than while parsing the flat token stream: The tree can be traversed or searched in various ways (*e.g.*, pre- or post-order, and depth- or bread-first respectively) or its structural appearance

```
public class FooBar {
      public void foo() {
            System.out.println("foo");
      }
}
```

Listing 3.1: An example class in Java.

can be used to determine possible compiler optimizations.

Listing 3.1 shows the source code of a simple Java compilation unit, *i.e.*, a class `FooBar` containing a single method called `foo()`. The method contains a single method-invocation-statement that causes the Java Virtual Machine to print a short string to the standard output. Figure 3.3 shows the same class represented by an abstract syntax tree. As mentioned before, we will find a node-equivalent in the tree for each semantically relevant token in the source code: There is for example a node *MODIFIERS* present, having a leaf-descendant *KEYWORD: 'public'* that reflects that the class `FooBar` was declared with *public* access. There is also a subtree rooted by the node *BODY_DECLARATIONS* that contains the declaration- and body-part of the method `foo()` including the above-mentioned method-invocation-statement.

This tree representation of source code is not only useful for the compiler, it can also be used to perform other analysis since it covers the structure of the program. It is easy to traverse a tree while collecting infos on its nodes. For example, we can write a simple visitor that returns nodes representing a method declaration to quickly gather all methods of a program together. Last but not least, we can use the trees of two source code entities to compare them on a structural basis. Abstract syntax trees are therefore fundamental to our approach.

3.2.3 Source Code Characteristics

This section outlines some of the specifics of source code in contrast to other (structured) documents and discusses the implications while using tree-structures to represent and analyze syntactic program structure.

Vocabulary for Node Values

When dealing with natural language, *e.g.*, while comparing LaTeX documents, one faces a very broad and diverse vocabulary, making it more unlikely to find two identical leaf-values in one sub-tree. The pool of possible instructions in a programming language, such as Java, is usually more limited and a lot of statements follow the scheme `object.message(list of params)`. Furthermore, programmers often use reoccurring idioms and patterns and short identifiers, leading to more than one possible pair of matching nodes in many cases. If this issue is not taken into account, algorithms for source code change detection will produce non-optimal matching results compared to algorithms for structured documents in general.

Ordered vs. unordered Trees

The most common form of a tree is an ordered structure, *i.e.*, a tree where there is an order imposed for the children of any given node, established *e.g.*, by assigning a natural number to each child. Unordered trees, on the other hand, do not posses an ordering on their nodes. We illustrate

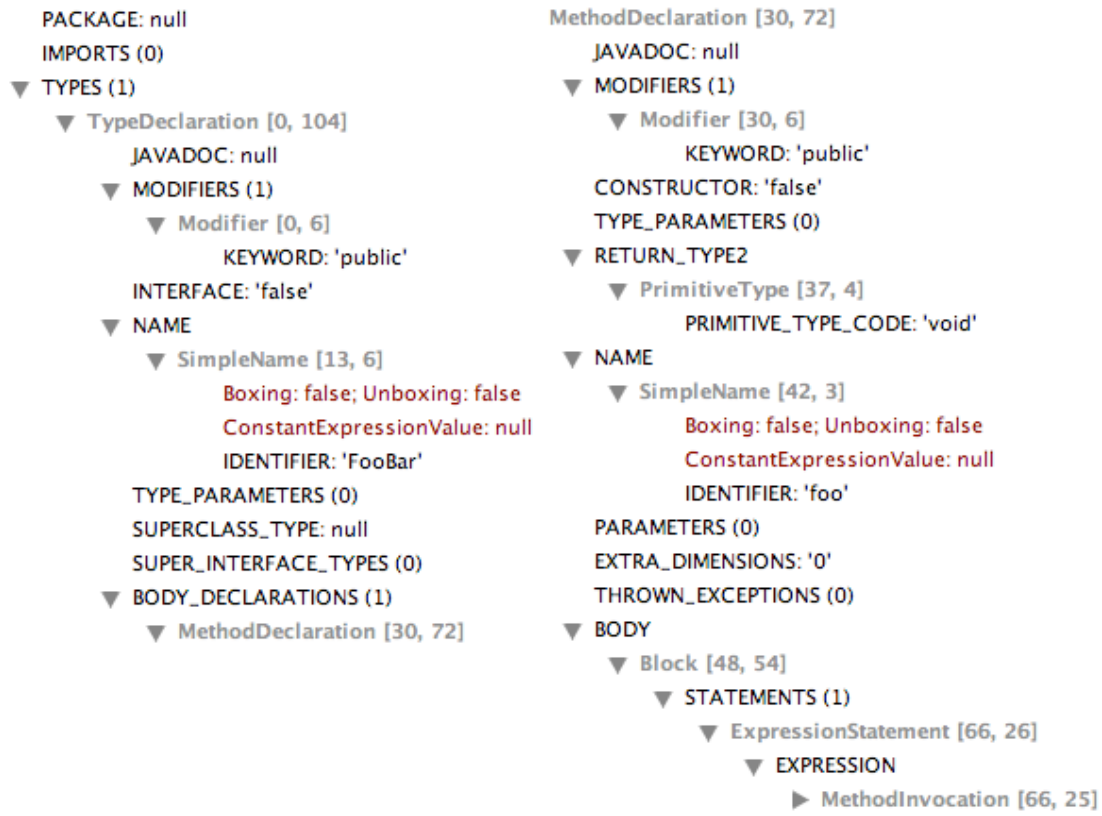

Figure 3.3: The AST generated by `org.eclipse.jdt.core.dom` and visualized by the Eclipse plug-in AST View for the Java class `FooBar` introduced in Listing 3.1.

this issue by discussing tree representations for source code: In most object-oriented languages, the order in which attributes and methods are declared inside the class body is completely irrelevant, *i.e.*, it doesn't matter if method `m1()` is declared before or after method `m2()` or even before or after field `f1`. Hence, an unordered tree is perfectly suited for representing the parent-child relationship between a class and the attributes and methods declared inside of the class. Completely different for individual statements inside of the method body, where the sequence of their execution plays a decisive role: It is crucial, whether the declaration of a variable `var1` took place before using it for example as operand in an assignment such as `var = m1();` or not. Figure 3.4 shows the relation between ordered and unordered tree-levels in Java.

The question, whether ordered or unordered trees are on hand, impacts the applicability of particular matching algorithms: Although any algorithm, designed solely for comparing ordered trees, is also valid for unordered ones, it will fail to match the trees if the ordering of nodes has changed. Even if the (sub-)tree is unordered and—consequently—ordering change should have no relevance. An algorithm for unordered trees allows to overcome this limitation, but in return it misses relevant ordering changes on the statement-level.

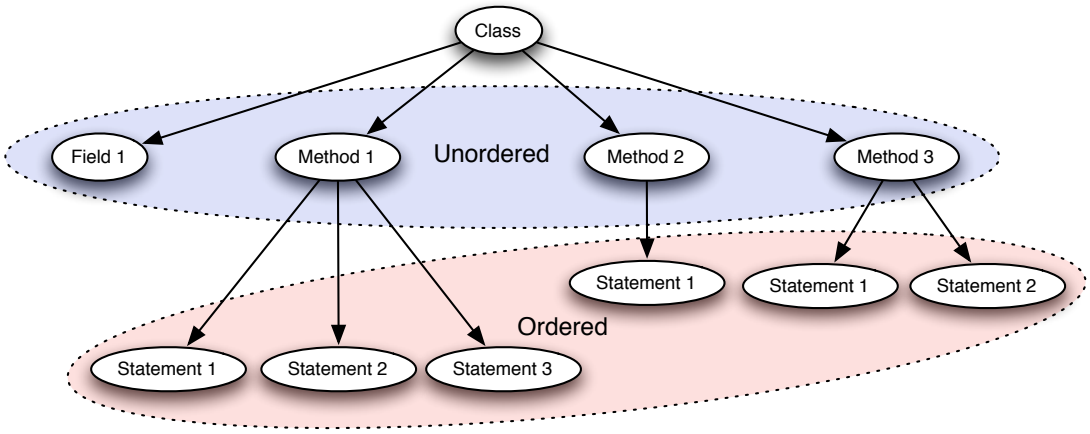

Figure 3.4: A tree representing a Java class containing a field and three methods. On the second level of the tree, ordering is irrelevant, whereas on the third level—inside of the method-body—the order among the statements must be preserved.

Acyclic vs. Cyclic Labels

Chawathe *et al.* stated in [CRGMW96] that nodes in tree-representations of hierarchical information often follow a so-called *acyclic labels* condition, meaning that there is a natural ordering $<_{\text{label}}$ between possible values of a label for a tree-node, so that a node with the label l_1 can appear as the descendant of a node with the label l_2 only if $l_1 <_{\text{label}} l_2$.

A structured document usually consists of *chapters*, which are divided into *sections*, which are composed of *paragraphs* and so on, whereas it is not possible for *e.g.*, paragraph-nodes to have children that are labeled as chapter or even as paragraph. The acyclic labels condition is satisfied. However, labels of source code nodes do not follow this ordering in general; Code usually consists of nested if-/else-blocks and nested loops.[2]

Figure 3.5 shows possible AST representations of a part of a structured document in comparison with a source code fragment. Values of the nodes have been omitted due to reasons of convenience. *L: CHAP* denotes a node labeled as chapter, *L: SEC* denotes a section and *L: PARA* denotes a paragraph. There is a natural ordering so that $L_{\text{paragraph}} < L_{\text{section}} < L_{\text{chapter}}$ is assured. The labels are acyclic. *L: IF* denotes an `if`-statement, *L: THEN* denotes the then-clause, whereas *L: ELSE* denotes the else-clause. *L: MI* denotes a method invocation. The condition $L_{\text{then}} < L_{\text{if}}$ and $L_{\text{else}} < L_{\text{if}}$ does not hold, since it is possible to have for example a nested if-statement in the else-clause. In this case, the labels are cyclic.

The presence of acyclic labels can be exploited for building simpler or faster tree matching algorithms. Since we cannot assert this condition to source code, we are not able to use the exploits for our concerns.

[2]Note: There are also relationships between certain (sub-)structures so that they cannot have cyclic labels, *e.g.*, methods cannot contain qualified class definitions

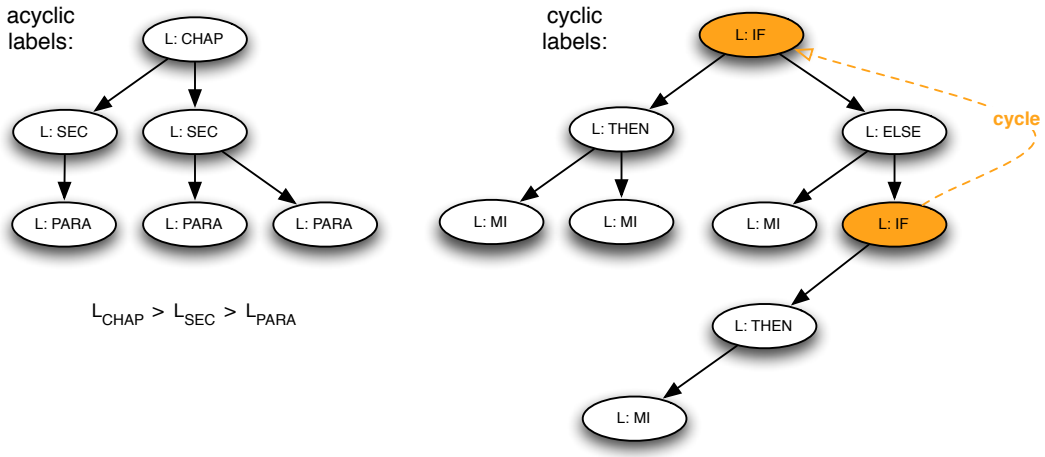

Figure 3.5: Simplified AST representations of a part of a structured document on the left opposite to a source code fragment on the right.

3.3 Outline on the Change Detection Algorithm by Chawathe et al.

The approach taken in CHANGEDISTILLER builds upon the algorithm for change detection which was presented in detail in [CRGMW96]. Nevertheless, we give a short outline on the whole algorithm and have an elaborate look on those parts where we think that we can contribute enhancements regarding source code change detection. We start our discussion by showing how Chawathe *et al.* split the problem of change detection into two subtasks:

- *Finding a "good" matching between the nodes of the two trees.*
- *Finding a minimum "conforming" edit script for the two trees given a computed matching.*

Finding a correct and accurate matching between the nodes is *crucial* to the outcome of the edit script algorithm. Assume that all nodes are matched perfectly, except those that were actually inserted or deleted. In this case, the edit script, calculated as suggested by Chawathe *et al.*, will be correct and minimal by all means. Hence, we found only little clearance in improving the edit script calculation, but focussed on customizing the matching algorithm for source code change detection instead. For this reason, we will explain edit script calculation very briefly in the next section, before we will cover the matching procedure in more detail in Section 3.3.2. Later on, in Section 3.4, we will discuss the circumstances under which matching unintentionally fails.

3.3.1 Calculating an Edit Script

The whole algorithm starts by matching the left and the right (*i.e.*, the original and the modified) tree in a bottom-up manner (see Section 3.3.2 below for a detailed discussion on the matching procedure) and then passes the calculated matching set of node pairs to the next step. The edit script generation sub-algorithm then runs through five phases, each designed for detecting a particular edit operation. In each phase, we will traverse the trees in a breadth-first order, which ensures that parent nodes will be visited before their children. The five phases are discussed below and illustrated in Figure 3.6. Note that some details were omitted due to reasons of simplicity and—again—that a thorough discussion including a more sophisticated running example can be found in [CRGMW96]:

1. **The update phase:** The algorithm first looks for pairs of nodes that made it into the matching set but differ in terms of their value. Whenever such a condition is met, an update-operation is added to the edit script.

2. **The insert phase:** Now that all nodes are aligned, the algorithm searches for nodes to insert, *i.e.*, nodes that are present in the right tree, but not in the left one. For that, unmatched nodes in the right tree are located. If their parent belongs to the matching set, then they are considered as insertion and a corresponding operation is added to the edit script.

3. **The move phase:** Whenever two nodes *n* and *m* belong to the matching set, the set is queried for their parents *p(n)* and *p(m)*. If we cannot find a matching node pair containing both, *p(n)* and *p(m)*, then we add a move-operation to the edit script.

4. **The align phase:** In this phase, only parent-preserving moves are applied. After all updates have been applied, the algorithm aligns the children of each inner node by performing move-operations. Two children are considered 'aligned' if they have the same relative order in the original as in the modified tree, *i.e.*, if node *n* is left of node *m* in the first tree then its counterpart has to be also on the left of *m*'s counterpart in the second tree. Note, that it does not matter if there are any nodes in-between *n* and *m*. Only their relative position to each other decides whether they are aligned or misaligned. Using an algorithm based on the notion of a *longest common subsequence* ensures that the number of moves is held minimal.

5. **The delete phase:** Last but not least, the remaining nodes, *i.e.*, nodes found in the left tree, but not in the right tree, will be deleted. Therefore, the algorithm adds the according delete-operations to the edit script.

3.3.2 The Matching Procedure in Detail

Finding a appropriate matching for hierarchical keyless data is a non-trivial task, especially when the *acyclic label condition* (see Section 3.2.3) does not hold. The first question that arises is, "Under what circumstances can two trees be considered as matching each other?" The answer is challenging, especially since we do not want to test the trees on exact *equality*, but rather on *similarity*. Chawathe *et al.* starts by defining three fundamental matching criterions in [CRGMW96], necessary for the algorithm to produce an optimal result:

> ***Matching Criterion 1:*** *For leaf nodes $x \in T_1$ and $y \in T_2$, (x, y) can be in a matching only if $l(x) = l(y)$ and $compare(v(x), v(y)) \leq f$ for some parameter f such that $0 \leq f \leq 1$.*

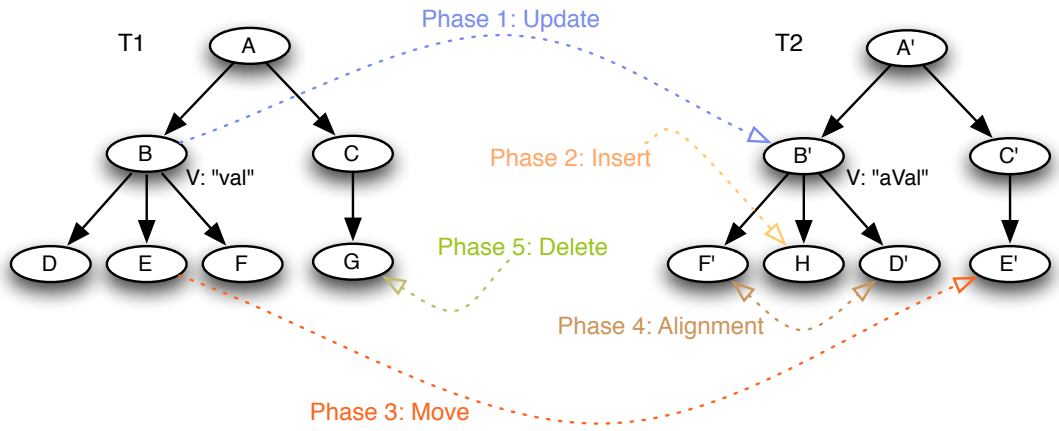

Figure 3.6: Phases 1-5 of the edit script generation algorithm by Chawathe et al. Nodes with the same letter are intended to match (example: A matches A') and values have been omitted unless they changed from T1 to T2.

The first matching criterion defines how leaves are matched. Only leaves—or nodes in general—of the same kind are allowed match: It is not possible for *e.g.*, a `if`-statement to be updated into a method invocation statement. This is assured by pre-testing their labels for equality. The function $compare(s_1, s_2)$ then compares the values of the leaves and returns a normalized number in $[0, 1]$ representing the edit distance between them. A distance value of 0.0 means that they are completely equal, whereas 1.0 denotes that they have nothing in common. The parameter f denotes the threshold above which the edit distance is considered to long and leaves are no longer allowed to match. In contrast to Chawathe *et al.*, we prefer to use similarities later on, rather than distances: We change the matching condition for values from $compare(v(x), v(y)) \leq f$ to $sim(v(x), v(y)) \geq f$ and consider a similarity of 1.0 as a 100%-match and a similarity of 0.0 as a complete mismatch. Although there is not a big difference between both notations, we claim that similarities are slightly more intuitive than distances. But for now, we will finish this section using Chawathe *et al.*'s notation.

> ***Matching Criterion 2:*** *Consider a matching M containing (x, y), where x is an internal node in T_1 and y is an internal node in T_2. Define:*
>
> $$common(x, y) = \{(w, z) \in M | x \text{ contains } w, \text{ and } y \text{ contains } z\}$$
>
> *Then in M we must have $l(x) = l(y)$ and*
>
> $$\frac{|common(x, y)|}{max(|x|, |y|)} \geq t$$
>
> *for some t satisfying $\frac{1}{2} \leq t \leq 1$.*

The next matching criterion addresses inner nodes. Again, labels are pre-tested for equality to assert that the nodes are of the same kind. However, the inner node matching does not use similarities of the node-values. Instead it uses a more natural measure, taking into account how many leaves the trees have in common. The number of common leaves—previously matched using matching criterion 1—is put into proportion to the number of total leaves of either the left or the right tree, depending on which one has more leaf-nodes. This tree similarity measure puts a strong focus on the leaf nodes and is therefore good for *e.g.*, LaTeX documents, where

the leaves—words or sentences of natural language—cover a lot of semantics, whereas inner nodes—chapters, sections, paragraphs, etc.—serve primarily for structuring purpose. However, intuitively, we claim that this measure is not well-suited for source code since inner nodes, representing for example `if`-statements, have a much stronger semantic value and should be therefore taken into consideration for making a decision whether two sub-trees should match or not. We discuss alternative tree similarity measures later on.

> ***Matching Criterion 3:*** *For any leaf $x \in T_1$, there is at most one leaf $y \in T_2$ such that $compare(v(x), v(y)) \leq 1$. Similarly, for any leaf $y \in T_2$, there is at most one leaf $x \in T_1$ such that $compare(v(x), v(y)) \leq 1$.*

The last matching criterion is one of the main reasons why the approach by Chawathe *et al.* often produces suboptimal results for source code comparisons. The assumption that there is at most one leaf in the right tree that matches a particular leaf in the left tree (and vice versa) is a necessary precondition for the algorithm to produce an optimal matching and a minimal conforming edit script, consequently. Even if the criterion fails, Chawathe *et al.* can often *post-process the sub-optimal solution to obtain an optimal solution* provided that the *acyclic labels condition* is taken for granted. Neither the matching criterion nor the *acyclic labels condition* (as argued before, see Section 3.2.3 for more information) can be guaranteed for source code. While the matching criterion is likely to hold for documents containing sentences of natural language, *i.e.*, sentences composed of broad and diverse vocabulary, it is not applicable to source code in general. Subsequent similar statements, however, are very common in programs. A popular example is shown in Listing 3.2. Although only the statement ordering was changed between the original and the modified code-block, the algorithm is very likely to detect statement updates instead, since more than one pair of leaves fulfills the condition $compare(v(x), v(y)) \leq 1$. We will discuss this issue further in Section 3.4.

Now that we have defined the three matching criterions, it is time to present the complete matching algorithm: The algorithm starts by initializing a datastructure that shall accommodate the pairs of matching nodes,—the matching set. Additionally, it marks all nodes of the both trees as *unmatched*. That followed, it compares each node of the left tree with each node in the right trees, starting at the left-most leaf and traversing the tree in a post-order manner. Comparison of nodes takes place by invoking an `equal`-function. Whenever a node pair is considered *equal*, it is added to the matching set. Marking both nodes as *matched*, ensures, that they will not be compared with other nodes again. According to Criterion 3, the first time a nodes matches has to be both, the best and the only match, so proceeding in this way is perfectly correct. The $equal(x, y)$-function is defined for leaves as follows. f is a parameter such that $0 \leq f \leq 1$:

$$equal(x, y) = \begin{cases} true, & \text{if } l(x) = l(y) \text{ and } compare(v(x), v(y)) \leq f \\ false, & \text{otherwise} \end{cases}$$

Comparison of inner nodes is done by a dedicated $equal$-function, where $t \geq \frac{1}{2}$ is a parameter:

$$equal(x, y) = \begin{cases} true, & \text{if } l(x) = l(y) \text{ and } \frac{|common(x,y)|}{max(|x|,|y|)} > t \\ false, & \text{otherwise} \end{cases}$$

The complete matching algorithm can be reviewed in Algorithm 3.1. Next, we will discuss the circumstances under which matching fails to produce an optimal solution.

```
//original
JButton button1 = new JButton("OK!");
button1.addActionListener(this);

JButton button2 = new JButton("NOT OK!");
button2.addActionListener(this);

JPanel panel = new JPanel();
panel.add(button1);
panel.add(button2);

//modified
JButton button2 = new JButton("NOT OK!");
button2.addActionListener(this);

JButton button1 = new JButton("OK!");
button1.addActionListener(this);

JPanel panel = new JPanel();
panel.add(button2);
panel.add(button1);
```

Listing 3.2: Two versions of a sequence of statements for intializing a graphical user interface in Java.

Data: Trees: T_1, T_2
Result: The matching set: M

1 $M \leftarrow \phi$;
2 Mark all nodes of T_1 and T_2 "unmatched";
3 **foreach** *unmatched node* $x \in T_1$*, if there is an unmatched node* $y \in T_2$ *such that* $equal(x, y)$ **do**
4 | Add (x, y) to M;
5 | Mark x and y "matched";
6 **end**

Algorithm 3.1: The algorithm *Match* by Chawathe *et al.* (source: [CRGMW96]).

3.4 When Does Matching Fail?

Before we go deeper into the discussion on the shortcomings of the algorithm by Chawathe *et al.*, outlined in the preceding sections, we have to render more precisely what *failing* means: The whole algorithm cannot fail in a sense that it will yield incorrect results, *i.e.*, leading to edit scripts that do not transform the left into the right tree correctly. The edit script will always be correct, but if the matching is inadequate, the solution may not be optimal in terms of a non-minimal result.

We have already mentioned some of the issues in the sections before: The quality of the $compare$-function and the associated threshold parameter f introduced in the first matching criterion are crucial for an optimal matching on the leaf-level. When Matching Criterion 3 does not hold, a mismatch on leaves—no matter if legitimate or not—can be propagated to inner nodes, leading to a mismatch on higher levels. This can happen whenever a certain number of children of a inner node violate Matching Criterion 3, although this is particularly pronounced for small subtrees. We will take a closer look on issues concerning leaf-matching based on node values next, before we will illustrate mismatch-propagation later on, using small subtrees as example.

3.4.1 Node Values

Matching leaf-nodes is based on two conditions: The leaves have to be of the same kind, which we can verify by testing their labels for equality. The second condition applies to the values of the leaves and is evaluated using the function introduced in Matching Criterion 1. In terms of the abstract syntax trees that we use, values correspond to statements (or to the condition in case of a `if`-statement) which are stored as strings. Figure 3.7 shows a snapshot of the matching algorithm, taken while trying to match a leaf x from T_1 with its counterpart y in T_2. From a humans' point of view, we state intuitively, that they are similar enough to be considered as an original and a modified version of the same statement, especially against the background that the statements where found in the same context, *i.e.*, in subsequent versions of the same method of a class.

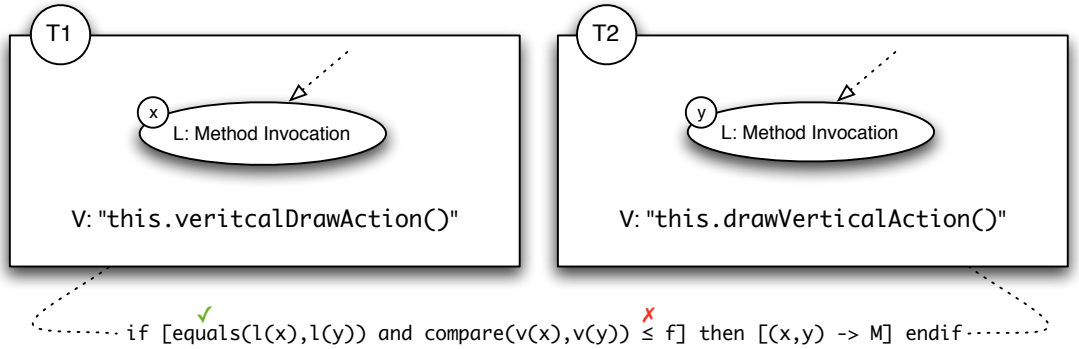

Figure 3.7: Snapshot taken while matching leaves. Node x is one of the leaves of the left tree T1 and y its counterpart in the right tree T2.

Deciding whether the two leaves, are similar or not is a trivial task for a human, since he knows the semantics of the words *draw*, *vertical* and *action*. From a machine's point of view, it

```
if (a > b) {
   a.b();
   a.c();
}
```

Listing 3.3: The original if-statement.

is not that easy. We will discuss some approaches later on (see Section 3.5.2), but for now, let us just accept, that the string similarity measure originally implemented in CHANGEDISTILLER—the Levenshtein Distance—is very susceptible to word or even character ordering changes. As we can see in our example, common renaming of identifiers during refactoring often involves changing the word order. Correcting typos (take a close look at leaf x: the characters 'i' and 't' were interchanged by accident) leads to a even worse miscalculation. If we want to allow the example-leaves to match, we have to lower the string similarity threshold f significantly, possibly resulting in a lot more false negatives.

3.4.2 Small Subtrees

A mismatch on a single leaf-pair, as it occured in the example discussed in the section above, does not have a noteworthy impact on the quality of the outcome of the algorithm; We find additional insert- and a delete-operations instead of an update-operation in the edit script. Unfortunately, as mentioned before, these mismatches can be propagated to higher levels of the tree, leading to a complete mismatch of a whole subtree and therefore to many unnecessary operations making the edit script not even approximately minimal. The situation even escalates in case of lowering f too much while Matching Criterion 3 does not hold and while the statement ordering has changed awkwardly at the same time. If these three unfavorable constraints meet, unintended moves may not only propagate mismatching to their whole subtree of origin, but also to the whole subtree of destination. This will cause the edit script to deteriorate further.

We discuss the propagation of mismatches using small trees as an example: Between Listings 3.3 and 3.4, a single statement was deleted and new one was inserted instead. Assume, that we experienced a swap of unrelated statements, rather than an update and that the surrounding code did not change at all. In addition, threshold t introduced in Matching Criterion 2, shall be **more** than 0.5. This results to the following matching criterion for the inner nodes $x \in T_1$ and $y \in T_2$:

$$l(x) \stackrel{!}{=} l(y) \text{ and } \frac{1}{2} \stackrel{!}{<} \frac{|common(x,y)|}{max(|x|,|y|)}$$

Figure 3.8 visualizes the same source code using an abstract syntax tree representation. *L: IF* denotes an `if`-statement. Its value corresponds to the `if`-condition. *L: THEN* denotes the `then`-block. Since it has no immanent value, it inherits that of its parent to emphasize their affiliation. Last but not least, *L: MI* denotes method invocation statements, that are listed as values.

We traverse the trees post-order from left to right, thus starting with the left-most leaves. The leaf nodes representing the method invocation `a.b();` in T_1 and T_2 do match according to Matching Criterion 1. So, they are added to the matching set and marked as *matched*. Since the first leaf of the left tree is now already matched, we can directly proceed to the second one. We do

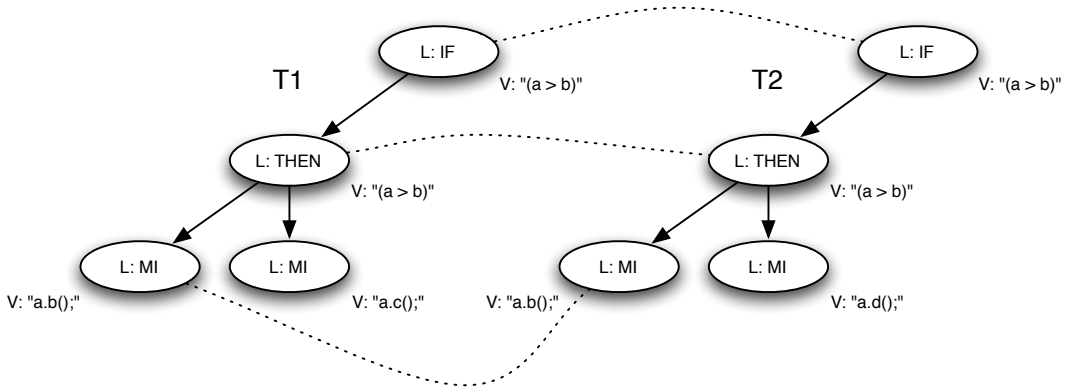

Figure 3.8: An example of two similar trees T1 and T2 for which the algorithm fails to calculate a minimal edit script.

```
if (a > b) {
   a.b();
   a.d();
}
```

Listing 3.4: The modified if-statement: The method invocation a.c(); was replaced by a.d();

not have to compare the right leaf of T_1 with the left leaf of T_2, because we just now have matched the latter. Therefore we skip the first leaf of the right tree and continue with the second one. Although the labels are the same, the values `a.c();` and `a.d();` cannot be matched. Figure 3.9 illustrates the situation after all leaves were visited. We proceed to the next level in the tree and reach the inner node representing the `then`-statement in T_1. We do not have to compare it with the leaves of T_2, since their labels differ. Hence, we can proceed directly to the next node with the same label, *i.e.*, the `then`-node in T_2. Remember, inner nodes are matched in accordance to Matching Criterion 2, so we count the number of common leaf-descendants of the both nodes under investigation and divide them by the maximum number of leaves of either trees, leading to the following tree similarity and therefore to a mismatch:

$$\frac{|common(x,y)|}{max(|x|,|y|)} = \frac{1}{2}$$

The tree and its matchings after these steps are illustrated in Figure 3.10. The `then`-blocks can not be matched, because their similarity was 0.5 (remember, threshold t was fixed above this value). No other nodes with the same label can be found in the tree. We proceed to the root of the subtree, in this case the `if`-statement, which we are not able to match either, due to the same reasons: We count the common leaves (disregarding the inner node descendants, so the common nodes stay the same as for the `then`-node before), divide them by the maximum number of leaves and thus calculate a similarity of 0.5 again. The final (mis-)matching is shown in Figure 3.11.

Although the trees T_1 and T_2 are very similar and—intuitively spoken—should match, the al-

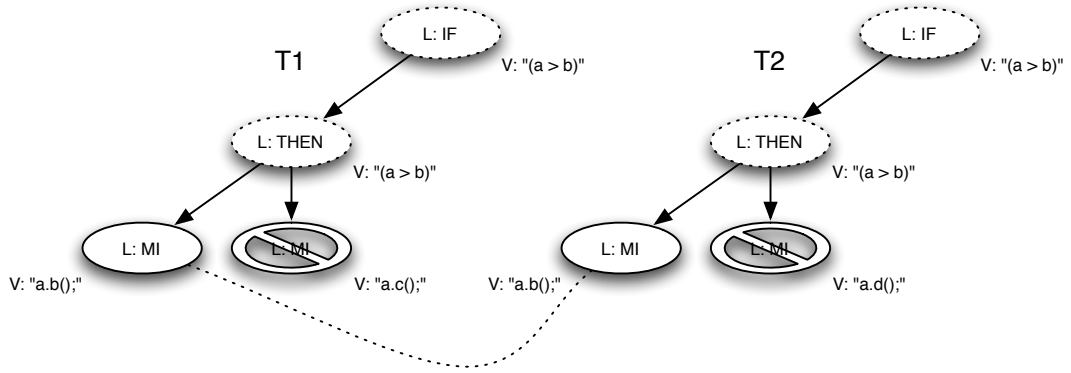

Figure 3.9: First step of bottom-up-matching-example: We decide wether the leaf-nodes match or mismatch by using a dedicated leaf-comparator.

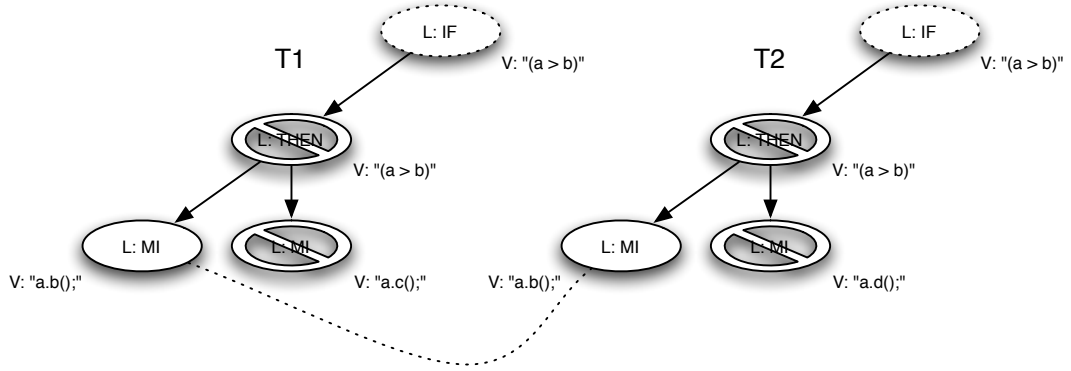

Figure 3.10: Second step of bottom-up-matching example: If a certain amount of leaf-nodes does not match, we decide to mismatch the parent node.

gorithm fails to do so. Regarding small subtrees, the question therefore arises, whether a matching based on labels and values only, rather than similarities based on descendants of the root of subtrees, produces better results. We will take this question into account while discussing possible improvements in Section 3.5. To anticipate a little bit, we prefer to keep on using similarities for subtrees in spite of the mentioned concerns, since we think that it is a good way to determine changes, especially when *e.g.*, more than one `if-else`-blocks with similar conditions are involved in a single source code entity.

3.4.3 When Matching Criterion 3 Fails...

We already mentioned that Matching Criterion 3 has to hold in order to get a minimal conforming edit script. We also mentioned the dependence of Matching Criterion 2 on Matching Criterion 1's ability to to match values accurately. But what happens if the values of *several* node pairs (x, y_i), where $x \in T_1$ and y_i is one of the *many* possible counterparts of x in T_2, are too similar, *i.e.*, if their edit distance lies above threshold f?

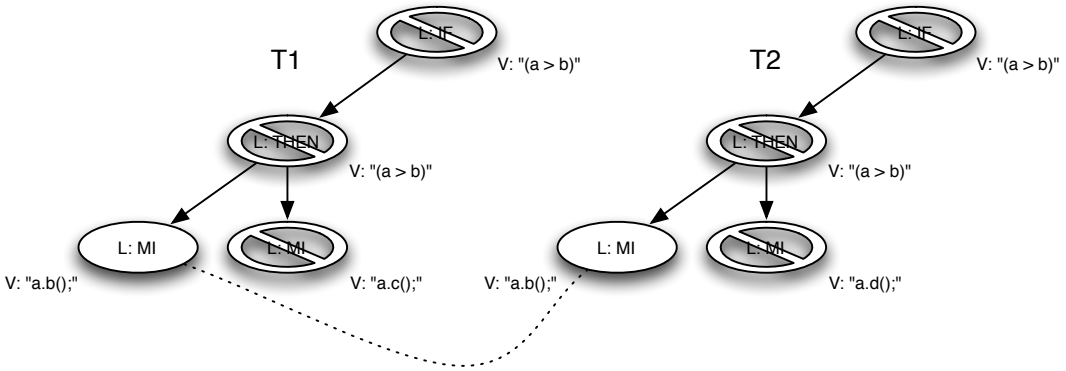

Figure 3.11: Third and last step of bottom-up-matching-example: The whole subtree is considered as mismatched.

Figure 3.12 shows the tremendous consequences that a single unfortunate, but common, statement insert can have. The matching algorithm starts again, traversing the tree post-order. The

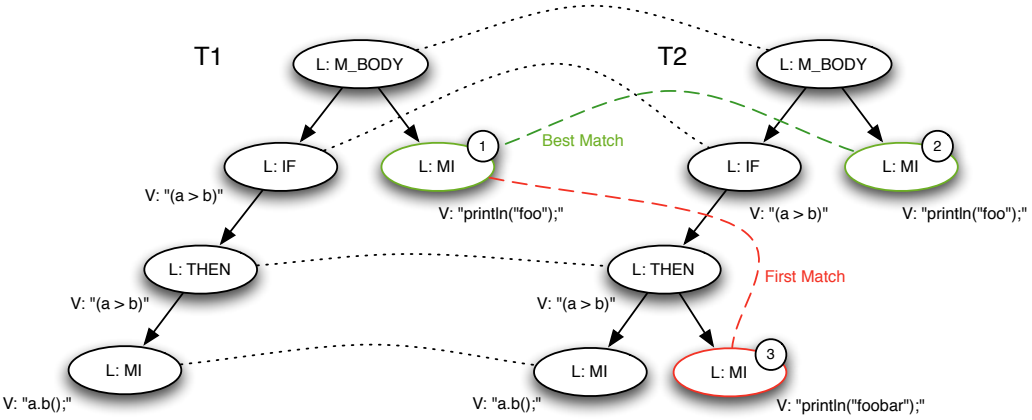

Figure 3.12: Suboptimal results are very likely to occur whenever Matching Criterion 3 does not hold.

left-most leaves match, but there is no leaf counterpart in T_1 that matches Leaf 3 in T_2 at this time. Hence, the `then`-nodes cannot be matched due to Matching Criterion 2: They have one common leaf, whereas the maximum number of leaves in either trees is two, resulting in a similarity of 0.5. The mismatch is propagated to the `if`-statements. And finally, the remaining leaves are tried to match.

This is where the influence of Matching Criterion 3 becomes apparent: There is more than one possible counterpart for node 1 in the right trees, namely Nodes 2 and 3. Unfortunately, the tree is searched in a post-order manner again and thus, Nodes 1 and 3 are put into the matching set, whereas the better match, *i.e.*, the pair of identical Nodes 1 and 2, is not even considered to match. In T_1, the root is the only node that remains.

Due to the simplicity of our example, we are able to intercept mismatching propagation on this

third level: According to Matching Criterion 2, the roots match because they have two common leaves divided by a maximum of three leaves in T_2, leading to a similarity of $\frac{2}{3}$, which lies above threshold t. Table 3.1 summarizes the changes that we expected on the one hand, compared to the changes found on the other hand. Even for our trivial example, the algorithm found six times the changes we expected.

Change type	expected	found
Insert	1	4
Delete	0	3
Move	0	1
Total	**1**	**6**

Table 3.1: Changes expected and found for Figure 3.12

During our research on source code taken from real life projects such as ARGOUML,[3] we encountered mismatch propagations over two or three levels very often, *e.g.*, in nested `if-else`- and `try-catch`-blocks. Every now and then, we discovered even more sophisticated examples, where matching failed disastrously, leading to dozens of unnecessary change operations. The levels of propagation seem to correlate with the nesting depth of *e.g.*, `if-then`-statments or loops and the number of involved statements.

Albeit their low frequency, these excessive propagations can have huge implications on classifying the significance of the impact of changes on other source code entities. This motivates the heuristics, that we will present during the next few sections, more than sufficiently.

[3] http://argouml.tigris.org

3.5 Customizing the Algorithm for Source Code Changes

In the sections above, we motivated that the hierarchical change detection algorithm by Chawathe *et al.* needs to be adapted further to take the specialties of source code into account. In addition, we have discussed the circumstances under which the algorithm fails to calculate a minimal solution for the edit script that transforms an original tree T_1 into a modified tree T_2. Next, we summarize the improvements that we intend to achieve, before we outline our efforts in detail.

3.5.1 Desired Improvements

Consequentially to the shortcomings that were summarized before, we can compile the enhancements that we aim to achieve as follows:

1. We present a **string similarity measure that is customized for matching tree node values** representing source code statements. We contrast several possibilities in Section 3.5.2
2. Propagation of mismatches leads to an enormous amount of deletions and insertions under some circumstances. This is especially pronounced for small trees. Thus, we aim at finding a solution for **matching small trees more adequately**. We present our thoughts on this issue in Section 3.5.4.
3. Chawathe *et al.*'s Matching Criterion 3 does not apply to source code in general. Often we can find more than matching candidate for an original node. Under this circumstances, it is very likely, that the *first* match will not always be the *best* match. Our approach for **finding the best match** is outlined in Section 3.5.4.

We proceed by developing measures for reaching the desired improvements step by step. Before we gather them altogether into a single algorithm, we are going to provide some helpful basics along the way of outlining the string and tree similarity measures, that we have taken into consideration.

3.5.2 Evaluated String Similarity Measures

String similarities—or similarities in general—are an important point of research in many scientific fields. Text processing, *e.g.*, information retrieval systems or data mining, DNA-research in biology, spell checkers and many more applications are depending on string similarities, since simple equality checks fail way too much often on data where misspellings and modifications are daily business.

Next, we discuss some approaches for string matching. Most of the presented metrics are provided by SIMPACK, a generic Java library for similarity measures in ontologies [BKKB05]. SIMPACK is not only useful in ontologies. It can contribute to many of the domains that were mentioned above; Amongst others to source code change detection.

The Levenshtein String Similarity Measure

The Levenshtein Distance is a well-known measure to determine the similarity of two given strings s_a and s_b and is often used *e.g.*, by spell checkers. It is relatively cheap to calculate, since the problem can be reduced to a runtime-complexity of $O(n * m)$, where n is the number of characters in s_a and m is the number of characters in s_b, using an algorithm based on dynamic programming. Levenshtein calculates the minimum number of operations needed to transform one string into the other. These operations can be one the following:

1. Insert a character
2. Delete a character
3. Substitute a character

A larger distance means, that the strings are less similar, *i.e.*, that more operations are necessary to transform one string into another, whereas a distance of 0 operations denotes that the strings are the same. The algorithm is based on the problem of the *longest common subsequence*, since it must find the characters first, that s_a and s_b have in common. Afterwards, we can decide whether the subsequence was interrupted and if so, applying which operations will fill the gaps. Let us discuss an example:

s_1	**L**	**e**	**v**	e	**n**	**s**		**h**	**t**	e	**i**	**n**
s_2	**L**	**e**	**v**		**n**	**s**	c	**h**	**t**	a	**i**	**n**
Op				*D*			*I*			*S*		

The two strings $s_1 = Levenshtein$ and $s_2 = Levnschtain$ are intuitively as well as phonetically very similar. The letters in bold mark the longest common subsequence of s_1 and s_2, namely '**Levnshtin**'. The bottom line shows the edit operations that have to be applied to transform s_1 into s_2: *D* denotes a *Delete*-Operation, *I* a *Insert*-Operation and *S* an *Substitution*. In our example, there are three edit operations necessary to change the first string into the second one: A delete of the letter 'e', an insert of the letter 'c' and a substitution of 'e' against 'a'. The cost for all operations are the same for Levenshtein[4], thus leading to the following distance- or cost-function:

$$D(s_a, s_b) = c(s_a, c_b) = d + i + s$$

for d, i and $s \in N$, representing the number of delete-, insert- and substitution-operations needed. In our example, the edit distance between s_1 and s_2 is 3. For our concerns, distances are less useful than similarities, since we cannot state that a distance of 3 is generally better than a distance of 4. - It depends on the lengths of the compared strings. If both strings have a length of three characters, then a distance of 3 would be terribly bad. If both strings have a length of 100 characters, the same distance would suggest, that they are almost identical. To overcome this issue, we have to normalize and convert the distance, using a distance-to-similarity conversion:

$$sim_{\mathrm{Lev}}(s_a, s_b) = 1.0 - \frac{D(s_a, s_b)}{D_{\mathrm{worstcase}}(s_a, s_b)}$$

[4]Other related measures use variable cost adjustment. *Needleman-Wunch*, for example, uses variable costs for gaps, *i.e.*, insert/delete, depending on *e.g.*, typographic frequencies or amino acid substitutability, — according to the domain requirements.

The denominator $D_{\text{worstcase}}$ is equal to the maximum costs experienced under the assumption that the longest common subsequence of s_1 and s_2 has a length of 0, *i.e.*, that they have no characters in common. If s_1 and s_2 are of the same length, then $D_{\text{worstcase}}$ is equal to their length, since each single character provokes a substitution. If their lengths differ, then we additionally have to take insertions, and deletes respectively, into account. In our example, the strings s_1 and s_2 are both 11 in length and the edit distance was 3. This results in a similarity of:

$$sim_{\text{Lev}}(s_1, s_2) = 1 - \frac{3}{11} = 0.\overline{72}$$

The Levenshtein Distance has some limitations. The measure is very susceptible to changes of word or even of character order. The longest common subsequence lacks in robustness against such kind of changes as we will show next:

s_3	L	e	v	e	n	**s**	**h**	**t**	**e**	**i**	**n**					
s_4						**s**	**h**	**t**	**e**	**i**	**n**	L	e	v	e	n
Op	*D*	*D*	*D*	*D*	*D*							*I*	*I*	*I*	*I*	*I*

The strings $s_3 = Levenshtein$ and $s_4 = shteinLeven$ are, again, intuitively very similar. If they are even found at about the same place in two version of a source code document, then it is very likely that someone has performed refactorings *e.g.*, by unifying identifier-nomenclature or changing the order of method parameters. The Levenshtein Distance does not recognize this similarity as our example illustrates: The longest common subsequence shrunk drastically to '**shtein**'. The remaining characters cause five insertions and five deletions, *i.e.*, a total of ten change operations or a distance of 10 respectively. In terms of similarity:

$$sim_{\text{Lev}}(s_3, s_4) = 1 - \frac{5+5}{11} = 0.\overline{09}$$

Levenshtein's performance is rather poor in this case. Since we noticed during prototyping that a lot of unintentional mismatches on the leaf-level were actually based on the deficiencies of the string similarity measure, we are eager to find an algorithm showing more robustness. We will present an alternative to the Levenshtein Distance in the next section.

String Similarity Measures using n-grams

A family of string similarity measures is based on the *Dice Coefficient* [Dic45]—a modification of the *Jaccard Coefficient* [Jac12]. Adamson and Boreham used the Dice Coefficient to rate the similarity of strings by setting their n-grams into relation [AB74]. n-grams are constructed by putting a sliding-window of length n over a string and extracting at each position the n underlying characters. For instance, the 3-grams of the string "vertical" are: $3\text{-}grams(\text{vertical}) = \{$'ver', 'ert', 'rti', 'tic', 'ica', 'cal'$\}$. The n-gram similarity measure defined by Adamson and Boreham is the ratio of twice the number of shared n-grams and the total numbers of n-grams in two strings:

$$sim_{\text{ng}}(s_a, s_b) = \frac{2 \times |n\text{-}grams(s_a) \cap n\text{-}grams(s_b)|}{|n\text{-}grams(s_a) \cup n\text{-}grams(s_b)|}$$

Dice Coefficient with bi- and tri-grams are popular word similarity measures. In combination with source code, bi-grams are used by Xing and Stroulia for their UMLDiff approach [XS05], tri-grams by Weidl and Gall for their CORET approach [WG98]. In the following, we will restrict our investigations to Dice with $2\text{-}grams$.

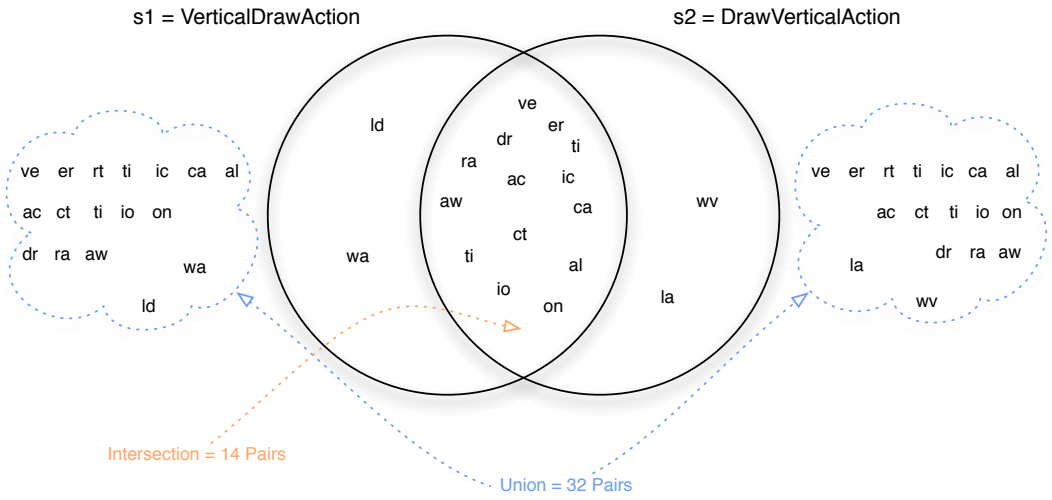

Figure 3.13: The two strings s_1 and s_2 have 14 pairs of characters in common out of a total of 32 pairs.

Figure 3.13 shows an example for two similar strings, namely $s_1 = VerticalDrawAction$ and $s_2 = DrawVerticalAction$. Levenshtein almost fails to recognize their similarity ($sim_{\text{Lev}}(s_1, s_2) = 0.\overline{55}$), whereas Dice perform very well:

$$sim_{2\text{g}}(s_1, s_2) = \frac{2 * 14}{32} = 0.875$$

Using a hash-table to store the n-grams of both strings, the runtime complexity of the n-gram similarity measure is in $O(n + m)$—one order of magnitude faster than Levenshtein.

The Dice Coefficient seems to be more robust to changes to the order of words, since it does not rely on the longest common subsequence. Instead, it focusses on common characters and treats ordering subordinate. For our concerns regarding source code in general and source code identifiers in special, the metric seems to allow a more intuitive similarity scoring. The measure performs worse than Levenshtein only under rare circumstances (rare, at least in conjunction with source code): It seems to be more susceptible to (real, *i.e.*, not caused by moves) substitutions including misspellings due to phonetical reasons that are common in natural language. The strings *Levenshtein* and *Levnshtain* for example, score with a similarity $0.\overline{72}$ when Levenshtein is used, but only with 0.5 when the Dice algorithm is used. We assume furthermore that the measure is limited to strings of a certain maximum length (*i.e.*, fuzzy scoring can occur when comparing huge classes or whole programs) due to the following reason: The given number of different characters is finite. As a string gets longer, it will become more likely that most permutations between characters are exhausted. The amount of character pairs in the intersection will therefore increase, leading to a kind of blurred similarity. However, we were not yet able to prove this expectation experimentally, but instead, we were able to confirm the applicability of the algorithm to source code on the statement- and even method-level.

Other Similarity Measures

Besides the Levenshtein Distance and the Dice Coefficient, we also experimented with most of the other measures that are implemented in SIMPACK. Although some of them performed very well in special cases, their overall performance in terms of source code statement comparison was not as promising as the approach using 2-$grams$. Nevertheless, we describe two of them representatively and discuss the circumstances under which they will score best in a very briefly manner. A more elaborate view on different string similarity measures for name-matching purpose can be found in [CRF03]:

- **Jaro:** Jaro presented his string distance metric first in 1989 in [Jar89]. The metric tries to cover typical spelling deviations between two strings s_a and s_b and is defined as follows:

$$sim_{\text{Jaro}}(s_a, s_b) = \frac{1}{3}\left(\frac{|s'_a|}{|s_a|} + \frac{|s'_b|}{|s_b|} + \frac{|s'_a| - T_{s'_a,s'_b}}{2|s'_a|}\right)$$

  $|s'_a|$ is the number of characters in s_a that are common with characters in s_b. $|s'_b|$, in return, is the number of characters in s_b that are common with those in s_a. Next, $|s_a|$ and $|s_b|$ are the lengths of s_a or s_b, respectively. $T_{s'_a,s'_b}$ is the number of necessary character-transpositions.

  On the one hand, the approach by Jaro shows very robust behavior against changes in word or character order. On the other hand, it behaves often too fuzzy and matches unintentionally statements such as the assignments `b = Math.round(Math.random());` and `b = Math.abs(number);` with high similarity scores. Furthermore, it did calculate a similarity of 0.0 in rare cases, even if the strings where almost the same. We were not able to determine whether this is an implementation issue of SIMPACK or desired behavior.

- **Monge-Elkan:** Monge and Elkan suggest a recursive matching scheme in [ME96]. We found that it shows a good overall performance (as also stated empirically in [CRF03]). Nevertheless, it was not as robust to ordering changes during our experiments as the approach using Dice.

Conclusions on String Similarity Measures

We have evaluated several ways to compare the similarity between a string s_a and another string s_b during the last few sections. Our goal was to find a measure that allows intuitive scoring of node-values, taking the peculiarities of source code on the statement-level into account. The measure must therefore meet the following requirement: Common refactorings, for instance renaming including insertions or removals of single words and changes in the order of the words, must be detected. Furthermore, correction of misspellings, including changes on single characters, should not be overrated. Hence, the desired metric must be robust to random small changes as well as to larger changes caused particularly by reordering. On the other hand, it should not behave too fuzzy, *i.e.*, it must not produce too many false negatives. Levenshtein cannot fully satisfy this conditions, since it is too susceptible to changes to the longest common subsequence of two strings. The Dice Coefficient using 2-*grams* yields a very good overall performance, although other approaches are superior under rare circumstances. We therefore prefer this algorithm for comparisons of leaf-values. For values of inner nodes, *e.g.*, for `if`-conditions and other short strings, where it is more likely that the longest common subsequence remains almost equal in subsequent versions to the whole strings, Levenshtein scores more precisely at least in theory. However, as we will explain later on in Chapter 4, we were not able to verify this speculation during our case study, possibly due to the character of our test data.

3.5.3 Evaluated Tree Similarity Measures

Chawathe *et al.* presented a simple but quite powerful tree similarity measure for inner node matching when they introduced Matching Criterion 2 (see Section 3.3.2). In this section, we will evaluate its theoretical suitability in terms of source code compared to an approach using also the Dice Coefficient, which was for example used by Baxter *et al.* for code clone detection.

Tree Similarity after Chawathe et al.

Next, we review the similarity measure for inner nodes that Chawathe *et al.* use in their work:

$$sim_{\text{Chawathe}}(x, y) = \frac{|common(x, y)|}{max(|x|, |y|)}$$

Although its solid performance during our experiments speaks for it, we see some theoretical limitations concerning source code. The measure takes only leaf descendants into account, when deciding whether two nodes should match. Inner node descendants are ignored completely. This seems to be an adequate approach for similarity analysis of structured documents such as those that are written in LaTeX, where the inner nodes are only used for structuring means and do not hold any—or only few—semantics. For source code, inner nodes are far more important, since some of them additionally cover fundamental constructs, for instance iterations and alternatives or exceptions handling. We are not yet sure about the implications on the quality of our similarity analysis and whether we should try to cover this issue with our syntactical/structure-based approach or if we should leave this to purely semantic work such as the already mentioned algorithm based on control-flow in [AOH04].

A second concern applies to the influence of leaves that are *not* common between the left and the right tree. We noticed that the amount of common leaves is somewhat underrated when it comes to large changes. We think that common statements should be weighted more significantly, since we compare subsequent versions of methods where it is very likely that inner nodes with

a certain critical number of common leaf-descendants should match: We found examples, where dozens of statements were extracted from an `if`-statement into a new method during refactoring. Although the core of the `if`-block remained the same, the original subtree could not be matched anymore with the modified one using the formula by Chawathe *et al.*

Dice Coefficient for Inner Nodes

Baxter *et al.* use another metric for computing the similarity of abstract syntax trees in [BYM+98]. Although their work applies to code clone detection, it yields some interesting perspectives to our approach. They use the *Dice Coefficient* to calculate the similarity between two subtrees:

$$sim_{\text{Dice}}(T_a, T_b) = \frac{2 \times |nodes(T_a) \cap nodes(T_b)|}{|nodes(T_a) \cup nodes(T_b)|}$$

where $nodes(T_x)$ denotes all nodes of T_x including the root. Note that Baxter *et al.* involve all nodes—not only leaves—in the similarity calculation. This accommodates to one of the points that we have criticized in the approach of Chawathe *et al.* The other point that we have questioned, was whether commonalities amongst descendants should have more influence on the similarity value of two inner nodes. The Dice Coefficient takes this issue into account by weighting common nodes with a factor of 2. Let us do an exemplary calculation on the example in Figure 3.14. T_1 and T_2 are isomorphic, except the following nodes: Node 2 has been deleted;

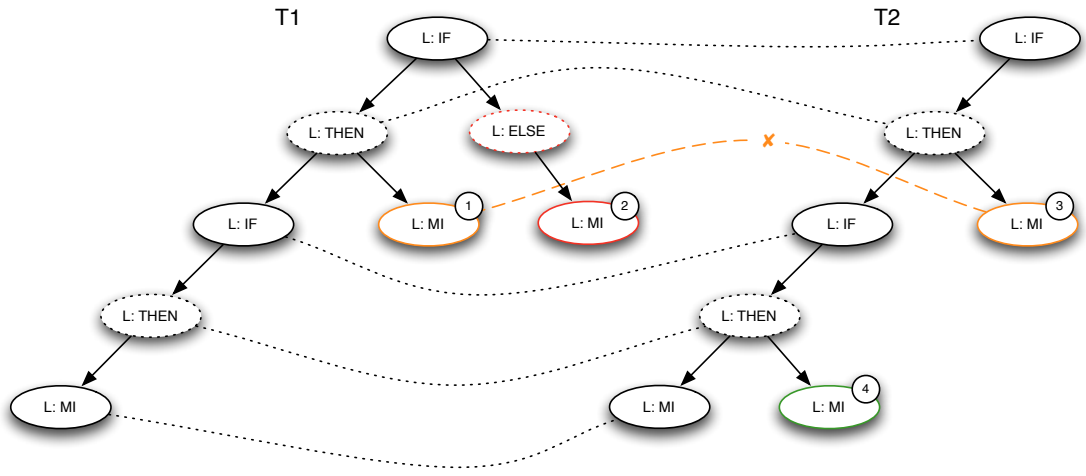

Figure 3.14: Nodes 1 and 3 do no longer match. Node 2 was deleted whilst node 4 is an insertion. Otherwise, the trees T_1 and T_2 are isomorphic.

Nodes 1 and 3 do not match and Node 4 has been inserted between T_1 and T_2. Note that we choose to ignore the `then`- and the `else`-nodes, since they belong to each `if`-statement implicitly and add nothing of interest to the tree structure. So, we count three shared nodes, *i.e.*, the two `if`'s and a single method invocation statement. There are two nodes in T_1 that do not have a counterpart in T_2: Nodes 1 and 2. In T_2, however, there are the nodes 3 and 4 which have no

partner in T_1. This leads to: $S = 3$, $L = 2$ and $R = 2$ and hence to the following calculation:

$$sim_{\text{Dice}}(T_1, T_2) = \frac{2 * 3}{2 * 3 + 2 + 2} = 0.6$$

A similarity of 0.6 denotes that the trees T_1 and T_2 match. In contrast, Chawathe *et al.*'s Matching Criterion 2 states a similarity of $0.\overline{33}$, which lies clearly below threshold t. As we can see, when we use the Dice Coefficient for calculation, the differences between the both trees have somewhat taken a back-seat in comparison to the commonalities.

Conclusions on Tree Similarity Measures

Even though the approach by Baxter *et al.*, using the Dice Coefficient, was able to convince us regarding artificial examples such as shown above, we see some issues. To illustrate them, we draw the conceptual difference between Chawathe *et al.*'s and Baxter *et al.*'s work as follows: Our matching algorithm proceeds through the tree in a bottom-up manner, *i.e.*, using post-order traversal. Consequently, we match leaves before inner nodes. Next, Chawathe *et al.* use leaf-node-descendants only, to decide whether two *inner nodes* are similar or not. That is completely unproblematic, since all leaves were visited before we did proceed to the inner node. In contrast, Baxter *et al.*'s metric involves the root of the subtree to decide whether two *code blocks* are clones or not, which is a kind of more holistic view on subtrees, rather than a focus on individual inner nodes.

The conclusion is, that if we simply exchange both tree similarity measures with each other, we change the semantics of our comparison slightly. In other words, the first approach—that by Chawathe *et al.* —focusses on determining whether two inner nodes are the same merely because of their children, whereas the second approach—that by Baxter *et al.* —already involves the equality of the labels and the similarity of the values of the inner nodes under investigation to determine whether the whole subtrees match. The first approach seems to apply better to source code changes, since we are interested whether two particular inner nodes are similar (for example to perform an update-operation on them if their values undergone a slightly change between two versions), whereas the second approach is customized for detecting whole blocks of duplicated code, where a single node is somewhat subordinate.

Suprisingly, as we show in Chapter 4, Chawathe *et al.*'s tree similarity metric produced better results on our test data than the Dice Coefficient. Nevertheless, we like the way how Baxter *et al.* weight common nodes stronger than mismatches and that they also involve inner-node-structure, rather than just leaves. Future work should therefore aim at integrating these two aspects into CHANGEDISTILLER. Unfortunately, good alternatives for tree similarity measures that are applicable to ASTs are rare. Most of other work in tree similarity research seems to be locked to a particular domain, *e.g.*, chemistry or biology. The tree edit distance approach by Sager *et al.* in [SBPK06] might contribute to our efforts, but we were not yet able to evaluate this assumption.

3.5.4 A Better Matching Algorithm

We have discussed the limitations of Chawathe *et al.*'s matching algorithm in Section 3.4. Next, we present an algorithm which is still based on the work by Chawathe *et al.*, but has been improved in terms of its applicability to source code. Our enhancements allow us to overcome most of the shortcomings that we have mentioned earlier. This proposition is proven in Chapter 4 by providing an extensive benchmark based on a medium-sized real world software project.

We introduce our approach in a bottom-up manner, starting with improvements regarding matching of leaf nodes. Next we discuss solutions for inhibiting propagation of mismatches to higher levels of the tree. Last but not least, we suggest a heuristic for damping down the negative impact of Matching Criterion 3 which was introduced in Section 3.3.2 and generally fails to hold for source code. Interestingly, we experience synergetic effects between our efforts, which we discuss later on. A second improvement—at least theoretically, since we were not yet able to prove better results using our test data—is the way how Baxter *et al.* weights shared or common nodes by using the Dice Coefficient: By weighting shared nodes by a factor of two, common nodes have a greater influence on the similarity of two subtrees than different ones.

Improving Leaf Matching

Leaves of the same kind are matched on a per-value basis by comparing strings. A good string similarity measure is crucial to the outcome of the algorithm. We have outlined several options and have concluded that the Dice Coefficient using 2-$grams$ for the calculation of string similarities yields more promising results than the originally implemented Levenshtein Distance measure. Since we no longer use a distance metric to define the *compare*-function, but rather a similarity, we have to invert the relationship between the *compare*-function and the threshold parameter f. By applying the new metric to Chawathe *et al.*'s Matching Criterion 1, we can conclude the first step of improvement. The modifications are highlighted in blue:

> **Modified Matching Criterion 1:** For leaf nodes $x \in T_1$ and $y \in T_2$, (x, y) can be in a matching only if $l(x) = l(y)$ and $compare(v(x), v(y)) \geq f$ for some parameter f such that $0 \leq f \leq 1$. The $compare$-function is defined as follows:
>
> $$compare(v(x), v(y)) = sim_{\text{XS}}(v(x), v(y))$$

Inhibiting Propagation of Mismatches for Small Trees

Solving the problem of propagation of mismatches is much harder to achieve than customizing leaf-matching. We are not yet able to get rid of it completely, but we can improve the algorithm's behavior in terms of small subtrees. We propose two different approaches to the problem: The first and probably most obvious solution are *dynamic thresholds*. The second approach relies on a tree similarity measure that is equivalent good for small- as well as for larger subtrees. The approaches are—at least theoretically—complementary.

- **Dynamic Thresholds:** The idea behind lowering the threshold t for trees that do not reach a certain number of children, is to weaken the disproportionately high impact that small changes can have to small subtrees. We have experienced a lot of small subtree structures while analyzing common software, such as small `if`-blocks where single statements were added or removed. Such an example is drawn between Listing 3.5 and Listing 3.6: The developer decided somewhere between committing the original code block and the modified one that he wants to implement a logging factory and therefore he introduced log-statements *e.g.*, along the program path for debugging purpose. The inner nodes representing the `if`-statement have one single leaf-descendant in common, whereas the maximum

```
if (cancelled()) {
   close();
}
```

Listing 3.5: A small `if`-block.

```
if (cancelled()) {
   close();
   logger.debug("user has cancelled the action);
}
```

Listing 3.6: A logging statement has been added to the example introduced in Listing 3.5

number of leaves in either trees is two. The node similarity is therefore 0.5 according to Matching Criterion 2. Thus, a mismatch occurs if we fix threshold t above 0.5. In case that there are additionally *e.g.*, ten unchanged statements in both `if`-blocks while the single insert remains, the similarity would rise up to more than 0.9. We experience some kind of imbalance.

As a quick and dirty solution, we can decide to lower the threshold for *all* inner nodes, no matter how many leaf-descendants they count. This injects undesired behavior into the algorithm: Imagine a god-class with methods that contain `if`-blocks made up of 500 statements. Can we still consider the `if`-blocks as similar when 250 statements change completely between two versions? Especially, when there are other `if`-blocks involved that show also a significant similarity relationship to the modified one. Hence, we summarize our thoughts in one sentence, before we turn towards a better approach: We cannot achieve a intuitive similarity scoring if we use the same thresholds for all inner nodes, disregarding their number of leaf-descendants.

Alternatively, the threshold t shall be chosen *dynamically*. Dynamically, meaning in regard to the size of the subtrees under investigation. Hence, we customize Matching Criterion 2. Again, the modifications are highlighted in blue:

> **Modified Matching Criterion 2, Variant a:** Consider a matching M containing (x, y), where x is an internal node in T_1 and y is an internal node in T_2. Define:
>
> $$common(x, y) = \{(w, z) \in M | x \text{ contains } w, \text{ and } y \text{ contains } z\}$$
>
> Then in M we must have $l(x) = l(y)$ and
>
> $$\frac{|common(x, y)|}{max(|x|, |y|)} \geq t$$
>
> for some t satisfying
>
> $$\frac{1}{2} \leq t \leq 1$$
>
> if each, x and y, have more than $n \in N$ leaf-descendants or

$$0 \leq t \leq \frac{1}{2}$$

otherwise.

We experience good matching results for $t = 0.6$ if $n > 4$ or $t = 0.4$ for $n \leq 4$. See Chapter 4 for details. It is even imaginable to use a more nuances for setting the threshold. This should be a subject to future research.

- **Another Tree Similarity Measure:** Another approach aims at integrating another tree similarity measure into Matching Criterion 2, - in order to be exact, a measure that is more robust to single mismatches. More robust in a way, that more changes between the descendants are accepted, before a mismatch is propagated to a higher level.

  A good solution involves all node descendants—not only leaves—on the one hand and assesses a bigger importance to commonalities on the other hand. We have discussed the reasons for both claims before, but until now we have failed to mention an acceptable side-effect of the first one: If we use a measure that takes inner nodes into account, we do not only cover more of the structure of the tree, we furthermore distribute mismatches over a greater number of nodes, supporting our efforts to dilute mismatch propagation. A possible candidate for replacing Chawathe *et al.*'s tree similarity measure has been introduced above: Baxter *et al.* weight common nodes (inner nodes as well as leaves) twice the times that changes are taken into account. Their Dice-based metric is therefore predestinated to replace the one of Chawathe *et al.*'s Matching criterion 2. The modifications that were necessary are highlighted in blue as usual:

  > **Modified Matching Criterion 2, Variant b:** Consider a matching M containing (x, y), where x is an internal node in T_1 and y is an internal node in T_2. Define:
  >
  > $$common(x, y) = \{(w, z) \in M | x \text{ contains } w, \text{ and } y \text{ contains } z\}$$
  >
  > and
  >
  > $$d_x = \{\forall w | x \text{ contains } w, \text{ but } y \text{ does not contain } w\}$$
  >
  > and
  >
  > $$d_y = \{\forall w | x \text{ does not contain } w, \text{ but } y \text{ contains } w\}$$
  >
  > Then in M we must have $l(x) = l(y)$ and
  >
  > $$\frac{2 * common(x, y)}{2 * common(x, y) + d_x + d_y} \geq t$$
  >
  > for some t satisfying $\frac{1}{2} \leq t \leq 1$.

Take a closer look at the differences between the Modified Matching Criterion 2, Variant b and the original one: d_x and d_y denote the number of nodes that are found in either tree, but not in both. Moreover, the formula is no longer limited to leaves only, but to all descendants—including inner nodes—of x and y.

We have now applied all promised enhancements that we were able to implement without modifying the matching algorithm by Chawathe *et al.* fundamentally. At the present point in time, the improved algorithm is more flexible than its predecessor when it comes to matching node-values and it is able to prevent most propagations of unintended mismatches from lower to higher levels in small subtrees. What is still left over, is the instance, that a non-minimal edit script will be calculated, every time that Matching Criterion 3 does not hold, *i.e.*, whenever matching

more than one counterpart of a node from the left tree with a node from the right tree is possible (or vice versa).

To overcome this severe insufficiency—at least severe in terms of source code—we have to review the whole matching process. This will happen in the next section.

Rating the Similarities to Determine the Best Match

During the last few sections, we have presented some tweaks to the original matching algorithm as presented by Chawathe *et al.*. The enhancements were rather easy to apply and involved a new string similarity measure for comparisons of node values as well as a finer gradation on thresholds for inner node similarities. In this section, our endeavors will go a step further. We fundamentally change the way how tree matching—in particular leaf matching—works. By doing so, we expect to cut down the numbers of those mismatches significantly that are related to the fact, that Matching Criterion 3 does not apply to source code in many cases.

Let us quickly review the problem, before we are going to talk about solutions: Chawathe *et al.*'s Matching Criterion 3 *states (informally) that the compare function is a good discriminator of leaves* [CRGMW96]. In Section 3.4.3, we have disproved this statement for source code. Figure 3.15 shows another example for a configuration of statements, where the algorithm will fail. If not

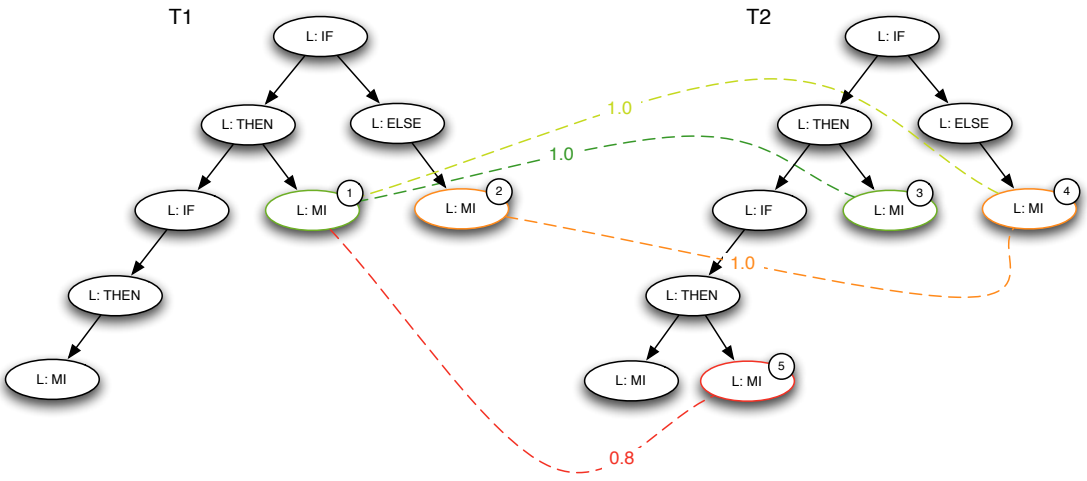

Figure 3.15: Another matching example: The trees are isomorphic except that node 5 has been inserted between T_1 and T_2. The labels of the dashed lines represent the similarity between the values of the interconnected leaves.

stated otherwise, we refer to this example during this section to illustrate our improved approach. Node 1 and 3 should match, but instead, the algorithm assigns Node 5 to Node 1 as the partner in the matching set. Remember, this is due to the way the algorithm proceeds from bottom to the top and from left to right: While comparing node 1 with each other leaf in T_2, we will visit node 5 before we can reach the best matching node, which is in this case Node 3. Since the similarity calculated by $compare(v(node_1), v(node_5))$ lies above the threshold (we still assume that $f > 0.5$), the node pair is added to the matching set, both nodes are marked as *matched* and therefore they will not even be considered in further comparisons.

Chawathe *et al.* propose the following post-processing step:

> Proceeding top-down, we consider each tree node x in turn. Let y be the partner of x according to the current matching. For each child c of x that is matched to a node c' such that $parent(c') \neq y$, we check if we can match c to a child c'' of y , such that $compare(c, c'') \leq f$, where f is the parameter used in Matching Criterion 1. If so, we change the current matching to make c match c''. [CRGMW96]

Whenever the *acyclic label condition* (introduced in Section 3.2.3) holds, the suggested post-processing step will be able to correct most of the sub-optimal matchings. Nevertheless, regarding our example, post-processing improves matching indeed: The post-processing step proceeds top-down in T_1 and eventually reaches the parent of node 1, let us call it node x. Node x's counterpart y in T_2 is the parent of Node 3. The algorithm detects that the counterpart of Node 1 in the matching set has not y as parent. Therefore, it checks each of the children of y, whether they are also matchable with Node 1. This is the true for Node 3 and thus, matching can be corrected.

Figure 3.16, however, shows the limitations of post-processing: Node 1 has moved between T_1 and T_2 to a new position: It has moved one level up (which is per definition not possible whenever the *acyclic label condition* is present) and is now represented by node 2. Post-processing is not possible under these circumstances; The parent of node 1 has not even a partner in T_2.

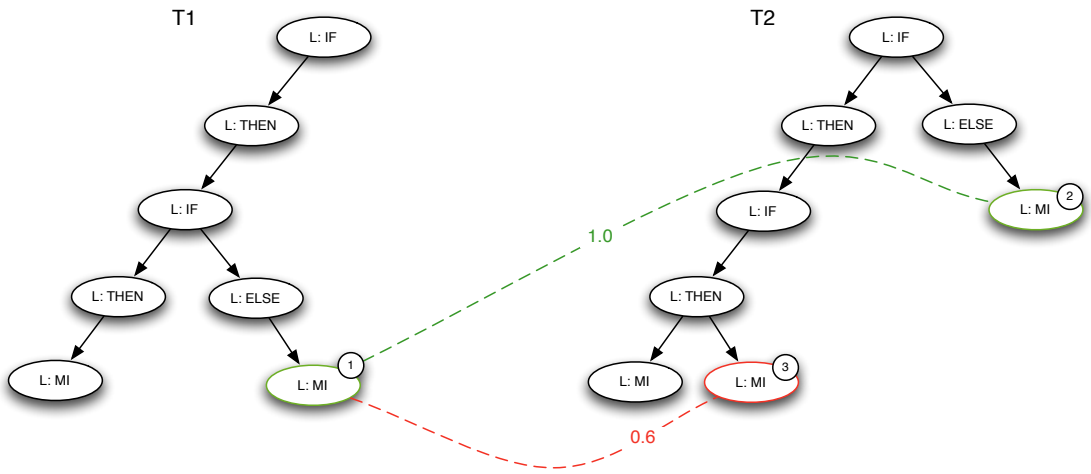

Figure 3.16: A trivial example of two trees, where the post-processing step will not be able to improve matching.

Although the mismatch in the last example is negligible—either way, a single statement parent change and a single statement insert will be classified—other more sophisticated examples involving a lot of unnecessary operations are thinkable. Thus, we are going to present an algorithm that covers both special cases; The one resolved by Chawathe *et al.*'s post-processing phase and the one that we have discussed now. We have seen a lot of examples during the last few sections, where failing of Matching Criterion 3 to hold, leads to a non-minimal edit-script in the bottom-line. Now, we are able to break down the insufficiencies to a single condition:

> Let x be a leaf in T_1 and y be his partner in T_2. Furthermore, let z be another leaf in T_2, so that
>
> $$compare(v(x), v(y)) \geq f$$

and

$$compare(v(x), v(z)) \geq f$$

but

$$compare(v(x), v(y)) > compare(v(x), v(z))$$

Whenever z will be visited before y during post-order traversal, a sub-optimal matching will be calculated.

Above, we state that the *first* match might not always be the *best* match, which usually results to sub-optimalities. According to this, we can now derive a solution for our problem:

Again, let x be a leaf in T_1. Furthermore, let p_i be its i-th possible partner in T_2, such that $i \in N$ and

$$compare(v(x), v(p_i)) \geq f$$

We mark (x, p_i) as *best match* until we find another possible partner p_{i+n}, such that $n \in N$ and

$$compare(v(x), v(p_{i+n})) > compare(v(x), v(p_i))$$

In this case, we mark (x, p_{i+n}) as *best match*. We will proceed by doing so, until we have tried to match all possible partners in T_2 to x.

The solution involves finding the leaf-partner $y \in T_2$ that matches $x \in T_1$ best. There are configurations of statements thinkable, so that there is more than one possible partner for x in T_2, *e.g.*, when one and the same statement can be found over and over again in a block of code (for example `System.out.println()'s` for debugging). In this case, we apply the heuristic, that unchanged statements stay *in situ* between subsequent versions of a source code entity: The first 'best' match, *i.e.*, the matching pair with the highest similarity score that has been visited during post-order traversal first, will make it into the final matching set; No matter if there are other matching pairs that have the same similarity.

We have now developed an approach for finding the best partner $y \in T_2$ for leaf $x \in T_1$. We did not yet discuss the fact, that this relationship is not always a two-way optimum, *i.e.*, whether x is also the best partner for y. Consider the example in Figure 3.17: The leaf in T_2 that matches best with Node 1 is Node 4. It is, in fact, the only possible partner for node 1, although Node 2 and 4 should make it into the final matching set instead.

We can overcome this issue: We simply calculate the similarity of each leaf pair $(x_i, y_j) \in T_1 \times T_2$ and consider those ones to be added to the final matching set first, that show highest similarity.

Finally, we present our improved matching algorithm in Algorithm 3.2. We call it *BestMatch*. Before we conclude this chapter, we point out noteworthy things on the algorithm. M_{final} is a set-like data-structure. It does not matter whether an order is preserved between the node pairs in the final matching set. M_{temp} must be a list-like data-structure that accounts for the order of its elements, because we perform sorting on them. Lines 4 to 17 perform leaf matching. This task is divided into three parts:

1. Lines 4 to 11 calculate a similarity for each leaf pair (x, y) in $T_1 \times T_2$. Very important: This is performed in a bottom-up and left-to-right manner (post-order traversal) and only pairs of leaves whose similarity lies above threshold f, can make it into the matching set. Otherwise, we miss real insert and deletes, since they will far more often be classified as updates.

Data: Trees: T_1, T_2
Result: The set containing the final matchings: M_{final}

$M_{\text{final}} \leftarrow \phi$;
$M_{\text{temp}} \leftarrow \phi$;
Mark all nodes of T_1 and T_2 "unmatched";
foreach *leaf* $x \in T_1$ **do**
    foreach *leaf* $y \in T_2$ **do**
        $sim(x, y) \leftarrow compare(v(x), v(y))$;
        if $sim(x, y) \geq f$ **then**
            Add $(x, y, sim(x, y))$ to M_{temp};
        end
    end
end
Sort M_{temp} into descending order, according to the leaf-pair-similarity;
foreach *leaf-pair* $(x, y, sim(x, y)) \in M_{\text{temp}}$ **do**
    Add $(x, y, sim(x, y))$ to M_{final};
    Remove all leaf-pairs from M_{temp} that contain either leaf x or y;
    Mark x and y "matched";
end
foreach *unmatched node* $x \in T_1$, *if there is an unmatched node* $y \in T_2$, *such that* $equal(x, y)$ **do**
    Add (x, y) to M_{final};
    Mark x and y "matched";
end

Algorithm 3.2: Our improved algorithm called *BestMatch*.

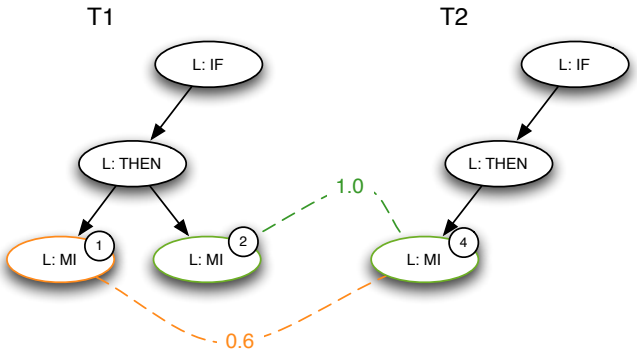

Figure 3.17: To determine the best match overall, we cannot simply focus on finding the most similar partner $y \in T_2$ for $x \in T_2$. We have to ensure that this relation is symmetric.

2. Line 12 sorts the leaf pairs into an ascending order, according to their similarities, *i.e.*, pairs that where more similar are moved to the beginning of the list of matches. We use a modified mergesort algorithm that offers $n \log n$ performance. Furthermore, the sorting algorithm is stable, *i.e.*, it preserves the order amongst equal elements. This is rather important for our concerns: Review Figure 3.15, where Nodes 1 and 3, as well as Nodes 1 and 4 match, both pairs with a similarity score of 1.0. A stable sorting algorithm accounts for the heuristic mentioned above, that unchanged statements stay in situ between subsequent versions of a source code entity.

3. Lines 13 to 17 decide whether a leaf pair will make it into the final matching set. In the end, a leaf l can be matched at most once (or not at all), so we have to remove all pairs containing l from M_{temp}, after a pair containing l was added to M_{final}.

The last step, Lines 18 to 21, was adopted from Chawathe *et al.* as it stands. The step performs the matching of the inner nodes based on Matching Criterion 2 or the modified criterions, respectively. This happens again bottom-up. What is left over, is to define the $equal(x, y)$ for the inner nodes more explicitly.[5] We provide two definitions, which will be evaluated against each other during the next chapter:

$$equal(x,y) = \begin{cases} true, & \text{if } l(x) = l(y) \text{ and } \frac{|common(x,y)|}{max(|x|,|y|)} > t \\ false, & \text{otherwise} \end{cases}$$

The parameter t is lowered or raised dynamically, depending on the number of descendants of x and y, as discussed above. The other possible definition follows:

$$equal(x,y) = \begin{cases} true, & \text{if } l(x) = l(y) \text{ and } \frac{2*common(x,y)}{2*common(x,y)+d_x+d_y} > t \\ false, & \text{otherwise} \end{cases}$$

[5] In fact, the function does not evaluate equality, but rather similarity of two inner nodes. Nevertheless, we decide to keep the identifier that Chawathe *et al.* have chosen, for the reason that we do not want to claim that this part of the algorithm is different in contrast to their approach.

In the latter case, we chose not to set threshold t dynamically, since the tree similarity measure already does a good job for small tree structures. Instead, we fix t at 0.5.

We have now applied all the improvements to our algorithm as promised. It will now yield optimal results in most cases in terms of matching and consequently, in terms of a minimum conforming edit script. There are still some minor shortcomings, that we will discuss in the next section.

3.6 Conclusions and Shortcomings

In the beginning of this chapter, we have introduced AST-based source code change detection. Later, we have seen that there are often significant differences between other structured documents and source code in terms of change detection. During our research, it became evident that a good string similarity measure is crucial to reach good and intuitively comprehensible matchings. Levenshtein's String Edit Distance was not able to fully convince us. The Dice Coefficient using 2-$grams$ seems to apply better to our requirements in most cases. Nevertheless, no matter how good a string similarity can be calculated, there are some limitations which we cannot overcome. Under circumstances where syntactical similarity is not present, our algorithms fail, no matter if a human's understanding of context and semantics is able to resolve the same difficulties. For example, we doubt, that it ever will be possible to match the two identifiers `draw` and `paint` reliably when using a purely syntactic approach.

Another limitation of the original matching algorithm by Chawathe *et al.* is its susceptibility to propagation of mismatches in small subtree structures and sub-optimalities whenever Matching Criterion 3 does not hold, *i.e.*, when there is more than one potential matching partner for a node present. Our improvements cover both aspects by computing the *best* rather than the *first* match for leaves. Unfortunately, it is still possible that our approach will fail under rare circumstances, which we describe informally as follows: Imagine, that there are two quite similar source code statements—statement a and statement b—and that a is updated between two subsequent revisions to a'. Otherwise, no changes were applied. Assume further, that a' is now more similar to b than to a. If this conditions are met, then our approach will possibly fail to match a with a' and instead considers b as the partner of a', depending on the tree structure.

However, under these rare circumstances, it is usually even for a human developer hard to classify the change relations correctly. Remember, the algorithm will always yield correct results—even under the occasions outlined above—but it might not be able to find a *minimum* conforming edit script. In most other cases, our algorithm performs well, as we will prove next, in particular in Chapter 4.

Chapter 4

Establishing a Benchmark

In Chapter 3, we have proposed several improvements for the change detection algorithm for hierarchically structured information presented by Chawathe *et al.*. The desired outcome was an algorithm that is customized for source code. Besides some artificial examples, we did not yet prove that our efforts are superior to the originally implemented solutions in CHANGEDISTILLER. We establish an elaborate benchmark in this chapter, before we will draw some final conclusions and give some perspectives on future work in Chapter 5.

Our benchmark for assessing the quality of our proposals is founded on two important issues: On the one hand, we use artificial test cases, that pin down tricky set-ups of statements. Although this approach comes very handy while prototyping, we realize that only real world data will put our improved algorithm to its paces. Hence, we integrated test data from the ARGOUML[1] project repository.

4.1 Requirements

The main requirement for designing an adequate benchmark that compares the original implementation of the change detection algorithm with our sophisticated solution, can be summarized as follows: While prototyping, we are in need of quick feedback on the success of our attempts to overcome the issues that arise, when we apply Chawathe *et al.*'s change detection algorithm to source code, instead of LaTeX documents. Our desired outcome, was a benchmark, that runs through fast and puts out a limited number of metrics, that allow us to compare the performance of the original algorithm with that of the modified one. Furthermore, we want to gain an idea how good the overall performance of our algorithm is, *i.e.*, we are curios about the distance between a theoretical minimum conforming edit-script and the outcome that we experience in practice.

Our benchmark does not have to test CHANGEDISTILLER for correctness—correctness is guaranteed by an extensive JUnit-testsuite—but, in order to be expressive, it is a necessary precondition that the test-cases run through without any errors or failures. We concentrate therefore on testing accuracy, as we will outline in Section 4.1.2.

Since CHANGEDISTILLER was implemented as a plug-in, it relies heavily on the infrastructure provided by ECLIPSE. It is not executable as a standalone program. Thus, the general framework for building our benchmark is restricted a priori to the ECLIPSE platform. Furthermore, we choose to exploit the JUNIT4-framework[2] for our concerns: JUNIT4 is the latest release of the well-known regression testing framework by Gamma and Beck and it is already integrated into the latest

[1] http://argouml.tigris.org/
[2] http://www.junit.org/

```
public void testPerformBenchmarkPart1(){
   a.b();
}
```

Listing 4.1: The identifier of test methods in JUnit 3.8 or earlier had to begin with `test`

```
@Test
public void performBenchmarkPart1(){
   a.b();
}
```

Listing 4.2: In JUnit 4, test methods are simply annotated by `@Test`

version of ECLIPSE. The following points illustrate the advantages, that made us choose this solution:

- **Hit and Run:** ECLIPSE provides a very useful test-runner for JUNIT, which is integrated into the development environment. The *plug-in development environment*, also known as the PDE-tools, take care that this even works for plug-ins. Hence, after the benchmark has been implemented, a single click will load the so called run-time workbench and the necessary plug-ins—including CHANGEDISTILLER and its dependencies, for instance EVOLIZERBASE and the classes from `org.eclipse.jdt`—before it finally executes the benchmark. No further user interactions are necessary. This fast and uncomplicated way of benchmarking is very useful, since we want to evaluate quickly several options, *e.g.*, different string and tree similarity measures, various settings for the parameters f and t from Matching Criterion 2 or 3, respectively.
- **Annotations:** Since Release 4, annotation-support has been added to JUNIT. This means that the test-runner no longer uses reflection to look for methods whose identifier starts with `test` but rather for methods that are annotated by `@Test`. The difference is shown in Listing 4.1 and Listing 4.2. Although this is not a must-have feature for our benchmark, it still opens possibilities: We use several classes of test data; If we, for example, choose to suppress all other parts of the benchmark than those that apply to small tree structures, we can turn off everything else in a clean and comfortable way, by just removing the according annotations or adding `@Ignore`-tags.
- **One-time set up and tear down:** The methods `setUp()` and `tearDown()` are past. They have been replaced by the annotations `@Before` and `@After`. Additionally, the annotations `@BeforeClass` and `@AfterClass` were added. Methods annotated by these tags run exactly once before benchmarking/testing starts, or once after it ends respectively. This is useful, especially when having several benchmark-methods annotated by `@Test`. Using the old `setUp()/tearDown()`-facility, set-up and tear-down code is executed each time a test method is invoked. Using the new mechanism, we are able to speed up our benchmark significantly.
- **Miscellaneous:** There are other new features, that make the use of JUNIT more convenient. Examples are *static imports* or *timeouts* for tests. We use neither one extensively, so we are not going to discuss this features more deeply.

4.1.1 Matchings, Change Operations or Classified Changes?

We described, what outcome we anticipate and what tools we use to build our benchmark. We did not yet propose what exactly we will benchmark. Since no other tools similar to CHANGEDISTILLER exist yet, that we can use as a reference, we have to decide between three options:

- The Matching Set
- Source Code Change Operations
- Classified Changes

Investigating the quality of the matching set is the most obvious approach, since we focussed on improving the matching algorithm, whose direct output is the matching set. For that, we have to manually investigate each statement in each revision n of a source code file and find this partners in revisions $n-1$ and $n+1$. If we are able to complete this task—which is virtually impossible for a significant number of files—we have to compare the expected matching set with the one that the algorithm puts out and decide how we shall rate the mismatches relative to the matches. We discard this option and turn to the next one.

After the matching set has been calculated, the algorithm extracts the raw source code change operations, *e.g.*, whether statement x has moved from parent y to parent z. Thus, the source code changes provide the next possible hook for our efforts to develop a benchmark. Again, we have to check each revision by hand to find the modifications compared to its predecessor. Furthermore, let us assume that the statement `System.out.println()` moves from an `if`-statement with the condition `a>b` to the top-level of the method body. To test the change operation that we *expect* against the change operation that our algorithm *finds*, we have to specify exactly the tree-node of the statement, including the type and value of the statement (method invocation with value '`System.out.println()`'), the original parent-node, including type and value (`if`-statement with the value `a>b`) and its new parent (root of the method body). Again, if done manually, this does not scale for larger numbers of files containing source code over several revisions.

What is left over, are the classified changes: The source code changes from the step before are analyzed, consolidated and classified by *change types* [FG06]. A move-operation for example, can be classified as a *statement ordering change* or as a *statement parent change*, depending on the character of the move. We still have to go through each revision of the test data and mark changes by hand, but the change types are much easier to find and to implement manually, than matchings or source code changes. This is due to the fact, that change types represent more or less the kind of changes, that a human will intuitively find, when he compares two subsequent versions of a file.

We can therefore make a decision: Usually, we will classify the change types manually, by looking through each revision of each source code file that we choose to be in the pool of test data. Regarding our few small classes used in the artificial test cases, we can be more precise and check whether the correct source code change operations are found.

Before we can conclude this section and proceed by defining the metrics that state whether the algorithm scores well or not, we have to prove that change types are an adequate instrument for benchmarking the algorithm:

There is one shortcoming in terms of change types: We cannot evaluate exactly, where the change occurred, since we do not store its exact location, but rather in which version and method or class it was found. This means that we can tell that *e.g.*, two *statement inserts* were found in method `foo()` between revision 1.11 and 1.12, but not whether the statements were for example inserted into a particular `if`-block or somewhere else. Nevertheless, we claim that comparing expected changes with changes found is sufficient for testing the accuracy of the algorithm, due to the following reasons:

We assume that classification of change types will always yield correct results—under any circumstances. We can be sure about this, since there is not much margin left and our test cases cover this issue in particular. Furthermore, an optimal matching will lead to an optimal amount of source code changes. This is also guaranteed by our test cases. Thus, we can state that if our matching algorithm works as intended, the complete change detection algorithm will produce the outcome, that we have defined by manual inspection. The opposite case is also derivable: Sub-optimal matchings will impact change type classification and produce an increased amount of detected changes. There is one exception: When pairs of inserts and deletes are mistakenly classified as updates, the number of detected changes will decrease unintentionally (since one update eliminates two operations: an insert and a delete). We can trace these situations by comparing the number of detected inserts/deletes and the number of detected updates with the number of expected changes of each type.

4.1.2 Accuracy - Precision and Recall

Precision and *Recall* are common for measuring the overall accuracy of information retrieval systems. Both measures were first proposed in [Cle67].

- ***Precision*** *is the ratio of the number of relevant records retrieved to the total number of irrelevant and relevant records retrieved. It is usually expressed as a percentage.* [LC06]

  We adapt the definition to our domain as follows:

$$Precision = \frac{\text{\# of changes found that were also classified manually}}{\text{Total \# of changes found}}$$

- ***Recall*** *is the ratio of the number of relevant records retrieved to the total number of relevant records in the database. It is usually expressed as a percentage.* [LC06]

  In terms of our work, we use:

$$Recall = \frac{\text{\# of changes found that were also classified manually}}{\text{Total \# of changes that were classified manually}}$$

The fact that our algorithm cannot 'retrieve' incorrect but only non-minimal results introduces some issues in terms of recall: Our approach is guaranteed to find all changes, leading to a recall of 100 percent—or even above 100 percent if we claim that also the operations that are not contained in the minimum conforming edit script are relevant. Recall does not seem to apply to our domain, since we can neither state that a higher recall is better nor that the opposite is true. Therefore we measure accuracy only by calculating precision.

4.1.3 Performance

We give an overall idea on the runtime performance of the change detection algorithm by measuring the effective time that it takes to calculate and classify all changes for our test data.

Change calculation is performed under MacOSX on a MacBook Pro, 2GHz Intel Core Duo with 2048M physical memory. The memory available to the Java Virtual Machine was restricted to 1024M. Test data for the benchmark is stored locally and results are written to a MySQL5 database via local loop back. The plug-in itself needs access to the CVS-repository of ArgoUML. We use a copy of the repository that is stored on a Linux server (Intel Pentium 4 CPU, 2.8 GHZ with 2048M physical memory) which resides in the local area network.

4.2 Choosing the Test Data

As stated in the introduction, we chose to use a combination of artificial test cases and real-world data from the ARGOUML-project. We will discuss how we have chosen the data and what preparation steps it has undergone. Furthermore, we present how both parts—the artificial test cases as well as the classes from ARGOUML—contribute to our benchmark.

4.2.1 Artificial Test Cases

We do not measure accuracy and performance for the test cases: They have to run through without errors or failures in order to be successful, since we used them for test-driven development of our improvements. The test cases represent simple issues that were objectionable in the originally implemented algorithm based solely on the work of Chawathe *et al.*. If the tested algorithm cannot accomplish them, we can claim that it does not meet our fundamental requirements. Each test case can be classified as addressing one of the following issues:

- Matching of values based on string similarities
- Matching of small tree structures
- Other 'special' concerns

We substantiate these classes next by giving some examples for test cases that belong to them.

Test Case for String Similarities

We have proposed the Dice Coefficient using 2-*grams* for achieving a more intuitive matching of node values. Review Section 3.5.2 for details.

We use classes similar to the ones in Listing 4.3 to test measurement on typical renaming refactorings. In this particular example, we check whether the following source code change operations were classified correctly:

- Update of the *field declaration* from:

  `treeDifferencersAlgorithmName`

  To:

  `nameOfTreeDifferencingAlgorithm`

- Update of the *method declaration* from:

  `verticalBarPrint`

  To:

  `printVerticalBar`

- Update of the *variable declaration statement* from:

  `DrawVerticalAction dva = new DrawVerticalAction(actionName);`

  To:

  `VerticalDrawAction vdAction = new VerticalDrawAction(actionName);`.

```
//Left
public class Test {
   private String treeDifferencersAlgorithmName = "algo";

   public void verticalBarPrint() {
      System.out.println("------");
   }

   public void aMethod(String actionName) {
      DrawVerticalAction dva = new DrawVerticalAction(actionName);
   }
}

//Right
public class Test {
   private String nameOfTreeDifferencingAlgorithm = "algo";

   public void printVerticalBar() {
      System.out.println("------");
   }

   public void aMethod(String actionName) {
      VerticalDrawAction vdAction = new VerticalDrawAction(actionName);
   }
}
```

Listing 4.3: Data for the first test case. The class denoted by 'Left' is the original version, while 'Right' denotes the modified one.

```
//Left
public class Test {
   public void aMethod() {
      if (aString.equals("test_string")) {
         System.out.println("test_string");
      } else {
         Math.abs(-10);
      }
   }
}

//Right
public class Test {
   public void aMethod() {
      if (aString.equals("test_string")) {
         System.out.println(
         "test_string_new_string_which_is_to_long_
          to_match_even_for_DiceCoefficient"
         );
      } else {
         Math.abs(-10);
      }
   }
}
```

Listing 4.4: Data for the second test case.

Test Case for Small Trees

Small tree structures are often a problem when the approach by Chawathe *et al.* is applied to source code. We have discussed this issue in Section 3.4.

Test cases which use data such as listed in Listing 4.4 decide whether the benchmarked algorithm handles them correctly at least in trivial cases. The following two source code change operations must be found:

- Delete of the *method invocation* representing:

  ```
  System.out.println("test_string");
  ```

- Insert of the *method invocation* representing:

  ```
  System.out.println(
  "test_string_new_string_which_is_to_long_
  to_match_even_for_DiceCoefficient");}
  ```

```
//Left
public class Test {
   public void aMethod(MAssociationEnd from, MAssociation to) {
      System.out.print("Placeholder");
      System.out.print("Placeholder");
      System.out.print("Placeholder");
      System.out.print("Placeholder");
      System.out.print("Placeholder");
   }
}

//Right
public class Test {
   public void aMethod(Object from, MAssociation to) {
      System.out.print("Placeholder");
      System.out.print("Placeholder");
      System.out.print("Placeholder");
      System.out.print("Placeholder");
      System.out.print("Placeholder");
   }
}
```

Listing 4.5: Data for the third test case.

Test Case for a Special Concern — Parameter Ordering Change

During our research, mainly while using data from ARGOUML, we have encountered some interesting situations where the algorithm by Chawathe *et al.* fails terribly. One example is an unfortunate parameter ordering change such as shown in Listing 4.4. This configuration of source code statements is interesting: The type of the parameter `from` changes from `MAssociationEnd` to `Object`. This causes a mismatch, since the strings have nothing in common. The crux in this case, is the type of the second parameter; It is very similar to the first one. In other words, Matching Criterion 3, discussed first in Section 3.3, does not hold. We illustrate the situation in Figure 4.1 and discuss the implications below, when we apply the finished benchmark to the different versions of the algorithm.

The `System.out.print("Placeholder")`-statements were inserted to ensure that the plugin `org.eclipse.compare` is able to match the methods,[3] but they show also a side-effect that is very welcome: Since the five statements are identical, there are theoretically 5^2 possibilities to match them. The test case checks therefore whether the similarity-rating algorithm is stable *i.e.*, if it changes the ordering of leaves that have the same similarity.

[3]In case that the method declaration changes, we use the whole method body, represented as string, to identify the same method in subsequent revisions of a source code file.

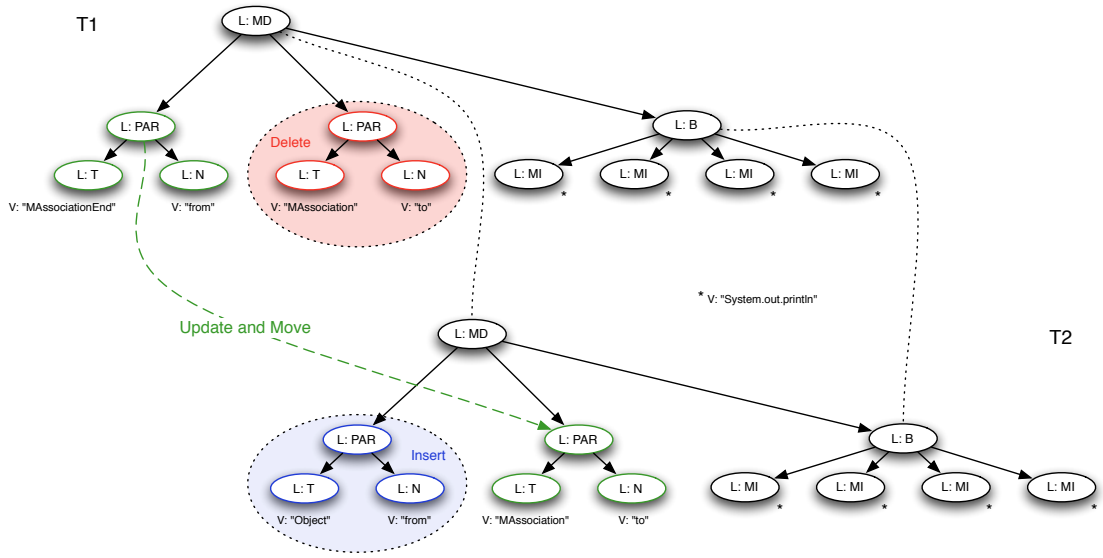

Figure 4.1: Tree representation of the classes from Listing 4.5. MD denotes a method declaration, PAR denotes a parameter declaration. T denotes type, N denotes name, B denotes the method body, and MI denotes a method invocation.

4.2.2 Real Life Data from ArgoUML

Artificial test cases are well-suited to investigate specific or theoretical issues. They are insufficient for claiming whether an approach applies to real world problems or not. Therefore, we decided to integrate data form a real mid-sized project. Further criteria are that the software is open source, that it has undergone some evolution, and that the development team uses CVS in a disciplined manner, *i.e.*, that there are enough changes, distributed over a significant amount of revisions. The ARGOUML-projects fits well into this schema, because a small team of people has developed it further actively during several years. Since then, the tool has reached a size of several ten thousand lines of code.

Choosing representative test data among the about 1400 classes is challenging. We applied several heuristics and tools to find appropriate classes:

- **A lot of changes over time, few changes between revisions:** We have used the *CVS Repository Exploring* perspective in ECLIPSE to locate classes that have about 100 to 200 revisions. Furthermore, we looked through the corresponding CVS-logs and investigated the classes manually. The *compare with...*-functionality in ECLIPSE has proved itself helpful for this task. We preferred classes which contain methods that show about ten to twenty changes per revision.

- **Medium sized methods:** The larger a method, the more likely it is, that it contains interesting structures and that it has undergone an evolution. Small examples are already covered by our artificial test cases. Methods that are to big are difficult to classify by hand. We have looked therefore for sizes in-between.

- **Hierarchies:** Methods that have a big nested statement depth are most interesting in terms of the small-subtree-problem.

- **Diversity of changes:** We used the original implementation of CHANGEDISTILLER to restrict the number of candidates further. The *class- and method statistics* views as well as the *class change history* gave us a general impression where the hotspots in terms of changes are located within ARGOUML. We preferred classes with different types of changes, since we want to benchmark a broad variety of structures. We had to inspect the candidates again by hand; By doing this, we were able to locate files where the edit script was far from minimal. These classes are particularly interesting: If our work leads to a significant decrease in the number of detected changes, we can claim that we have reached an improvement.
- **Significance levels:** Changes that have a high significance level are critical. If to many—or to few—of them are found, the quality and the expressiveness of the approach will suffer. Classes, where high significance levels occur, are therefore candidates for our benchmark data.

According to the heuristics above, we were able to locate four interesting methods—each one in another class—that we have integrated into our benchmark. We performed a checkout of every revision in which the selected methods experienced changes. Preparation of the classes was done by deleting all fields and methods except the chosen ones. During manual inspection, we have finally classified 447 changes in a total of 111 revisions. We introduce the methods briefly:

- **org.argouml.uml.ui.ActionOpenProject.actionPerformed (ActionEvent) (37 revisions):** The class is responsible for loading projects. The method is involved into event-handling and covers most of the functionality of the class. The example shows a typical sequence of evolutionary steps: The method grows incrementally, experiences several smaller bug-fixes, and becomes less maintainable until revision 1.18 where, eventually, major refactoring steps were applied. It has an impressive nesting depth with a lot of `if`-statements and exception-handling.
- **org.argouml.uml.diagram.static_structure.ui.FigClass.getPopUpActions(MouseEvent) (32 revisions):** With a total of about 200 revisions, the class is one of the most changing ones in the whole project. It is responsible for displaying graphics for a UML Class in a diagram. The method builds a collection of menu items relevant for a right-click popup menu on a package. It contains several small structures for challenging our algorithm.
- **org.argouml.uml.reveng.java.Modeller.addOperation(short, String, String, Vector, String) (36 Revisions):** The class is responsible for building the UML model from data received by a source code parser. The method is invoked whenever the parser detects an operation that has to be added to the model. The structure is similar to the one described for `FigClass.getPopUpActions`: There are `for`-loops, `if`-statements and `try/catch`-blocks.
- **org.argouml.persistence.ZargoFilePersister.loadProject(URL) (6 Revisions):** The class is responsible for file storage. The method loads the project into memory that resides under a given url. We classify only declaration changes of the method. The body has therefore been deleted and was replaced by dummy statements.

The artificial test cases as well as the benchmark based on real data are realized as common JUNIT test cases.

4.3 Results

We are able to prove the claims that we have made during Chapter 3: We stated that calculating string similarities using the Dice Coefficient with 2-*grams* allows more intuitive matching of leaves than Levenshtein does. We claimed furthermore, that the original change detection algorithm by Chawathe *et al.* does not apply well to the characteristics of source code. We proposed an alternative—the so called *BestMatch*-Algorithm. We speculated further, which tree similarity measure, the one used by Chawathe *et al.* or the Dice-based one suggested by Baxter *et al.*, yields better results in terms of inner node matching.

We present and discuss selected comparisons between different configurations of algorithms, *i.e.*, we show how the different approaches perform against each other: We benchmark different combinations of:

- The original matching algorithm for leaves by Chawathe *et al.* or our *BestMatch* respectively.
- Either the tree similarity measure suggested by Chawathe *et al.*, or the one using Dice are used for inner node comparisons.
- The 'Dynamic thresholds'-approach, as long as we do not use Dice to calculate the inner node similarity. We lower the threshold t to 0.5 whenever the left and the right tree roots have four or less descendants.
- We combine the 'base' algorithms with either Levenshtein- or the Dice-based string similarity measure for matchings leaf-values.

We fix the thresholds at $f = 0.7$ and $t = 0.6$ if not stated otherwhise. Further benchmark results can be found in Appendix A.

4.3.1 Running the Artificial Test Cases

Remember, in Section 4.2.1 we have defined three different classes of problems that are covered by particular test cases: String matching issues, small subtrees and a particular special concern. They are the first barrier that our test candidates have to overcome. We can state generally that configurations using Levenshtein do not pass the first test, *i.e.*, the one that involves renaming of identifiers. The approach by Chawathe *et al.* fails on the small tree structures, unless it uses the Dice Coefficient to calculate inner node similarity or unless it is enhanced with dynamic thresholds. Very interesting is our special concern: The parameter type change (see Figure 4.5), where the type of the first parameter has changed from `MAssociationEnd` to `Object`. Each configuration, except the ones that use our *BestMatch*-approach, fails terribly on this challenge. They match the first parameter of the original version with the second parameter of the modified version, which is not an intuitive matching and leads consequently to an exchange of `MAssociationEnd from` with `MAssociation to`, *i.e.*, to unnecessary update and move operations. The right parameter in the original version is deleted completely, whereas the first parameter in the modified version is inserted again. As indicated, *BestMatch* passes this challenge with flying colors by detecting that there is a better match for `MAssociationEnd` than `MAssociation`.

4.3.2 Declaration Changes

None of the algorithm configurations has experienced problems while classifying declaration changes. All of them were able to detect each change type correctly. Nevertheless, in order to be able to generalize this result, we have to find test data with a lot of tricky declaration changes. This will be subject to future work.

4.3.3 Body Changes

In this section, we confront the different approaches with various changes taken from the classes out of the real life project ARGOUML. We start benchmarking by investigating the unmodified algorithm by Chawathe *et al.* as it was originally implemented in CHANGEDISTILLER. Its accuracy is shown in Table 4.1.

Operation	Expected	Found	Precision
Condition Expression Change	33	24	1.0*
Alternative Part Insert	2	13	0.15
Alternative Part Delete	0	14	0
Statement Update	181	133	1.0*
Statement Insert	96	205	0.47
Statement Delete	94	196	0.48
Statement Ordering Change	3	44	0.07
Statement Parent Change	31	55	0.56
Total	440	684	0.64

Table 4.1: Original algorithm by Chawathe et. al., using Levenshtein. $f = 0.7$. Distilling took 8053 ms.

Values annotated by '*' illustrate the limitations that arise when focussing solely on precision as a measure for accuracy. The value of 1.0 suggests that the performance is very good, although we have missed a lot of changes of some particular types. To be exact, we did not *miss* any changes, but we did not find the minimal amount of operations to apply them. Calculating recall would be helpful to qualify precision in this case. But remember, we are not as much interested in the precision in terms of a particular change types, as we are in the total amount of changes, since we use precision to measure minimality. This limitations are taken into account by discussing the situation in detail whenever precisions reaches 1.0.

For example: Instead of applying the missing *condition expression changes*, *i.e.*, updates on the conditions of `if`-statements, we have deleted whole `if`-blocks. Then we have re-inserted very similar ones instead, as we can infer when we look at the huge amount of *statement inserts* and *deletes*. The same accounts for *alternative part inserts/deletes*. Furthermore, we missed a lot of *updates*. Again, manual inspection shows, that the algorithm classified pairs of *statement inserts* and *statement deletes* instead. A lot of unintended *Statement Ordering and Parent Changes* are an evidence for second-best matches, or for problems while handling small subtree structure respectively.

We conclude: The original algorithm by Chawathe *et al.* is not able to match nodes accurately. Total precision is only 0.64 and we have found about 250 redundant changes.

In Table 4.2, we have exchanged Levenshtein through the Dice Coefficient using $2\text{-}grams$:

Operation	Expected	Found	Precision
Condition Expression Change	33	25	1.0
Alternative Part Insert	2	8	0.25
Alternative Part Delete	0	9	0
Statement Update	181	166	1.0
Statement Insert	96	163	0.58
Statement Delete	94	154	0.61
Statement Ordering Change	3	29	0.10
Statement Parent Change	31	55	0.56
Total	440	609	0.72

Table 4.2: Original algorithm by Chawathe et. al., using Dice with $2\text{-}grams$ for string similarities. $f = 0.7$. Distilling took 4815 ms.

We can recognize a significant improvement: Overall precision has been increased by eight percent in contrast to the first configuration. It has reached 0.72. The number of *statement inserts* and *statement deletes* in sum has decreased in about the same degree that the number of *statement updates* was increased. Remember, each update eliminates one delete and one insert. Besides this improvement, the classified changes remained about the same. Minor fluctuations can be traced back to the instance that better leaf matching (which we have reached due to the customized string similarity measure) reduces propagation of mismatches to higher levels. Note that, the time that distilling takes, was reduced from about 8000 ms to only 4000 ms when using Dice instead of Levenshtein—thus runtime was reduced by 50%.

Improving string matching was not the only proposal for customizing change detection for source code; We also suggested that, the Dice-based tree similarity measure used by Baxter *et al.* is theoretically better suited for source code. Table 4.3 shows whether this assumption holds.

Operation	Expected	Found	Precision
Condition Expression Change	33	40	0.83
Alternative Part Insert	2	8	0.25
Alternative Part Delete	0	9	0
Statement Update	181	140	1.0
Statement Insert	96	187	0.51
Statement Delete	94	178	0.52
Statement Ordering Change	3	35	0.09
Statement Parent Change	31	48	0.65
Total	440	645	0.68

Table 4.3: Base algorithm by Chawathe et. al., using Dice for inner nodes and Levenshtein. $f = 0.7$. Distilling took 7924 ms.

The first impression is promising: The precision increased by 4% in contrast to the first configuration. But take a closer look at the particular changes: Dice seems to be a mixed blessing for the calculation of the inner node similarity. On the one hand, it is able to improve some numbers,

such as that *statement ordering* and *parent changes* are reduced. On the other hand, it shows too much tolerance in terms of *condition expression changes*: Although we were only able to find 33 *condition expression changes* during manual inspection, the algorithm detects as much as 40. We assume that it matches pairs of `if`-blocks that are not intended to match, which results in updated conditions. We were not yet able to fully dissolve these issues. This will be subject to further investigations. Nevertheless, we can claim that the approach—or at least our implementation—is not perfectly suited for our concerns.

For investigating the performance of our *BestMatch*-approach, we discuss three more configurations: The results for the first one can be found in Table 4.4. It uses the Levenshtein Distance for matching leafs, or the tree similarity measure by Chawathe *et al.* for inner nodes respectively. In Table 4.5, the second-to-last configuration additionally involves dynamic thresholds. Eventually, Table 4.6 shows the results of the last configuration. Its setup is identical to the former one, except, that it uses Dice using 2-$grams$ instead of Levenshtein and that threshold f has been lowered to 0.6. We recommend to use this configuration in CHANGEDISTILLER.

Operation	Expected	Found	Precision
Condition Expression Change	33	23	1.0
Alternative Part Insert	2	13	0.15
Alternative Part Delete	0	14	0
Statement Update	181	126	1.0
Statement Insert	96	205	0.47
Statement Delete	94	196	0.48
Statement Ordering Change	3	46	0.07
Statement Parent Change	31	53	0.58
Total	440	676	0.65

Table 4.4: Base algorithm: BestMatch, using tree similarity by Chawathe et. al. and Levenshtein. $f = 0.7$. Distilling took 27964 ms.

In Table 4.4 and Table 4.5 some changes were classified more precisely but the improvement in contrast to the first configuration is not significant. Furthermore, the runtime-performance decreased by a factor of four, which bears no proportion to the gains in precision.

Operation	Expected	Found	Precision
Condition Expression Change	33	23	1
Alternative Part Insert	2	13	0.15
Alternative Part Delete	0	14	0
Statement Update	181	127	1.0
Statement Insert	96	202	0.48
Statement Delete	94	193	0.49
Statement Ordering Change	3	43	0.07
Statement Parent Change	31	53	0.58
Total	440	668	0.66

Table 4.5: Base algorithm: BestMatch, using dynamic thresholds, tree similarity by Chawathe et. al., and Levenshtein. $f = 0.7$. Distilling took 28587 ms.

Operation	Expected	Found	Precision
Condition Expression Change	33	28	1.0
Alternative Part Insert	2	8	0.25
Alternative Part Delete	0	9	0
Statement Update	181	177	1.0
Statement Insert	96	131	0.73
Statement Delete	94	122	0.8
Statement Ordering Change	3	26	0.12
Statement Parent Change	31	58	0.53
Total	440	559	0.79

Table 4.6: Base algorithm: BestMatch, using dynamic thresholds, tree similarity by Chawathe et. al., and Dice with 2-$grams$ for string similarities. $f = 0.6$. Distilling took 6106 ms.

Table 4.6 summarizes all the improvements that we are able to achieve. Precision increased by as much as 15%. Furthermore, by getting rid of the Levenshtein Distance, we were able to reduce runtime by a factor of five. The most interesting fact is that our improvements showed synergetic behaviour: When we used the Dice Coefficient with 2-$grams$ for calculating string similarities in combination with the original approach by Chawathe *et al.*, we were able to improve precision by eight percent. *BestMatch* on its own yields an improvement of one percent. Eventually, dynamic thresholds raised precision by two percent. Applying all three of them together and lowering the threshold slightly[4] does not bring an improvement of about eleven percent, as one might think, but rather increases precision by 15 percent, which is impressive.

4.4 Conclusions and Limitations

We have achieved significant improvements for almost all change types: The most impressive success concerns *statement updates*, where we were able to detect 177 out of 181 using our last configuration. As a consequence, the number of *statement inserts/deletes* decreased by magnitudes. We can trace this back to the intuitive string similarity scoring that the Dice Coefficient with 2-$grams$ provides.

The Dice Coefficient integrates well into *BestMatch*, our similarity ranking algorithm for leaves, which addresses mainly situations where Matching Criterion 3 does not hold. We also enhanced *BestMatch* with dynamic thresholds. By doing so, we can handle small subtrees more accurate than before.

On the other side, the shortcomings of our work can be described as follows:

- The Dice-based tree similarity measure used by Baxter *et al.* produces many false negatives in terms of inner node matching.
- Our benchmark itself yields some insufficiencies too: Its results need a lot of interpretation and time-consuming manual verification in order to be expressive. Finding better metrics for measuring accuracy improves this situation.

Nevertheless, future work can rely on the test data, *i.e.*, on the artificial test cases as well as on the classes taken from ARGOUML, since the data covers most structures that our algorithm will encounter 'in the wild'.

[4]See Appendix Chapter A for more test data where $f = 0.6$ was also applied to the other configurations. For example, lowering the threshold for the original algorithm by Chawathe *et al.* in combination with the Dice Coefficient for calculating string similarities, increased precision by one percent.

Chapter 5

Conclusions

5.1 Summary of Contribution

In this thesis, we have investigated whether the approach for *change detection in hierarchically structured information* by Chawathe *et al.* is an adequate way to track changes in source code. First, we have pointed out the differences according to source code change detection between source code and structured documents in general. The implications can be summarized as follows:

- We need a two-staged algorithm that ignores order on the class-level, but takes into account that statements are ordered on a method-level.
- Labels of tree nodes representing source code do not satisfy the acyclic label condition under most circumstances. Therefore we cannot exploit them to improve the results of source code change detection algorithms.
- The Matching Criterion 3, defined by Chawathe *et al.*, holds infrequently for source code statements. In most of the encountered examples, it could not be satisfied.
- We claimed also that the Levenshtein Distance does not allow intuitive similarity scoring on statements, leading to unintended mismatches.
- In conjunction with the Matching-Criterion-3-problem, we can notice inordinate propagation of mismatches to higher levels in the tree, especially when the algorithm has to deal with small subtree structures.

We were able to prove that these insufficiencies lead to sub-optimalities in the matching set and therefore to a non-minimal conforming edit script of source code change operations. To overcome this issue, we have evaluated different options for string and tree similarity measures, as well as customized matching algorithms and eventually, we propose the following improvements:

- **More intuitive leaf matching:** The Dice Coefficient using $2\text{-}grams$ scores string similarities more intuitively than the Levenshtein Distance does.
- **Dynamic thresholds:** Setting the thresholds for inner node matching according to the number of descendants, reduces the number of mismatches on small trees significantly.
- **BestMatch:** We have introduced an extended version of the algorithm by Chawathe *et al.* for establishing similarity-ranking of leaves. This allows us to overcome most of the issues that are related to Matching Criterion 3.

Last but not least, we have developed an extensive benchmark to evaluate our approach. The benchmark combines artificial test cases, as well as data taken from a real medium-sized project.

We were able to improve matching significantly for all types of changes. We have raised overall accuracy by 15 percent, *i.e.*, from 0.64 to 0.79. For example, using our approach, we classify about 98 percent of all updates in our test data correctly. CHANGEDISTILLER still finds too many unintended *statement ordering changes* and we are currently investigating how we can improve the change detection algorithm further.

5.2 Lessons Learned

In developing improvements for CHANGEDISTILLER, we have learned that a good string similarity measure is crucial in order to achieve a good matching between leaves in subsequent revisions of a source code file. We noticed that a lot of refactorings, such as renaming identifiers, lead to a change in the order of words or characters. Metrics, that are based on the longest common subsequence approach, such as the Levenshtein Distance, fail to handle this adequately. While using the Dice Coefficient with $2\text{-}grams$, we do not overrate character ordering but rather focus on common character pairs of two strings.

In contrast, tree similarity measures are not that important, since changes to the structure of the program seem to occur less frequently than changes to single statements. In return, whenever the tree similarity unintentionally fails to match inner nodes, it is very likely that a lot of unnecessary operations will be found, because whole subtrees have to be replaced in these cases.

5.3 Future Work

Future work includes a closer investigation of the impact of the string similarity threshold f, and the inner node threshold t respectively, on the quality of the matching set. An interesting point of research can be found by introducing similarity-ranking to inner nodes. This is not straightforward and will possibly involve a fundamental change on how the algorithm works. Furthermore, we will extend our source code change detection to find changes in documentation that belong to a source code unit. We will investigate whether changes to *e.g.*, a method body usually trigger changes in documentation.

By investigating the similarity of the targets of *e.g.*, a method invocation statement, we will be able to improve matching. For this, we have to resolve bindings. This implies that source code has to compile in order to allow more sophisticated change analysis.

Future work on the benchmark will involve better metrics for measuring accuracy and additional test data from real projects other than ARGOUML. We will extend the artificial test cases, whenever we encounter challenging examples.

Using insights from machine learning yields interesting perspectives: If we assume that more or less the same persons are involved into development on a particular project over time, we can expect that we will encounter specific nomenclature, idioms, and patterns. We can take them into account by using adaptive similarity measures. The measures will possibly improve the outcome of our algorithm, once they are trained. Another approach related to this subject, aims at integrating the functionality of CHANGEDISTILLER directly into a versioning system. Whenever a developer commits a source code unit, changes are calculated and—optionally—presented for verification. Possible corrections, applied by the developer, are used to detect and classify future changes more accurately.

Last but not least, we suggest further research on a topic that is not directly related to our work, but on the EVOLIZER-platform as a whole: During prototyping and evaluation, it is crucial

to locate quickly data that is stored in the release history database. Since the RHDB covers most of the information that is relevant to a project's life cycle, unexperienced users and developers are faced with an enormous amount of entries and therefore, they will possibly find it hard to locate the information that they are interested in. Using semantic web features, *e.g.*, such as the quasi natural language querying presented in [BKKK06] can guide them through the huge amount of data.

Appendix A

Additional Benchmark Data

Operation	Expected	Found	Precision
Condition Expression Change	33	25	1.0
Alternative Part Insert	2	9	0.22
Alternative Part Delete	0	10	0
Statement Update	181	161	1.0*
Statement Insert	96	171	0.56
Statement Delete	94	162	0.58
Statement Ordering Change	3	33	0.09
Statement Parent Change	31	56	0.55
Total	440	627	0.64

Table A.1: Original algorithm by Chawathe et. al., using Levenshtein. $f = 0.6$. Distilling took 7673 ms.

Operation	Expected	Found	Precision
Condition Expression Change	33	25	1.0
Alternative Part Insert	2	9	0.25
Alternative Part Delete	0	10	0
Statement Update	181	196	1.0
Statement Insert	96	140	0.58
Statement Delete	94	131	0.61
Statement Ordering Change	3	23	0.10
Statement Parent Change	31	67	0.56
Total	440	601	0.73

Table A.2: Original algorithm by Chawathe et. al., using Dice with $2\text{-}grams$ for string similarities. $f = 0.6$. Distilling took 4779 ms.

Operation	Expected	Found	Precision
Condition Expression Change	33	24	1.0
Alternative Part Insert	2	13	0.15
Alternative Part Delete	0	14	0
Statement Update	181	134	1.0
Statement Insert	96	203	0.47
Statement Delete	94	194	0.48
Statement Ordering Change	3	41	0.07
Statement Parent Change	31	55	0.56
Total	440	678	0.65

Table A.3: Original algorithm by Chawathe et. al., using dynamic thresholds, Levenshtein. $f = 0.7$. Distilling took 8735 ms.

Operation	Expected	Found	Precision
Condition Expression Change	33	27	1.0
Alternative Part Insert	2	9	0.15
Alternative Part Delete	0	10	0
Statement Update	181	161	1.0
Statement Insert	96	168	0.47
Statement Delete	94	159	0.48
Statement Ordering Change	3	31	0.07
Statement Parent Change	31	55	0.56
Total	440	620	0.71

Table A.4: Original algorithm by Chawathe et. al., using dynamic thresholds, Levenshtein. $f = 0.6$. Distilling took 7336 ms.

Operation	Expected	Found	Precision
Condition Expression Change	33	25	1.0
Alternative Part Insert	2	8	0.25
Alternative Part Delete	0	9	0
Statement Update	181	166	1.0
Statement Insert	96	162	0.59
Statement Delete	94	153	0.61
Statement Ordering Change	3	27	0.11
Statement Parent Change	31	55	0.56
Total	440	605	0.73

Table A.5: Original algorithm by Chawathe et. al., using dynamic thresholds, Dice with 2-$grams$ for string similarities. $f = 0.7$. Distilling took 4456 ms.

Operation	Expected	Found	Precision
Condition Expression Change	33	26	1.0
Alternative Part Insert	2	9	0.22
Alternative Part Delete	0	10	0
Statement Update	181	196	0.9
Statement Insert	96	137	0.70
Statement Delete	94	128	0.73
Statement Ordering Change	3	22	0.14
Statement Parent Change	31	68	0.46
Total	440	596	0.74

Table A.6: Original algorithm by Chawathe et. al., using dynamic thresholds, Dice with $2\text{-}grams$ for string similarities. $f = 0.6$. Distilling took 4615 ms.

Operation	Expected	Found	Precision
Condition Expression Change	33	43	0.76
Alternative Part Insert	2	5	0.4
Alternative Part Delete	0	6	0
Statement Update	181	166	1.0
Statement Insert	96	154	0.62
Statement Delete	94	145	0.64
Statement Ordering Change	3	34	0.09
Statement Parent Change	31	47	0.66
Total	440	600	0.73

Table A.7: Base algorithm by Chawathe et. al., using Dice for inner node similarity and Levenshtein. $f = 0.6$. Distilling took 7257 ms.

Operation	Expected	Found	Precision
Condition Expression Change	33	43	0.76
Alternative Part Insert	2	4	0.5
Alternative Part Delete	0	5	0
Statement Update	181	171	1
Statement Insert	96	146	0.66
Statement Delete	94	137	0.69
Statement Ordering Change	3	30	0.1
Statement Parent Change	31	46	0.67
Total	440	582	0.76

Table A.8: Base algorithm by Chawathe et. al., using Dice for inner node and string similarities. $f = 0.7$. Distilling took 4408 ms.

Operation	Expected	Found	Precision
Condition Expression Change	33	44	0.75
Alternative Part Insert	2	2	1.0
Alternative Part Delete	0	3	0
Statement Update	181	201	0.9
Statement Insert	96	120	0.8
Statement Delete	94	111	0.85
Statement Ordering Change	3	30	0.1
Statement Parent Change	31	58	0.53
Total	440	569	0.78

Table A.9: Base algorithm by Chawathe et. al., using Dice for inner node and string similarities. $f = 0.6$. Distilling took 4324 ms.

Operation	Expected	Found	Precision
Condition Expression Change	33	26	1.0
Alternative Part Insert	2	9	0.22
Alternative Part Delete	0	10	0
Statement Update	181	150	1.0
Statement Insert	96	169	0.57
Statement Delete	94	160	0.59
Statement Ordering Change	3	35	0.086
Statement Parent Change	31	53	0.58
Total	440	612	0.72

Table A.10: Base algorithm: BestMatch, tree similarity by Chawathe et. al. and Levenshtein. $f = 0.6$. Distilling took 28286 ms.

Operation	Expected	Found	Precision
Condition Expression Change	33	23	1.0
Alternative Part Insert	2	10	0.2
Alternative Part Delete	0	11	0
Statement Update	181	155	1.0
Statement Insert	96	164	0.59
Statement Delete	94	155	0.60
Statement Ordering Change	3	31	0.1
Statement Parent Change	31	58	0.53
Total	440	607	0.72

Table A.11: Base algorithm: BestMatch, tree similarity by Chawathe et. al. and Dice with $2\text{-}grams$ for string similarities. $f = 0.7$. Distilling took 6001 ms.

Operation	Expected	Found	Precision
Condition Expression Change	33	28	1.0
Alternative Part Insert	2	8	0.25
Alternative Part Delete	0	9	0
Statement Update	181	176	1.0
Statement Insert	96	133	0.72
Statement Delete	94	124	0.75
Statement Ordering Change	3	28	0.11
Statement Parent Change	31	58	0.53
Total	440	564	0.78

Table A.12: Base algorithm: BestMatch, tree similarity by Chawathe et. al. and Dice with $2\text{-}grams$ for string similarities. $f = 0.6$. Distilling took 6084 ms.

Operation	Expected	Found	Precision
Condition Expression Change	33	26	1.0
Alternative Part Insert	2	9	0.22
Alternative Part Delete	0	10	0
Statement Update	181	151	1.0
Statement Insert	96	167	0.57
Statement Delete	94	158	0.59
Statement Ordering Change	3	33	0.09
Statement Parent Change	31	53	0.58
Total	440	607	0.72

Table A.13: Base algorithm: BestMatch, using dynamic thresholds, tree similarity by Chawathe et. al. and Levenshtein. $f = 0.6$. Distilling took 27414 ms.

Operation	Expected	Found	Precision
Condition Expression Change	33	40	0.83
Alternative Part Insert	2	8	0.25
Alternative Part Delete	0	9	0
Statement Update	181	134	1.0
Statement Insert	96	188	0.51
Statement Delete	94	179	0.53
Statement Ordering Change	3	39	0.08
Statement Parent Change	31	53	0.58
Total	440	650	0.68

Table A.14: Base algorithm: BestMatch, Dice for inner node similarity and Levenshtein. $f = 0.7$. Distilling took 27304 ms.

Operation	Expected	Found	Precision
Condition Expression Change	33	42	0.79
Alternative Part Insert	2	5	0.4
Alternative Part Delete	0	6	0
Statement Update	181	158	1.0
Statement Insert	96	155	0.62
Statement Delete	94	146	0.64
Statement Ordering Change	3	39	0.08
Statement Parent Change	31	53	0.58
Total	440	604	0.73

Table A.15: Base algorithm: BestMatch, Dice for inner node similarity and Levenshtein. $f = 0.6$. Distilling took 27595 ms.

Operation	Expected	Found	Precision
Condition Expression Change	33	42	0.79
Alternative Part Insert	2	4	0.5
Alternative Part Delete	0	5	0
Statement Update	181	165	1.0
Statement Insert	96	146	0.66
Statement Delete	94	137	0.69
Statement Ordering Change	3	34	0.09
Statement Parent Change	31	54	0.57
Total	440	587	0.75

Table A.16: Base algorithm: BestMatch, Dice for inner node and string similarities. $f = 0.7$. Distilling took 6002 ms.

Operation	Expected	Found	Precision
Condition Expression Change	33	43	0.77
Alternative Part Insert	2	2	1.0
Alternative Part Delete	0	3	0
Statement Update	181	186	0.97
Statement Insert	96	121	0.79
Statement Delete	94	112	0.84
Statement Ordering Change	3	34	0.09
Statement Parent Change	31	58	0.53
Total	440	559	0.79

Table A.17: Base algorithm: BestMatch, Dice for inner node and string similarities. $f = 0.6$. Distilling took 5999 ms.

Appendix B

Contents of CD-ROM

We provide a CD-ROM with each copy of this thesis. The contents are:

- **Thesis.pdf:** Copy of the written elaboration of this thesis.
- **Abstract.pdf:** Abstract of the thesis in English.
- **Zusfsg.pdf:** Abstract of the thesis in German.
- **org.evolizer.astdiff.zip:** Improved version of CHANGEDISTILLER.
- **org.evolizer.astdiff.test.zip:** Test cases for CHANGEDISTILLER.
- **org.evolizer.astdiff.benchmark.zip:** Benchmark for CHANGEDISTILLER.

References

[AB74] George W. Adamson and Jillian Boreham. The use of an association measure based on character structure to identify semantically related pairs of words and document titles. *Information Storage and Retrieval*, 10(7-8):253–260, 1974.

[AOH04] Taweesup Apiwattanapong, Alessandro Oros, and Mary Jean Harrold. A differencing algorithm for object-oriented programs. In *Proceedings of the 19th International Conference on Automated Software Engineering (ASE'04)*, 2004.

[BKKB05] Abraham Bernstein, Esther Kaufmann, Christoph Kiefer, and Christoph Bürki. SimPack: A Generic Java Library for Similiarity Measures in Ontologies. Technical report, Department of Informatics, University of Zurich, 2005.

[BKKK06] Abraham Bernstein, Esther Kaufmann, Christian Kaiser, and Christoph Kiefer. Ginseng: A guided input natural language search engine for querying ontologies. In *2006 Jena User Conference*, May 2006.

[BYM+98] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees, 1998.

[Cle67] C. Cleverdon. The cranfield tests on english language devices. In *Aslib Proceedings*, volume 19, pages 173–194. Aslib, 6 1967.

[CRF03] W. Cohen, P. Ravikumar, and S. Fienberg. A comparison of string distance metrics for name-matching tasks, 2003.

[CRGMW96] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 493–504, New York, NY, USA, 1996. ACM Press.

[Dic45] Lee R. Dice. Measures of the amount of ecologic association between species. *ESA Ecology*, (26):297–302, 1945.

[DTS99] S. Demeyer, S. Tichelaar, and P. Steyaert. Famix – the famoos information exchange model, 1999.

[FG06] Beat Fluri and Harald C. Gall. Classifying change types for qualifying change couplings. In *Proceedings of the 14th International Conference on Program Comprehension (ICPC)*, pages 35–45, Athen, Greece, June 2006. IEEE Computer Society Press.

[FPG03] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance*, pages 23–32, Amsterdam, Netherlands, September 2003. IEEE Computer Society Press.

[Jac12] Paul Jaccard. The distribution of the flora in the alpine zone. *New Phytologist*, 11(2):37–50, February 1912.

[Jar89] M. A. Jaro. Advances in record linking methodology as applied to the 1985 census of tampa florida. *Journal of the American Statistical Society*, 64:1183–1210, 1989.

[KKI02] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, 2002.

[LC06] Creighton University Health Sciences Library and Learning Resources Center. Measuring search effectiveness. http://www.hsl.creighton.edu/hsl/Searching/Recall-Precision.html, October 2006.

[Leh80] Meir M. Lehman. Programs, life cycles, and laws of software evolution. In *Proceedings of the IEEE*, volume 68, pages 1060–1076, September 1980.

[ME96] Alvaro E. Monge and Charles Elkan. The field matching problem: Algorithms and applications. In *Knowledge Discovery and Data Mining*, pages 267–270, 1996.

[SBPK06] Tobias Sager, Abraham Bernstein, Martin Pinzger, and Christoph Kiefer. Detecting similar java classes using tree algorithms. In *MSR '06: Proceedings of the 2006 International Workshop on Mining Software Repositories*, New York, NY, USA, May 2006. ACM Press.

[Val02] G. Valiente. *Algorithms on trees and graphs*. Springer-Verlag, Berlin, 2002.

[WG98] Johannes Weidl and Harald C. Gall. Binding object models to source code: An approach to object-oriented re-architecting. In *Proc. Computer Software and Applications Conf.*, pages 26–31, Vienna, Austria, August 1998. IEEE Computer Society Press.

[XS05] Zhenchang Xing and Eleni Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 54–65, New York, NY, USA, 2005. ACM Press.

[ZWDZ04] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society.