STRUCTURAL COMPARISON OF SOURCE CODE BETWEEN MULTIPLE PROGRAMMING LANGUAGES

by

Rolf Biehn

B.S.C, The University of Guelph, 2003

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in
THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

April 2014

© Rolf Biehn, 2014
Abstract

Software developers are often faced with the task of comparing two or more versions of software. Typical usages of software comparison utilities include: a code-review prior to check-in, tracking down a recently introduced regression, and searching for code-clones in the source code (for future refactoring). However, most traditional source code comparison tools typically use simple text-to-text comparison (with some simple rule-based comparisons for comments), which has the drawback of showing superfluous differences during comparison.

Many projects, for a variety of business reasons, ship products and software development kits (SDKs) using multiple programming languages. It is desirable to compare amongst languages in order to detect potential errors and understand the meaningful differences between the two codebases. In some cases, fixes may be implemented in one language, but not in the other.

In this paper, we create a tool called the Software Difference Analyzer Tool (SDAT), a tool capable of comparing Java and CSharp code, to address some of the unique problems associated with cross-language comparison. Automated testing demonstrated SDAT reduces the number of reported differences by up to 40%. User testing has shown a 37% increase in speed and 28% increase in accuracy.
Preface

This dissertation is an original intellectual product of the author, Rolf Biehn. The research conducted as part of section 4.2 has been approved by the University of British Columbia Behavioural Research Ethics Board (ref: H13-02739)
Table of Contents

Abstract.......................................................................................................................................... ii
Preface........................................................................................................................................... iii
Table of Contents......................................................................................................................... iv
List of Tables ............................................................................................................................... vii
List of Figures ............................................................................................................................. viii
Acknowledgements .................................................................................................................... x
Dedication ..................................................................................................................................... xi

Chapter 1: Introduction ................................................................................................................1
  1.1 Related Work .................................................................................................................. 3
  1.2 Ontological Matching ..................................................................................................... 4
  1.3 ASMOV Algorithm ........................................................................................................ 4
  1.4 Visualizations .................................................................................................................. 5
  1.5 Contributions ................................................................................................................... 7

Chapter 2: Algorithm ....................................................................................................................8
  2.1 Definitions....................................................................................................................... 9
  2.2 AST Generation .............................................................................................................. 10
    2.2.1 Parser ....................................................................................................................... 10
    2.2.2 Common Abstract Syntax Tree Format .................................................................... 11
  2.3 Comparison ................................................................................................................... 12
    2.3.1 Symbol Matching ..................................................................................................... 13
      2.3.1.1 Lexical Matcher ................................................................................................... 15
2.3.1.2 Structural Matcher ............................................................................................ 16
2.3.1.3 Extensional Matcher .......................................................................................... 16
  2.3.1.3.1 Path Matching .......................................................................................... 17
  2.3.1.3.2 Statement Kind Similarity ....................................................................... 22
2.3.2 Code Block Comparison ........................................................................................... 24
  2.3.2.1 Variable Matching ....................................................................................... 24
  2.3.2.2 Node Stream Generation (Matchable Nodes) ................................................. 24
  2.3.2.3 Node Stream Comparison ........................................................................... 27
    2.3.2.3.1 Standard Matchable Node Comparison .............................................. 27
    2.3.2.3.2 Code Symbolic Matching ..................................................................... 27
    2.3.2.3.3 Second Round of Diff Algorithm ....................................................... 30
2.4 Rendering ...................................................................................................................... 31
  2.4.1 Formatting ........................................................................................................... 32

Chapter 3: Implementation .................................................................................................34
3.1 AST Generation ............................................................................................................ 34
  3.1.1 Parser ............................................................................................................... 34
  3.1.2 Common Abstract Syntax Tree Format Considerations ....................................... 35
3.2 Comparison ................................................................................................................ 35
  3.2.1 Diff Algorithm .................................................................................................. 35
3.3 Visualization ............................................................................................................... 36
  3.3.1 Mini-map Scroller ............................................................................................ 36
  3.3.2 The Pop-up Viewer .......................................................................................... 37

Chapter 4: Evaluation .............................................................................................................38
4.1 Automated Results ................................................................. 38
  4.1.1 File Matching ................................................................. 38
  4.1.2 Comparison ................................................................. 39
4.2 User Testing Results ............................................................ 41
  4.2.1 User Feedback .............................................................. 51
4.3 Threats to Validity .............................................................. 52

Chapter 5: Conclusion ............................................................... 53

Bibliography .................................................................................. 54

Appendices .................................................................................... 56

  Appendix A ................................................................................. 56
    A.1 Structural Differences and Corresponding SDAT Support .......... 56
    A.2 Parameters Used in the Implementation .................................. 58

  Appendix B ................................................................................. 59
    B.1 Permission Forms .............................................................. 59
List of Tables

Table 1 Path Matching Related Definitions .................................................................................. 17
Table 2 File Matching Results ...................................................................................................... 39
Table 3 SDAT Coverage ............................................................................................................... 40
Table 4 Automation Results .......................................................................................................... 40
Table 5 File Length Information for the Files Used in User Testing ........................................... 42
Table 6 Number of Errors per File Including and Excluding "Code Smells" (i.e., Validation,
Asserts, etc.) ................................................................................................................................ 43
Table 7 Average Time .................................................................................................................. 45
Table 8 Group Averages of Time Compared to Overall Averages .............................................. 46
Table 9 Accuracy per File ............................................................................................................. 50
Table 10 Group Averages of FMeasures Compared to Overall Averages ..................................... 51
Table 11 Some of the Structural Difference between Java and CSharp and Corresponding SDAT
support ........................................................................................................................................... 56
List of Figures

Figure 1 Definition of Weighted Calculator ................................................................. 5
Figure 2 Screen Shot of SDAT ....................................................................................... 6
Figure 3 Algorithm Overview ....................................................................................... 9
Figure 4 Examples of Preamble and Postamble ........................................................... 11
Figure 5 Simplified CAST Hierarchy ......................................................................... 12
Figure 6 Comparison Process Overview ................................................................. 13
Figure 7 Lexical, Structural and Extensional Scores are Combined into One Score ........ 14
Figure 8 Mapping Pair Selection from a Combined Score ........................................... 15
Figure 9 Sample Code ............................................................................................... 18
Figure 10 CAST Model for Figure 9 “Sample Code” ............................................... 18
Figure 11 Example of Path Generation for the “return true” Statement of Figure 9 Sample Code ................................................................................................. 19
Figure 12 Calculation of Statement Type Score Example ........................................... 23
Figure 13 Code Snippet (top left), CAST Model (top right) and Corresponding Node Stream Generated from the CAST Model (bottom) ......................................................... 26
Figure 14 Resolvable Symbolic Matching ................................................................... 28
Figure 15 Unresolvable Symbolic Matching (Inconsistent) ......................................... 29
Figure 16 Unresolvable Symbolic Matching (Alias) ...................................................... 29
Figure 17 Example of the Second Round of Diff Algorithm ....................................... 31
Figure 18 Code Before Alignment ............................................................................. 33
Figure 19 Code After Alignment .................................................................................................. 33
Figure 20 Basic Mini-map Scroller .............................................................................................. 36
Figure 21 A Screen-Shot of the Pop-up Viewer ......................................................................... 37
Figure 22 Total Time per File per User ...................................................................................... 44
Figure 23 The BaseColor Error .................................................................................................. 47
Figure 24 FMeasure Including "Code Smell" Errors .................................................................. 48
Figure 25 FMeasure Excluding "Code Smell" Errors .................................................................. 49
Acknowledgements

I would like to thank John Bridal for his advice and assistance. I would also like to thank my supervisor, Dr. Eric Whohlshadter, for his guidance, patience, wisdom and help in creating this thesis.
Dedication

Dedicated to my mother, Mari Biehn, for instilling a lifelong love of learning and my wonderful wife, Yuriko Biehn, who has been my support and encouragement.
Chapter 1: Introduction

Software developers are often faced with the task of comparing two or more versions of software. Typical usages of software comparison utilities include: a code-review prior to check-in, tracking down a recently introduced regression, and searching for code-clones in the source code (for future refactoring). However, most traditional source code comparison tools typically use simple text-to-text comparison (with some simple rule-based comparisons for comments), which has the drawback of showing superfluous differences during comparison.

Many projects, for a variety of business reasons, ship products and software development kits (SDKs) using multiple programing languages. Examples of these programming languages may include CSharp, Java, [1], [2], [3], [4]. The author of this study has experience working at a large software company with both Java and CSharp projects.

A typical use case for cross-language comparison is an organization that has multiple version of a project written completely in Java and completely in CSharp. The organization wishes to keep the two codebases as similar as possible to reduce the amount of testing and maintenance required. It is desirable for a developer to compare the two projects in order to ensure that a fix has correctly been checked into both projects. In other cases, one project may work correctly for one work-flow but the second project does not work correctly. A developer may wish to compare the two projects to gain insights into the difference and resolve the issue.

This paper focuses on describing a tool we have created to address some of the unique problems associated with cross-language comparison.

We propose creating a tool that uses the structure of the code as part of the comparison.
This tool offers many advantages over traditional comparison utilities. Some of the advantages include:

- Reduced noise, for example, renamed variables or reordered methods can be safely ignored during comparison;
- Cross-language comparisons become possible;
- Better visualizations, such as syntax highlighting, special visualizations for different types of statements and whitespace, an over-view of precisely which files have changed structurally, etc.; and
- Better user interaction, such as searching for a method by name, “jumping” to another method from one method call statement, filtering unwanted information, etc.

This paper introduces the implementation and evaluation of the *Structural Difference Analysis Tool* (or SDAT for short). This new software program improves the efficiency and effectiveness of comparing Java and CSharp files. The user specifies one CSharp file and one Java file and then the files are compared. The tool borrows aspects from the ASMOV algorithm (Automated Semantic Matching of Ontologies with Verification) [13] and the standard diff algorithm for comparison. SDAT also visualizes the structural differences between the two files.

Even though comparing several different types of languages may be required for a fully functioning software comparison tool, we chose to limit the scope to Java and CSharp comparison. In the current revision of the SDAT tool, we have limited the scope of the SDAT project by focusing on the differences that can be found by comparing the information contained within the two files.

Our assumption is that the code structure does not significantly change between the two languages, only the syntax changes. The approach described in this paper may not perform well
for languages which are radically structurally different. A structural difference between one of
the files may indicate the presence of a bug or an area requiring further investigation.

Our hypothesis is that this tool will be more effective in detecting potential errors and
differences between two different languages in terms of accuracy and speed than industry
standard source code comparison tools.

Chapter 1 of this paper will outline related work and background information. Chapter 2
describes the algorithm and Chapter 3 provides a discussion related to the implementation of the
algorithm. We discuss the evaluation results in Chapter 4, and the conclusion in Chapter 5.

1.1 Related Work

Numerous source comparison programs exist such as Beyond Compare [6], KDiff [7],
and WinMerge [8]. However, the comparison algorithms for these programs will usually only
consider simple heuristics such as ignoring comments and ignoring white space. They generally
do not use an abstract syntax tree (AST) to compare the source files. These programs typically
rely on the diff algorithm. The longest common sequence of lines is determined and the lines in
between the matched lines are typically considered lines that are missing. Several programs for
XML comparison exist such as HTM LDiff [9], Araxis Merge [10], and Guiffy [11]. However,
these programs are not aware of the source code structure and therefore may indicate differences
in code when there is not a semantic difference between two source files (such as a variable
rename or code comments).

Manzoor, Johnson and Nguyen [12] explored using structural comparison as part of
source code management. A case study demonstrated that this approach increases the amount of
conflicts that can be resolved automatically and reduces the number of merge errors However
they did not explore the issues related to cross-language comparison which requires a different algorithm. Cross-language comparisons require techniques to resolve differences in language syntax, structure and libraries. The SDAT tool described in this thesis addresses this cross-language comparison issue.

1.2 Ontological Matching

Source code comparison can be thought of as ontological matching relying on the structure of the abstract syntax tree.

An Ontology, O, consists of “a set of entities that are related by a number of relations” [12] A class is a representation of a concept within the ontology. This could be the concept of a class definition object or a method object of an abstract syntax tree. A literal is a concrete data value such as a number or string. Datatype defines the types of values a literal is allowed to have (e.g., “Number” or “Date”). Properties are the relations entities can have between each other. Properties and datatypes are defined by the grammar of the source code languages.

Ontological matching is the comparison of two ontologies. Since we can consider source code matching to be matching of an ontology, we can leverage existing ontological matching algorithms.

1.3 ASMOV Algorithm

Automated Semantic Matching of Ontologies with Verification [12] is an ontological matching algorithm. ASMOV uses lexical and structural characteristics to calculate similarity values and then verifies these results semantically.
We chose ASMOV because it is easy to extend and it has been proven to perform well. It was among the top matchers for most categories in the Ontology Alignment Evaluation Initiative 2008 [8], a competition that tested bibliographic references from the ontological matchers from more than 15 universities.

The ASMOV algorithm combines several aspects of lexical, element-level, and structural-level measures of similarities to calculate a score between entities by using a weighted calculator. The entity matches are chosen greedily (the ones with the highest score) and verified semantically.

**Weighted Calculator**

Let $F_i$ be a feature score of a certain match from $[0..1]$  
Let $W_i$ be a non-negative weight for $F_i$  
Then, for all $i$,  
$$\text{MatchScore} = \frac{F_i \cdot W_i}{\sum W_i}$$

**Figure 1 Definition of Weighted Calculator**

**1.4 Visualizations**

Once a code comparison is made, the results must be presented to the user in a clear, understandable manner. We call this *visualization*. Source code visualization related papers [14], [15], [16], and [17] explored issues regarding tree comparison visualizations. Several programs are available for visualizing source code such as CodeFlower [18] (which uses a radial gliff system) and aiCall (which uses flow-charts). Both Voinea et al. [8] and Jones et al. [9] used pixel maps to indicate areas of interest in source code files. However, one pixel line cannot
represent more than one document line and therefore multiple or large display maps are required in order to represent large sets of data. Appert and Fekete [10] introduced the idea of orthogonal zooming. We have extended this idea towards pixel-maps for software comparison. This component is called the *mini-map scroller* and is described in more detail in section 3.3.1.

![Figure 2 Screen Shot of SDAT](image)

The program differences are visualized side-by-side with red indicating lines of code which appear similar and blue indicating parts of code that are missing. The location of the mini-map scroller is also indicated.
1.5 Contributions

This paper outlines a flexible algorithm for comparing Java and CSharp (with possible parameter values), describes a generic object abstract syntax model & techniques for visualizing software comparison in general, and discusses some of the issues related to cross-language comparison (specifically Java to CSharp comparison). The implementation is validated using both automated and manual testing. The algorithm/implementation can be further enhanced in order to support future software comparisons.
Chapter 2: Algorithm

The fundamentals of the algorithm for this project can be divided into four parts: parser, comparison, renderer and visualization. The Parser stage consists of abstract syntax tree generation, with the result stored into a unified syntax tree (described in full in 2.2.2). Unified syntax trees from each of the source files are compared and the results are stored in memory. Comparison is the analysis phase. If the comparison phase considers two nodes to be a match, they are called Node Pairs. Rendering is the process of transforming the unified syntax tree back into source code (also known as Pretty Printing). This process includes calculating marker text positions to indicate differences amongst the sources. The visualization process displays the differences and permits user interaction, such as finding differences, highlighting the differences and finding text. The visualization process is discussed in more detail in section 3.3 of the implementation chapter.
We explored using existing comparison algorithms, but we were unable to find algorithms for cross-language comparison. Creating a new algorithm gives us the greatest degree of control over the parameters and implementation of the algorithm.

2.1 Definitions

The following definitions are used throughout the chapter. The occurrences will be marked in underline.

<table>
<thead>
<tr>
<th>Word</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array Declaration Syntax</td>
<td>The syntax for declaring arrays. i.e &quot;int x[] = 5&quot; or &quot;int[] x = 5&quot;</td>
</tr>
<tr>
<td>BlockNode</td>
<td>The syntax node that represents a block e.g. &quot;{ … }&quot;</td>
</tr>
<tr>
<td>ClassNode</td>
<td>The syntax node that represents a class</td>
</tr>
<tr>
<td>ClassType</td>
<td>Meta information about this node</td>
</tr>
<tr>
<td>Code Block</td>
<td>The contents of a block node</td>
</tr>
<tr>
<td>Code Block Comparison</td>
<td>Once a method (and its symbols) have been compared, this comparison will compare the contents of the block nodes</td>
</tr>
<tr>
<td>Common Abstract Syntax Tree Format(CAST)</td>
<td>A unified Abstract Syntax Tree capable of representing both Java and CSharp nodes.</td>
</tr>
<tr>
<td>FileNode</td>
<td>The Syntax Node that represents a file</td>
</tr>
<tr>
<td>Exact Node/ Match</td>
<td>A node or match which is considered to be exactly the same for all effective purposes.</td>
</tr>
<tr>
<td>Indexers</td>
<td>A CSharp concept that allows users to access objects using parameters. e.g. &quot;myCollection[index] = newValue;&quot;</td>
</tr>
<tr>
<td>Modifiers</td>
<td>Attributes of declaration. E.g. &quot;public&quot;, &quot;private&quot;, &quot;final&quot;, etc.</td>
</tr>
<tr>
<td>Orphan Node/Match</td>
<td>A node or match which has no correspond similar or exact match.</td>
</tr>
<tr>
<td>Postamble</td>
<td>Comments and white space that is semantically associated with a statement occurring after a statement.</td>
</tr>
<tr>
<td>Preamble</td>
<td>Comments and white space that is semantically associated with a statement occurring before a statement.</td>
</tr>
<tr>
<td>Similar Node/Match</td>
<td>A node or match which is determined to be semantically similar</td>
</tr>
<tr>
<td>Static Blocks</td>
<td>Blocks of static code (JAVA only)</td>
</tr>
<tr>
<td>Word</td>
<td>Definition</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Structural Difference Analysis Tool(SDAT)</td>
<td>The implementation tool of this thesis, capable of comparing JAVA and CSharp files.</td>
</tr>
<tr>
<td>Symbol Matching</td>
<td>Comparison involving the symbols (such as variable declarations, functions, inner classes and enums).</td>
</tr>
<tr>
<td>Syntax Node</td>
<td>A node of the abstract syntax tree.</td>
</tr>
<tr>
<td>Textual Span</td>
<td>The start and end of a syntax tree node</td>
</tr>
<tr>
<td>TypeNode</td>
<td>The syntax node that represents a type. Types can be defined natively (such as int, double, bool), as part of the runtime language (String, Double, Char) or user defined (such as a user defined class)</td>
</tr>
<tr>
<td>Variable Declaration</td>
<td>The declaration of a variable. E.g. int x = 5;</td>
</tr>
<tr>
<td>Variable Initializer</td>
<td>The expression used to initialize a variable declaration. int x = &lt;variableInitializer&gt;;</td>
</tr>
</tbody>
</table>

2.2 AST Generation

2.2.1 Parser

In the first stages of the parser algorithm, the source code is transformed into a syntax tree. An abstract syntax tree will give us access to the structural information of a program. The philosophy of the parsing stage is to retain the maximum amount of information possible. In addition to retaining information about the **textual span** of each of the nodes, leading and trailing whitespace and comments are retained, known as **preamble** and **postamble** respectively. The intention of **preamble** and **postamble** is to semantically associate whitespace and comments with a statement. The algorithm for calculating **preamble** and **postamble** is described as follows: any whitespace and comments found after a statement and until a new line character are considered **postamble**; whitespace and comments before and up to the previous statement’s **postamble** is considered the statement’s **preamble**.
Figure 4 Examples of Preamble and Postamble

Preambles are comments and white space that are semantically associated with a statement occurring before a statement. Postambles are comments and white space that are semantically associated with a statement occurring after a statement.

2.2.2 Common Abstract Syntax Tree Format

Although CSharp and Java are grammatically similar in many respects, there are several grammatical differences between the two languages. In addition, the two syntax tree generators often generate distinct syntax trees for syntactically equivalent statements. Some examples of this are: static blocks (Java only), array declaration syntax, properties, and indexers. This necessitates a strategy to resolve this scenario. We created an object model generic enough to describe both languages and called it the Common Abstract Syntax Tree (CAST). Nodes from each language are normalized into a standard representation, generally using the most generic representation possible. Static blocks, a Java only concept, are treated as functions. A CSharp property’s getter and setter functions are also treated as functions. Constructors are also considered functions. Appendix A.1 details further differences between functions in CSharp and Java, and outlines SDAT’s support of these differences.
This class diagram indicates the hierarchy to represent a source code file.

2.3 Comparison

Comparison is performed in a top-down order. The classes are first compared using Symbol Matching, which matches all symbol declarations including method declarations, variable declarations and inner class declarations. Matched methods and nested classes are then compared. If we match two methods, variable declarations between the two matched methods are compared using Symbol Matching and then we perform a second type of comparison called Code Block Comparison to compare the code blocks of the method. The results of this
comparison are stored in a bi-directional hash map-like structure for future use in the algorithm.

2.3.1 Symbol Matching

In the first stage of the matching algorithm, classes are compared at a high level. SDAT compares symbols declared in the class definition such as variable declaration, functions, inner classes and enums. Symbol matching allows us to determine potential node pairs for method nodes without the need for complex comparisons between all of the methods from one program to all of the methods of another program. The basis for this comparison is the ASMOV algorithm. Only nodes of similar type are compared, i.e., member variables are not compared to functions and vice versa. Nodes are compared using lexical, structural and extensional similarity (described in the following sections). The results from the individual matchers are combined into a single score. The node pair with the greatest score is removed from the set if its
score exceeds the threshold. This is repeated until there are no node pairs with scores greater than the threshold. If a node pair comparison’s structural score is 100%, it is considered an exact match, and otherwise, we consider this match to be a similar match.

**Figure 7 Lexical, Structural and Extensional Scores are Combined into One Score**
The mapping pair \{\texttt{main(String args[]), Main(String[] args)}\} is selected because it has the highest maximum score above the threshold. The associated rows and columns are removed from the table. Afterwards, the mapping pair \{\texttt{testProperties()}, PropertiesTest()\} is selected. After this step there are no more mapping pairs with a score greater than the threshold, therefore the algorithm is complete.

The semantic validation phase of ASMOV is not used because only semantically valid matches are considered, therefore this step is unnecessary.

The heuristic threshold and weighting values used are described in Appendix A.2.

### 2.3.1.1 Lexical Matcher

If an identifier is associated with a node, it is used for lexical matching. Examples of nodes with identifiers are classes, variable declarations and function declarations. In the case of a property getter or setter function, the property’s identifier is used. Constructors and static blocks are considered to have a null identifier so the identifier comparison feature can be used.
Unlike ASMOV, associated comments are not considered due to our observation that the comments add computational complexity without a significant increase in matching accuracy.

2.3.1.2 Structural Matcher

Structural Matching considers the intrinsic properties of each of the nodes. Examples of these properties are return type, parameters (both the number of parameters and type of each of the parameters), modifiers, and in the case of variables we also consider variable initializers. A score is given to each of these properties and it is combined using a weighted calculator. Comparison can be done exactly (for example comparing if the value of a constant is the same), or more generally such as type comparison (i.e., float vs. double, double vs. java.lang.Double).

2.3.1.3 Extensional Matcher

Extensional similarity compares relationships amongst other nodes in the same syntax tree. There are two types of extensional matching described, path matching and statement kind similarity.
2.3.1.3.1 Path Matching

We define a path and its related definitions as follow:

Table 1 Path Matching Related Definitions

<table>
<thead>
<tr>
<th>Path Node</th>
<th>A class containing the class type, a key and (optionally) an index. The key and index are usually defined relative to the parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path</td>
<td>A collection of Path Nodes. Since the CAST model consists only of member variables and ordered collections, the path information for a node and all of its ancestors is enough to uniquely identify the node</td>
</tr>
<tr>
<td>Key</td>
<td>A property that identifies the path node relative to its parent. Can be represented by using code reflection to determine the parent node’s fieldname. In the case of a FileNode, the file name is used as the key</td>
</tr>
<tr>
<td>Index</td>
<td>In the case of collections, this identifies the node’s index number</td>
</tr>
<tr>
<td>ClassType</td>
<td>Meta information about this node</td>
</tr>
</tbody>
</table>
public class IfTest_Base
{
    public static boolean isEven(int x)
    {
        assert x >= 0;
        if (x % 2 == 0)
            return true;
        return false;
    }
}

Figure 9 Sample Code

Figure 10 CAST Model for Figure 9 “Sample Code”

(additional attributes have been removed for clarity)
Path matching is defined as the comparison of two paths. Path matching is only done for variable declarations within a method block, (since class declarations will all share the same path). Two paths are compared for similarity and a score is obtained.

The path matching algorithm uses an exponentially decreasing scale and a relevance score to give a locational bias. We subtract an exponentially decreasing score from 1.0, rather than building up from zero, in order to avoid potential floating point arithmetic errors that may arise when there is a sum of increasing scores.

Path matching gives allows our algorithm a bit of locational bias. A symbol node near the top of a method block is more likely to map to a symbol node near the top of the method block instead as opposed to a symbol node near the middle that is heavily nested inside several layers of block nodes.
The algorithm for Path Matching is described below:

1: Let L and R represent left node and right node.
2: Let LPath[] && RPath[] represent the pathnodes of the path
3: Let maxDepth = MaxLength(LPath, RPath) // the maximum depth of both paths
4: Let i = Path Depth of method declaration + 1.
5: Let factor = .5
6: Let returnValue = 1; //value [0..1] 1 being a perfect match
7: Let factorFloor = .00001
8: While(i < maxDepth)
9:   Let pathNodeL & pathNodeR = null.
10:  If (i < LPath.length) pathNodeL = LPath[i]
11:  If (i < RPath.length) pathNodeR = RPath[i]
12:  score = CalculateScore(pathNodeL, pathNodeR)
13:  score = score * factor
14:  returnValue -= score
15:  factor = factor / 2
16:  factor = Max(factor, .00001)
17: return Min(0, score)
Perform variable initialization.

In this loop we iterate through each of the path segments until we reach the end.

We initialize pathNodeL and pathNodeR to be the current segment.

Call a function to calculate the score of the two between the two Path Nodes (explained later). The value will be [0..1] with 0 being a perfect match.

This code scales the score by the current factor and updates the returnValue. We use the floor of this score in order to avoid rounding errors associated with small float values.

Scales the factor in half. This makes segments near the beginning have a greater importance than segments near the end.

Make sure the returnValue is always >= 0 (due to possible floating rounding issues)

1: `CalculateScore(pathNodeL, pathNodeR)`
2:   If (pathNodeL ==null or pathNodeR == null) return 1
3:   If ((pathNodeL.index == null)
4:     If(pathNodeR.index == null) return 0
5:       else return 1
6:   If(pathNodeR.index == null) return 1
7:   diff = Min (PATH_MATCHING_MAX_DIFFERENCE,  
8:     ABS(pathNodeL.index - pathNodeR.index))
9:   return diff / PATH_MATCHING_MAX_DIFFERENCE
This function returns a score for two segments between [0..1], 0 being a perfect match. It will consider the differences between the index attribute.

If either of the Path Nodes is null, then return 1. If the indexes don’t match, then return 1. If the indexes match exactly, return 0.

Calculate the difference between the indexes, capped to. First, we calculate the absolute difference of the indexes (with a maximum value of PATH_MATCHING_MAX_DIFFERENCE because differences after this value probably don’t make much of a difference). Then we calculate the return value as a percentage of the maximum possible difference.

2.3.1.3.2 Statement Kind Similarity

Metadata information about the types of child statements present under a function is also considered. If one function contains a large number of method calls, while the other function does not, it is a good indication that these functions are not a match. In order to do this, we keep track of a count of each type of statement (e.g., for loop count, method call count, expression statement count, etc.). The algorithm for calculating the score is described below.

1: Let $L$ and $R$ represent left node and right node.

2: Let $\text{Statements}(L)$ and $\text{Statements}(R)$ be the set of statement types for $L$ and $R$.

3: Let $\text{LCount}(\text{statement})$ and $\text{RCount}(\text{statement})$ be count of individual statement kind in $L$ and $R$.

4: Let $SLR = \{\text{Statements}(L) \cap \text{Statements}(R)\}$

5: Let $SLonly = \text{Statements}(L) - SLR$
6: Let $SR_{only} = Statements(R) - SLR$

7: Let $W(weight, score)$ be a weighted calculator.

8: For each $StmtType \in SLR$

9: $score = \frac{\text{Min}(\text{LCount}(StmtType), \text{RCount}(StmtType))}{\text{Max}(\text{LCount}(StmtType), \text{RCount}(StmtType))}$

10: $W\cdot \text{add}(\text{Max}(\text{LCount}(StmtType), \text{RCount}(StmtType)), score)$

11: For each $StmtType \in SL_{only}$

12: $W\cdot \text{add}(\text{LCount}(StmtType), 0)$

13: For each $StmtType \in SR_{only}$

14: $W\cdot \text{add}(\text{RCount}(StmtType), 0)$

15: return $W\cdot \text{WeightedScore}()$

Lines 1-7: This code will initialize all of the values.

Lines 8-10: We iterate over all of the intersection of statement types. We give the score as a Percentage of the minimum count, over the maximum count. For example if L has 5 return statements and R has 3 return statements this would evaluate to:

$$\frac{\text{Min}(\text{LCount(ReturnStatement)}, \text{RCount(ReturnStatement)})}{\text{Max}(\text{LCount(ReturnStatement)}, \text{RCount(ReturnStatement)})}$$

$$\frac{\text{Min}(5, 3)}{\text{Max}(5, 3)}$$

$$\frac{3}{5}$$

Figure 12 Calculation of Statement Type Score Example

For the left with 5 return statements and the right with 3 return statements the result is $3/5$. 

23
We then add this score and weight it by the max of the number of statement types.

Lines 11-14: In the remaining loops, we add the statement types that are not contained in the intersections to the weighted calculation.

### 2.3.2 Code Block Comparison

Once functions are matched together, child nodes from the matched functions are then compared. Before performing code block comparison, variable declarations are matched, using the symbol matching algorithm. Variable declarations facilitate the alignment of comparisons.

For code block comparison, the abstract syntax tree is flattened into a stream of statement-like tokens and compared using a diff-like algorithm. The standard diff algorithm has been enhanced to include improved similar matches and conditional symbolic matches (explained further in Section 2.3.2.3.2).

#### 2.3.2.1 Variable Matching

We again use the Symbol Matching Algorithm described in Section 2.3.1, but this time we use the variable declarations of the methods instead of the classes.

#### 2.3.2.2 Node Stream Generation (Matchable Nodes)

Now that the objective has been simplified to comparing a method to another method, we translate CAST model into a form that can be used by an enhanced diff algorithm. The strategy is to flatten the hierarchical CAST structure into a stream of tokens in order to leverage standard diff comparison.
We chose to use a diff algorithm because it is relatively simple to implement and is already being used by other software comparison tools for code comparison. Tree based algorithms were also considered. And while a tree based comparisons do offer a greater deal of flexibility by enabling a greater range of comparisons; for example-- comparing entire sub-trees allows us to more naturally compare a list of cascading if statements to case statements -- they are generally more complicated and computationally more complex given the larger number of potential comparisons necessary. These types of comparisons were not in scope for this project, but if a need arises, techniques are available for handling this case such as segmenting the stream and performing tree comparison for this area.

To address the short-comings of the flattening approach we use unique terminals in the node stream. For example, BlockNodes are split into a “StartBlockNode” (for the open brace) and an “EndBlockNode” (for the closing brace). In general, nodes that typically occur in one line of code are considered “matchable” -- this facilitates side-by-side matching and visualization. The CAST nodes are traversed in pre-order and every time a matchable node is encountered it is written to the stream. When a child node is a statement that is not a block node, pseudo StartBlockNode and EndBlockNode are written to the stream before and after the statement. This allows statements like “if(<condition>) return;” and if(<condition>)
{ return; }” to compare as exactly equal.

Statements and class definitions are always considered matchable nodes. Conditional expressions of if statements, loops statements and case statements are also considered matchable nodes.
Figure 13 Code Snippet (top left), CAST Model (top right) and Corresponding Node Stream Generated from the CAST Model (bottom)

The BlockNode is split into a StartBlockNode and EndBlockNode in the Node Stream.
2.3.2.3 Node Stream Comparison

A standard diff algorithm is used to compare the two node streams. The logic for node-to-node comparison is described in 2.3.2.3.1 and 2.3.2.3.2. In the final stages of the algorithm, unmatched nodes are matched using less stringent comparison standards. This allows us to find similar matches, highlighting possible sources of error to the end user. The output from the diff algorithm is a collection of mappings, which indicate matches and orphan nodes.

2.3.2.3.1 Standard Matchable Node Comparison

Matchable nodes are compared by examining the type of the node and the properties of the node. If the node types do not match, or the properties do not match, the nodes do not match. CSharp Properties and Java setters and getters are handled using custom logic because their structure is different. Special comparison logic is also needed for TypeNodes and identifiers. TypeNodes that are similar, such as float, double and Double, can be configured to return similar matches or exact matches. Identifiers are compared by using Code Symbolic Matching.

2.3.2.3.2 Code Symbolic Matching

Often times a symbol will be named differently amongst the two programs because of naming conventions or different library naming support. (CSharp will often prefix their variables with “m_” and give properties and method names an upper case letter). Since the naming of a variable does not affect the structure of a program, we wish to establish conditions when two variables are considered the same.

There are two cases of code symbolic matching, resolvable and unresolvable.
In resolvable symbolic matching, the original definition of the symbol (be it a method declaration, class declaration or variable declaration) can be located within the file. If either file cannot locate the symbol definition, it is considered unresolvable symbolic matching. If the resolved definitions are consistent with the results from the previous (method/class) symbol mapping algorithm, the result is a match. Otherwise, the result is considered to be a non-match.

In this example, the variable declaration leftA maps to rightA, leftB maps to rightB and leftC maps to rightC. The ‘leftA’ variable from the print statement on the left for line#9 resolves to the declaration at line #6. However, the ‘rightB’ variable (on the right-hand side) from line #10 maps to the declaration on line #7 -- this does not match.

In unresolvable symbolic matching, the symbol cannot be resolved. This will be the case when the symbol is declared somewhere outside of the file. In this situation, the symbol is set to be conditionally equal to another symbol and remembered in a set of conditions.
After the diff algorithm has completed, conditionally equal matches are re-examined. In the set of all conditionally equal matches, a conditionally equal match is said to be inconsistent if one of the symbols is also used in another conditionally equal match. All conditional equal matches involving inconsistent matches are said to be similar matches. All conditional equal matches with consistent matches are considered to be exactly equal. This prevents the case where one symbol from one file maps to multiple symbols of the other.

Figure 15 Unresolvable Symbolic Matching (Inconsistent)

In this example, Token.LBRACKET cannot map to both Token.LEFT_BRACKET and Token.RIGHT_BRACKET, therefore it is shown as a similar match.

One additional consideration is that a symbol may have a fully qualified name (including the namespace) and a shortened name. Therefore, we allow a symbol to have exactly one alias as long as the shortened symbol matches the last segment of the fully qualified name.

Figure 16 Unresolvable Symbolic Matching (Alias)

In this case, the symbols are matched because “sdat.comparison.resources.Token.LEFT_BRACKET” appears to be an alias of “Token.LEFT_BRACKET”.

29
2.3.2.3.3  **Second Round of Diff Algorithm**

After the first round of the diff algorithm we will be left with a collection of longest matched common subsequences. However, unmatched nodes on the left and/or right side may exist in between these matched subsequences. In the case of the remaining nodes existing solely on one side, we mark these nodes as *orphans*. When there are both left and right nodes missing, we wish to distinguish *orphans* and *similar nodes*. For the remainder of the nodes in between each of these blocks, the diff algorithm is performed again, only this time using a less strict conditions. This gives us an opportunity to find nodes that are similar but not exact matches. Two nodes are considered “matched” in this phase if they are the same statement type and in the case of expressions, the same expression type. “Matched” nodes from this stage are considered similar nodes by the rest of the algorithm.
Figure 17 Example of the Second Round of Diff Algorithm

Unmatched nodes from the left hand side and the right hand side are matched together using relaxed matching rules. The result is a similar match.

2.4 Rendering

The process of transforming the CAST nodes back into a human readable form is called rendering. We chose to render as human readable source code, as opposed to a different structure such as a tree, because programmers are most familiar with this representation of their code. It is desirable to perform some formatting and add additional space in order to allow for side by side comparison. Matching functions and variables are re-ordered so they appear side by side.
In a process similar to the code block comparison algorithm described in 2.3.1, the CAST object model is visited in order and certain nodes are transformed into objects capable of rendering themselves called *renderable nodes*. Each renderable node has an ability to render in three stages: *pre, contents and post*. Renderable nodes render themselves to a *render stream*, which is a class that keeps track of text and marks ranges of text as being *orphaned* or *similar*. In the pre stage, *preamble* is rendered. In the contents stage, the textual representation of the syntax node is rendered. In the post stage, the *postamble* is rendered. Nodes that cannot be rendered using this framework, for example a *BlockNode* with an open curly brace and a closing curly brace and pre and postamble associated with the braces, are separated into multiple nodes each with pre, contents and post. Each renderable node also has child renderable nodes. By visiting the CAST model in order and rendering both pre and contents before visiting children, and then rendering post after visiting children, it is possible to exactly reconstruct the original source code.

### 2.4.1 Formatting

Another component, called a *Dual Renderer*, is composed of two render classes and is responsible for improving the rendering of both classes. Matched properties and methods are re-arranged so that they appear side-by-side in order to help present the differences to the user. In addition, statements that appear as one line on one side, but are multi line on the other side, are arranged as well (see Figure 18 Code Before Alignment” and Figure 19 Code After Alignment”).
Two files are compared using different formatting rules, which means the differences do not appear on the same line.

In order to accomplish this, the Dual Render will render the pre and contents of both renders and then ensure both render streams are equalized to the same line by adding empty lines (and adding indent if it is the first child of the parent). Afterwards, post is rendered and then both render streams are equalized to the same line.

In this chapter we provided an overview for the algorithm used in SDAT to compare two files together. The basics of the algorithm are parsing, comparison, and rendering.
Chapter 3: Implementation

The implementation uses a combination of Java and CSharp and is over 33,000 lines of code in total. Comparison limitations are discussed in Appendix A.1.

3.1 AST Generation

3.1.1 Parser

After evaluating several options for Java syntax tree generation, JSourceObjectizer [19] was chosen. The reasoning behind this decision is JSourceObjectizer provides a strongly typed object model, is open source with a permissive license (BSD), and has an active community. JSourceObjectizer is based on the ANTLR technology [3] and supports Java version 5.0. For CSharp syntax generation, Microsoft’s Roslyn Compiler [20] was utilized. We chose this compiler because it is provided by Microsoft (the primary owner of the CSharp language, so it is expected to be accurate); it has an active community; and it provides several tools for debugging issues (such as visualization of Syntax tree). The Roslyn compiler supports version 5.0 of CSharp.

JSourceObjectizer generates an abstract syntax tree while Roslyn generates a concrete syntax tree. Since JSourceObjectizer creates an abstract syntax tree that may not contain all of the information needed in the further steps (such as comments and whitespace), this information was augmented by examining the token stream and adding additional information. Using existing AST generators allows us to focus on the comparison logic.
3.1.2 Common Abstract Syntax Tree Format Considerations

Roslyn is a CSharp application while ANTLR is written entirely in Java. The CAST structure and comparison logic was written in Java.

One problem we must consider is that the CAST model exists in Java and the CSharp AST exists in CSharp, therefore we must find a way to transform the CSharp AST into the Java CAST byte code. We address this problem by using the IKVM.NET Byte Code Compiler [21], which is an open source utility which allows .NET code to interoperate with java byte code. The CSharp code uses a series of builders to translate the Roslyn Object model into CAST objects using the IKVM interop and then this object model can be serialized and de-serialized using standard Java serialization. This process saved time by eliminating the need to create custom serialization and deserialization logic between the two languages.

Reflection is used to generate code for the visitor pattern and comparison.

3.2 Comparison

3.2.1 Diff Algorithm

Part of the diff algorithm is based on “A generic, Reusable Diff Algorithm in CSharp - II” [22], which resembles the standard Unix diff algorithm. The code was ported from CSharp to Java.
3.3 Visualization

3.3.1 Mini-map Scroller

The Mini-map scroller is designed to work with any document, using a “lines and markers” concept. Markers are user objects consisting of a colour and a line number. It is recommended, but not required, that these colours match the same colours used in the document.
3.3.2 The Pop-up Viewer

![Image of Pop-up Viewer](image)

Figure 21 A Screen-Shot of the Pop-up Viewer

The Pop-up Viewer box is accessed by clicking the View Pop-up button (pictured in Figure 21 “A Screen-Shot of the The Pop-up Viewer”). This pop-up box will remain visible until the user clicks on the pop-up button again. The pop-up view provides a simple visualization of the user’s current view via the sliders. The Anchor Line is the first line displayed in the Mini-Map. The map length refers to how many lines the Mini-Map should show at a time and the current line refers to the top line of the document view. All of these values can be inputted directly into the text boxes or via the sliders. We changed the slider UI to respond to a single click gesture (i.e., click on the knob, move the mouse left to right, and then click again to commit the value). We hope that this new gesture makes it easier for the user to select certain areas because no drag clicking is required. In contrast, drag clicking makes it more difficult to move the mouse due to the additional muscle control required. The anchor line will update automatically if the Mini-map determines that it cannot fit the current line into the current map view. The layout of the pop-up widget is east to west. Another option we considered was laying out the sliders north to south. This may offer a more intuitive user experience, but causes a much larger number of lines to be obscured by this popup. When weighing user intuitiveness against effectiveness in the comparison, we found the north-south layout was not optimal.
Chapter 4: Evaluation

The program was evaluated using two evaluation methods. In the first method, two open source projects were evaluated using automated results, comparing the SDAT against a standard text diff utility, in terms of the number of lines of difference. In the second method, the program was evaluated by user testing by using a small subset of these source files.

The projects used for both evaluation methods are ANTLR and IText. ANTLR is an open source parsing engine available in both CSharp and Java. IText [2] is a PDF manipulation library also available in both CSharp and Java.

4.1 Automated Results

We performed automated testing to demonstrate the completeness and breadth of the SDAT tool. The correctness of the automation was verified by a series of sanity checks. AST nodes that cannot be handled correctly by default throw exceptions. Each AST is transformed into the CAST model and rendered into text. If the generated text is not the same as the original text, an exception is thrown. We also performed manual spot checks for several of the files.

First, files from each of the projects were compared and matched. Then the number of differences reported by SDAT and a baseline comparison tool were compared.

4.1.1 File Matching

A file name matching heuristic was used to perform the file matching. The results from file matching are shown in (Table 2 File Matching Results). Most of the unmatched files in ANTLR_Java are auto generated and/or extra testing related files. The reason for the large
number of unmatched files missing in ITEXT_CS is because the IText CS implementation uses a custom built encryption library, while the Java implementation uses a third party library.

Table 2 File Matching Results

Results show a non-trivial overlap of files between the two projects

<table>
<thead>
<tr>
<th>Project Type</th>
<th>Unmatched Files</th>
<th>Matched Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANTLR_Java</td>
<td>270</td>
<td>107</td>
</tr>
<tr>
<td>ANTLR_CS</td>
<td>48</td>
<td>107</td>
</tr>
<tr>
<td>ITEXT_Java</td>
<td>67</td>
<td>568</td>
</tr>
<tr>
<td>ITEXT_CS</td>
<td>1326</td>
<td>568</td>
</tr>
</tbody>
</table>

4.1.2 Comparison

Beyond Compare 3 [6] is a popular text based source code file comparison utility. It is the top ranking software comparison tool for CNET (http://download.cnet.com/) with nearly 400,000 downloads and is an industry standard for source code comparison. Beyond Compare 3 will be used as a base line for the number of differences. Import statements are ignored by both Beyond Compare 3 and SDAT as they are not considered structurally different. (In SDAT, import statements are not compared, whereas Beyond Compare 3 has a text pre-processor that removes the import statements). Beyond Compare 3 has been configured to “Ignore unimportant differences”, such as comments and whitespace.

Table 3 SDAT Coverage shows that SDAT was not able to compare all of the matched files, but it was able to cover more than 90% of the projects and almost 85% of the total lines of code. Most of the files not covered by SDAT are because of unsupported features (such as sparse matrix initialization) and bugs in the AST generation (text offsets are being reported incorrectly by the AST which fail sanity checks).
Table 3 SDAT Coverage

The following table shows most of the files/code in both projects can be compared by using SDAT.

<table>
<thead>
<tr>
<th>Project Type</th>
<th>Type</th>
<th>Files</th>
<th>Whitespace(lines)</th>
<th>Code(lines)</th>
<th>Total(lines)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANTLR(Java &amp; CSharp)</td>
<td>Matched</td>
<td>214</td>
<td>21551</td>
<td>29597</td>
<td>51148</td>
</tr>
<tr>
<td>ANTLR(Java &amp; CSharp)</td>
<td>Comparable</td>
<td>196</td>
<td>19052</td>
<td>25029</td>
<td>44081</td>
</tr>
<tr>
<td>%Diff</td>
<td></td>
<td>91.59%</td>
<td>88.40%</td>
<td>84.57%</td>
<td>86.18%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Project Type</th>
<th>Type</th>
<th>Files</th>
<th>Whitespace(lines)</th>
<th>Code(lines)</th>
<th>Total(lines)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANTLR(Java &amp; CSharp)</td>
<td>Matched</td>
<td>1136</td>
<td>157363</td>
<td>194685</td>
<td>352048</td>
</tr>
<tr>
<td>ANTLR(Java &amp; CSharp)</td>
<td>Comparable</td>
<td>1113</td>
<td>152850</td>
<td>188084</td>
<td>340934</td>
</tr>
<tr>
<td>%Diff</td>
<td></td>
<td>97.98%</td>
<td>97.13%</td>
<td>96.61%</td>
<td>96.84%</td>
</tr>
</tbody>
</table>

The number of diff lines reported as different by Beyond Compare 3 and the number of diff lines reported by the SDAT are shown in the table below. The results show that the number of differences SDAT reports is around 40% of those reported by Beyond Compare. This is a significant number. The reason for more CSharp orphans in ANTLR is because ANTLR uses conditional compiles for different frameworks, and also ANTRL CSharp formatting uses more newlines than the ANTLR Java formatting which inflates the number of lines for orphans.

Table 4 Automation Results

This table shows that SDAT significantly reduces the number of differences reported compared to Beyond Compare.

<table>
<thead>
<tr>
<th>Project Type</th>
<th>Type</th>
<th>CS Orphans</th>
<th>Java Orphans</th>
<th>Similar</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANTLR(Java &amp; CSharp)</td>
<td>Beyond Compare</td>
<td>4640</td>
<td>641</td>
<td>4509</td>
<td>9790</td>
</tr>
<tr>
<td>ANTLR(Java &amp; CSharp)</td>
<td>SDAT</td>
<td>2463</td>
<td>815</td>
<td>363</td>
<td>3641</td>
</tr>
<tr>
<td>%Diff</td>
<td></td>
<td>53.08%</td>
<td>127.15%</td>
<td>8.05%</td>
<td>37.19%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Project Type</th>
<th>Type</th>
<th>CS Orphans</th>
<th>Java Orphans</th>
<th>Similar</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANTLR(Java &amp; CSharp)</td>
<td>Beyond Compare</td>
<td>7759</td>
<td>7503</td>
<td>51885</td>
<td>67147</td>
</tr>
<tr>
<td>ANTLR(Java &amp; CSharp)</td>
<td>SDAT</td>
<td>6580</td>
<td>7458</td>
<td>19814</td>
<td>27272</td>
</tr>
<tr>
<td>%Diff</td>
<td></td>
<td>84.80%</td>
<td>99.40%</td>
<td>38.19%</td>
<td>40.62%</td>
</tr>
</tbody>
</table>
Clearly simply reducing the number of errors reported does not validate the ability of this tool to reduce the amount of time spent nor accuracy of comparing two cross-language source files, which is why user testing was performed to validate the project. However, automated results demonstrate a breadth of coverage and a potential for a great reduction in the number of differences an end user must focus on.

4.2 User Testing Results

We chose six computer programmers (4 grad students and 2 professional software developers) to participate in the user verification of SDAT. Participants were asked to simulate a code review looking for errors in one of the files. Participants were instructed to report any differences that “seem suspicious” or warrant further investigation (we will refer to these as error predictions). Proof of error was not required to “seem suspicious”. Participants were also instructed to focus on the differences that the tool is reporting, rather than searching for errors themselves. Participants were given some time to become familiar with each of the tools before using them.

Three files were chosen from the ANTLR project and three files were chosen from the IText project. File length ranged in size from 200 to 850 lines, each project having files of small, medium and large size. This information is detailed in Table 5. The order of file evaluation remained constant throughout the experiment. The participants were divided into two groups. Group one evaluated the first three files using SDAT and the remaining three files using Beyond Compare. Group two evaluated the first three files using Beyond Compare and the remaining
files using SDAT. Several errors were introduced into the code in order to create more testing points.

Table 5  File Length Information for the Files Used in User Testing

A variety of files of different sizes and complexity were chosen to showcase typical use cases.

<table>
<thead>
<tr>
<th>File Name</th>
<th>Project</th>
<th>Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATN.cs</td>
<td>ANTLR</td>
<td>325</td>
</tr>
<tr>
<td>ATN.java</td>
<td>ANTLR</td>
<td>213</td>
</tr>
<tr>
<td>BaseColor.cs</td>
<td>ITEXT</td>
<td>199</td>
</tr>
<tr>
<td>BaseColor.java</td>
<td>ITEXT</td>
<td>215</td>
</tr>
<tr>
<td>CommonToken.cs</td>
<td>ANTLR</td>
<td>292</td>
</tr>
<tr>
<td>CommonToken.java</td>
<td>ANTLR</td>
<td>228</td>
</tr>
<tr>
<td>Document.cs</td>
<td>ITEXT</td>
<td>735</td>
</tr>
<tr>
<td>Document.java</td>
<td>ITEXT</td>
<td>854</td>
</tr>
<tr>
<td>DocWriter.cs</td>
<td>ITEXT</td>
<td>373</td>
</tr>
<tr>
<td>DocWriter.java</td>
<td>ITEXT</td>
<td>434</td>
</tr>
<tr>
<td>Lexer.cs</td>
<td>ANTLR</td>
<td>610</td>
</tr>
<tr>
<td>Lexer.java</td>
<td>ANTLR</td>
<td>430</td>
</tr>
</tbody>
</table>

Some of the reported differences included common categories of differences. For example, in Java, some of the source files throw an IOException that is not thrown in CSharp. In this case, we treat all of these occurrences as one error prediction.

While some error predictions are logically provable as “warrants further investigation” (for example a missing case statement or an index off by one error), some error predictions are more difficult to classify. We will call these error predictions “Code Smells” as they may indicate an error but are not necessarily indicative of a logical error in the program. Examples of
code smell errors that fall into this category are: exception handling, validation (such as asserts), hashcode generation, logging, passing in null instead of a default locale for Locale, constants defined in other files (but with the same name and context) and caching/buffering. Table 6 summarizes the results, both including and excluding code smell errors.

Table 6 Number of Errors per File Including and Excluding "Code Smells" (i.e., Validation, Asserts, etc.).

These numbers include introduced errors and possible existing errors.

<table>
<thead>
<tr>
<th>File</th>
<th>Errors (Including Code Smells)</th>
<th>Errors (Excluding Code Smells)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATN</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>BaseColor</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>CommonToken</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Document</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>DocWriter</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Lexer</td>
<td>8</td>
<td>5</td>
</tr>
</tbody>
</table>

It is not possible to establish statistical significance due to the limited number of trials, users and test cases, however, the testing results may indicate certain patterns and trends. A statistically significant user study is left as future work. The following charts and tables demonstrate the timing results from user testing. It clearly shows that in all cases SDAT requires less time than Beyond Compare 3. On average, the testers were 37% faster when using SDAT versus Beyond Compare 3. In all cases, the average time for SDAT was less than beyond compare and it most cases the time spent on a per user basis is also less. We have also observed
that the group that uses SDAT has a lower average time than the group that uses Beyond Compare.

![Figure 22 Total Time per File per User](image)

Each bar represents a user’s individual time. The blue bars represent the results for Beyond Compare, while the red bars represent the results from SDAT. SDAT out preformed Beyond Compare in almost all cases.
Table 7 Average Time

On average, SDAT users completed their tasks 37% faster.

<table>
<thead>
<tr>
<th>File Name</th>
<th>Average Time</th>
<th>SDAT Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATN</td>
<td>6:30</td>
<td></td>
</tr>
<tr>
<td>BC</td>
<td>8:18</td>
<td></td>
</tr>
<tr>
<td>SDAT</td>
<td>4:41</td>
<td>43.58%</td>
</tr>
<tr>
<td>BaseColor</td>
<td>5:35</td>
<td></td>
</tr>
<tr>
<td>BC</td>
<td>5:47</td>
<td></td>
</tr>
<tr>
<td>SDAT</td>
<td>5:24</td>
<td>6.72%</td>
</tr>
<tr>
<td>CommonToken</td>
<td>2:30</td>
<td></td>
</tr>
<tr>
<td>BC</td>
<td>3:10</td>
<td></td>
</tr>
<tr>
<td>SDAT</td>
<td>1:51</td>
<td>41.40%</td>
</tr>
<tr>
<td>Document</td>
<td>7:05</td>
<td></td>
</tr>
<tr>
<td>BC</td>
<td>8:38</td>
<td></td>
</tr>
<tr>
<td>SDAT</td>
<td>5:33</td>
<td>35.69%</td>
</tr>
<tr>
<td>DocWriter</td>
<td>5:08</td>
<td></td>
</tr>
<tr>
<td>BC</td>
<td>6:23</td>
<td></td>
</tr>
<tr>
<td>SDAT</td>
<td>3:53</td>
<td>39.13%</td>
</tr>
<tr>
<td>Lexer</td>
<td>8:33</td>
<td></td>
</tr>
<tr>
<td>BC</td>
<td>12:03</td>
<td></td>
</tr>
<tr>
<td>SDAT</td>
<td>5:02</td>
<td>58.18%</td>
</tr>
<tr>
<td>Grand Total</td>
<td>5:53</td>
<td></td>
</tr>
</tbody>
</table>

Average Improvement 37.48%
**Table 8 Group Averages of Time Compared to Overall Averages**

Results demonstrate the group using SDAT always has an improved overall average time.

<table>
<thead>
<tr>
<th>File Name</th>
<th>Average Time</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATN</td>
<td>6:30</td>
<td></td>
</tr>
<tr>
<td>Group 1 (SDAT)</td>
<td>4:41</td>
<td>(38.79%)</td>
</tr>
<tr>
<td>Group 2 (BC)</td>
<td>8:18</td>
<td>(-43.57%)</td>
</tr>
<tr>
<td>BaseColor</td>
<td>5:35</td>
<td></td>
</tr>
<tr>
<td>Group 1 (BC)</td>
<td>5:47</td>
<td>(38.79%)</td>
</tr>
<tr>
<td>Group 2 (SDAT)</td>
<td>5:24</td>
<td>(3.40%)</td>
</tr>
<tr>
<td>CommonToken</td>
<td>2:30</td>
<td></td>
</tr>
<tr>
<td>Group 1 (BC)</td>
<td>3:10</td>
<td>(-21.05%)</td>
</tr>
<tr>
<td>Group 2 (SDAT)</td>
<td>1:51</td>
<td>(35.14%)</td>
</tr>
<tr>
<td>Document</td>
<td>7:05</td>
<td></td>
</tr>
<tr>
<td>Group 1 (BC)</td>
<td>8:38</td>
<td>(-17.95%)</td>
</tr>
<tr>
<td>Group 2 (SDAT)</td>
<td>5:33</td>
<td>(27.63%)</td>
</tr>
<tr>
<td>DocWriter</td>
<td>5:08</td>
<td></td>
</tr>
<tr>
<td>Group 1 (SDAT)</td>
<td>3:53</td>
<td>(32.19%)</td>
</tr>
<tr>
<td>Group 2 (BC)</td>
<td>6:23</td>
<td>(-19.58%)</td>
</tr>
<tr>
<td>Lexer</td>
<td>8:33</td>
<td></td>
</tr>
<tr>
<td>Group 1 (SDAT)</td>
<td>5:02</td>
<td>(69.87%)</td>
</tr>
<tr>
<td>Group 2 (BC)</td>
<td>12:03</td>
<td>(-29.05%)</td>
</tr>
</tbody>
</table>

We measured accuracy using an FMeasure. On average SDAT was 28% more effective. For nearly all of the files, SDAT was more accurate than Beyond Compare.

The reason for the large discrepancy in accuracy for BaseColor (950%) was likely because there was only one error for BaseColor. Only 1 out of 3 Beyond Compare users correctly predicted the error while 3 out of 3 SDAT users correctly predicted the error. The actual error is shown in Figure 23 “The BaseColor Error”. This may indicate that certain classes of errors are easier to see using SDAT.
Figure 23 The BaseColor Error

The same comparison is shown in both Beyond Compare (above) and SDAT (below). In this cases, color.R is mapped to getBlue(), but clearly this should be getRed() or something similar. Only 1 out of 3 Beyond Compare users correctly predicted the error while 3 out of 3 SDAT users predicted the error.
Figure 24 FMeasure Including "Code Smell" Errors

Each bar represents a user’s FMeasure for a given file. The blue bars represent the results for Beyond Compare, while the red bars represent the results from SDAT. SDAT out preformed Beyond Compare in most cases.
Figure 25 FMeasure Excluding "Code Smell" Errors

Each bar represents a user’s FMeasure for a given file. The blue bars represent the results for Beyond Compare, while the red bars represent the results from SDAT. SDAT outperformed Beyond Compare in most cases.
Table 9 Accuracy per File

The average accuracy of SDAT is better by 28%.

<table>
<thead>
<tr>
<th>File Name</th>
<th>Average of Beyond Compare</th>
<th>Average of SDAT</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATN</td>
<td>0.808838384</td>
<td>0.842592593</td>
<td>4%</td>
</tr>
<tr>
<td>Code Smell Errors Excluded</td>
<td>0.822222222</td>
<td>0.888888889</td>
<td>8%</td>
</tr>
<tr>
<td>Code Smell Errors Included</td>
<td>0.795454545</td>
<td>0.796296296</td>
<td>0%</td>
</tr>
<tr>
<td><strong>BaseColor</strong></td>
<td><strong>0.20952381</strong></td>
<td><strong>0.773809524</strong></td>
<td><strong>269%</strong></td>
</tr>
<tr>
<td>Code Smell Errors Excluded</td>
<td>0.095238095</td>
<td>1</td>
<td>950%</td>
</tr>
<tr>
<td>Code Smell Errors Included</td>
<td>0.323809524</td>
<td>0.547619048</td>
<td>69%</td>
</tr>
<tr>
<td><strong>CommonToken</strong></td>
<td><strong>0.429100529</strong></td>
<td><strong>0.411111111</strong></td>
<td><strong>4%</strong></td>
</tr>
<tr>
<td>Code Smell Errors Excluded</td>
<td>0.428571429</td>
<td>0.6</td>
<td>40%</td>
</tr>
<tr>
<td>Code Smell Errors Included</td>
<td>0.42962963</td>
<td>0.222222222</td>
<td>48%</td>
</tr>
<tr>
<td><strong>Document</strong></td>
<td><strong>0.419444444</strong></td>
<td><strong>0.344444444</strong></td>
<td><strong>18%</strong></td>
</tr>
<tr>
<td>Code Smell Errors Excluded</td>
<td>0.388888889</td>
<td>0.222222222</td>
<td>43%</td>
</tr>
<tr>
<td>Code Smell Errors Included</td>
<td>0.45</td>
<td>0.466666667</td>
<td>4%</td>
</tr>
<tr>
<td><strong>DocWriter</strong></td>
<td><strong>0.71468254</strong></td>
<td><strong>0.805555556</strong></td>
<td><strong>13%</strong></td>
</tr>
<tr>
<td>Code Smell Errors Excluded</td>
<td>0.746031746</td>
<td>0.777777777</td>
<td>4%</td>
</tr>
<tr>
<td>Code Smell Errors Included</td>
<td>0.683333333</td>
<td>0.833333333</td>
<td>22%</td>
</tr>
<tr>
<td><strong>Lexer</strong></td>
<td><strong>0.563321766</strong></td>
<td><strong>0.843945869</strong></td>
<td><strong>50%</strong></td>
</tr>
<tr>
<td>Code Smell Errors Excluded</td>
<td>0.595604396</td>
<td>0.824074074</td>
<td>38%</td>
</tr>
<tr>
<td>Code Smell Errors Included</td>
<td>0.531039136</td>
<td>0.863817664</td>
<td>63%</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>0.524151912</strong></td>
<td><strong>0.670243183</strong></td>
<td><strong>28%</strong></td>
</tr>
</tbody>
</table>
Table 10 Group Averages of FMeasures Compared to Overall Averages

We observe that SDAT has better accuracy for 10 of the 12 FMeasures being observed. Group 1 had a better FMeasure than Group 2 for 8 of the 12 measurements.

<table>
<thead>
<tr>
<th>File Name</th>
<th>Average</th>
<th>Group 1 Difference</th>
<th>Group 2 Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATN</td>
<td>0.825715488</td>
<td>(SDAT)</td>
<td>0.822222222</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(BC)</td>
<td>0.795454545</td>
</tr>
<tr>
<td></td>
<td>Code Smell Errors Excluded</td>
<td>0.888888889</td>
<td>(7.65%)</td>
</tr>
<tr>
<td></td>
<td>Code Smell Errors Included</td>
<td>0.796296296</td>
<td>(-3.56%)</td>
</tr>
<tr>
<td>BaseColor</td>
<td>0.491666667</td>
<td>(BC)</td>
<td>0.795454545</td>
</tr>
<tr>
<td></td>
<td>Code Smell Errors Excluded</td>
<td>0.095238095</td>
<td>(-80.63)</td>
</tr>
<tr>
<td></td>
<td>Code Smell Errors Included</td>
<td>0.323809524</td>
<td>(-34.14%)</td>
</tr>
<tr>
<td>CommonToken</td>
<td>0.42010582</td>
<td>(BC)</td>
<td>0.222222222</td>
</tr>
<tr>
<td></td>
<td>Code Smell Errors Excluded</td>
<td>0.428571429</td>
<td>(2.02%)</td>
</tr>
<tr>
<td></td>
<td>Code Smell Errors Included</td>
<td>0.42962963</td>
<td>(2.27%)</td>
</tr>
<tr>
<td>Document</td>
<td>0.381944444</td>
<td>(BC)</td>
<td>0.547619048</td>
</tr>
<tr>
<td></td>
<td>Code Smell Errors Excluded</td>
<td>0.388888889</td>
<td>(1.82%)</td>
</tr>
<tr>
<td></td>
<td>Code Smell Errors Included</td>
<td>0.45</td>
<td>(17.82%)</td>
</tr>
<tr>
<td>DocWriter</td>
<td>0.760119048</td>
<td>(SDAT)</td>
<td>0.466666667</td>
</tr>
<tr>
<td></td>
<td>Code Smell Errors Excluded</td>
<td>0.777777778</td>
<td>(2.32%)</td>
</tr>
<tr>
<td></td>
<td>Code Smell Errors Included</td>
<td>0.833333333</td>
<td>(9.63%)</td>
</tr>
<tr>
<td>Lexer</td>
<td>0.703633817</td>
<td>(SDAT)</td>
<td>0.531039136</td>
</tr>
<tr>
<td></td>
<td>Code Smell Errors Excluded</td>
<td>0.824074074</td>
<td>(17.12%)</td>
</tr>
<tr>
<td></td>
<td>Code Smell Errors Included</td>
<td>0.863817664</td>
<td>(22.77%)</td>
</tr>
</tbody>
</table>

4.2.1 User Feedback

Feedback was requested from the participants.

The users reported that the anchor system used by the Mini-map scroller was confusing. Similarly, the single click gesture instead of the typical gesture was confusing because it is contrary to expected Operating System interactions. There was little desire to zoom.
in and zoom out while performing tasks. Most users simply scrolled or used the Next Difference button.

5 out of 6 users felt this tool may be helpful to them for performing this task. Users indicated a strong preference to interact with the diff tool using “IDE like navigation” such as going to certain methods showing all referenced variables.

4.3 Threats to Validity

Threats to experimental validity include: a small number of files and users, some of the classifications are subjective, users were not chosen randomly, testers may experience fatigue (some sessions exceeded 40 minutes), and users may perform differently knowing that accuracy and time is being measured.
Chapter 5: Conclusion

Automated testing demonstrated SDAT reduces the amount of reported differences by up to 40%. User testing showed a 37% increase in speed and 28% increase in accuracy, versus an industry standard file comparison utility. We see this as a successful result. SDAT is able to complete the source code comparison both faster and more accurately. The user experience was generally positive, with of 83% of users stating that they “would likely” use this in their school or work projects.

The evaluation results showed that SDAT could be improved. For example, users requested improvements to the Visualization stage, such as navigation by function name, forward & back navigation, an ability to specify matching functions / variables, improved finding logic, and in place editing.

Overall, the SDAT development and implementation was a success. Yet there remains much work to be done in developing an optimal enhanced diff tool. An ability to remember or ignore areas of differences and resolving symbols defined in different classes would increase the usability and accuracy of the tool. Interesting possibilities for further research include further programming languages, bug fixes, performance improvements, comparing more than one file at a time, and using the tool to automatically detect potential errors, without the need for user intervention.
Bibliography


Appendices

Appendix A

A.1 Structural Differences and Corresponding SDAT Support

The following table outlines differences between Java and C# based on information collected [23], and the corresponding SDAT support.

Table 11 Some of the Structural Difference between Java and C# and Corresponding SDAT support

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>CSharp</th>
<th>SDAT support</th>
</tr>
</thead>
<tbody>
<tr>
<td>N/A</td>
<td>Jagged arrays</td>
<td></td>
<td>Supported. CAST uses a C# like model</td>
</tr>
<tr>
<td></td>
<td>int [][]myArray = new int[2][];</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N/A</td>
<td>Multiple inheritance</td>
<td></td>
<td>Supported. CAST uses multiple type names for inheritance</td>
</tr>
<tr>
<td>~ package name</td>
<td>N/A</td>
<td>Name space</td>
<td>Only supported for top level namespaces.</td>
</tr>
<tr>
<td>N/A</td>
<td>Finalizer</td>
<td>~MyClass</td>
<td>Not supported</td>
</tr>
<tr>
<td>Array Declarators</td>
<td></td>
<td>Array Declarator</td>
<td>Supported. SDAT adjusts comparison logic for equivalence</td>
</tr>
<tr>
<td>String []a;</td>
<td></td>
<td>string[] a;</td>
<td></td>
</tr>
<tr>
<td>String a[];</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java</td>
<td>CSharp</td>
<td>SDAT support</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>--------</td>
<td>--------------</td>
<td></td>
</tr>
<tr>
<td>Calling base constructor</td>
<td>Calling base constructor</td>
<td>Supported. SDAT will capture Java super statements and store them as a property of the constructor method. When rendering the code block of the constructor, this property is used.</td>
<td></td>
</tr>
</tbody>
</table>
| Public MyClass(int x) {  
   super(x);  
   ...  
} | public MyClass(int x) : base(x) {...} |

### Generics

| Generics | Generics | Currently not supported by SDAT (ignored). |

### Annotations

| Annotations | Annotations | Not supported. |

### N/A

| Operator overloading | Not supported. |

### ~functions

| setXYZ(String x) {...} | Properties | Supported. Getters and setters are considered functions. |
| String getXYZ() |

### ~function

| setXYZ(int index, int value) | Indexer | Supported. SDAT will perform special symbolic comparisons in this case. |
| getXYZ(int index) |

### N/A

| Preprocessor Directives | Not Supported. Ignored by SDAT |

### N/A

| Using statement | Supported |

### N/A

| Pointers and Unsafe Code | Not Supported |

### N/A

| Pass by Reference | Supported |
| public static void foo(out string param) |

### N/A

| Named parameters | Not Supported |
| foo(x:123) |

### N/A

| Optional Parameters | Not Supported |
| foo(string optional = “optional”) |

### Struct

| Not Supported |

| | |

57
A.2 Parameters Used in the Implementation

The parameters were chosen somewhat arbitrarily and adjusted to fit through experimentation.

Values used for structure comparison:

RETURN_VALUE_WEIGHT = .3
INITIALIZER_WEIGHT = .1
PARAMETER_WEIGHT = .25
MODIFIER_WEIGHT = .05

Combination of Scores:

LEXICAL = .15
STRUCTURAL = .65
EXTENSIONAL = .2

Comparison Threshold = .70

Path Matching

PATH_MATCHING_MAX_DIFFERENCE=9
Appendix B

B.1 Permission Forms

Project Title: A Comparison Between Structural Comparison and Standard Diff Software tools.

Study Team
Principal Investigator: [redacted]
Co-Investigator(s): Rolf Biehn, UBC graduate student. [redacted]
This research is being undertaken as part of a graduate level thesis, the results of which will be made publicly available. No personal data will be shared publicly.

Study Procedures
In this study, you will be asked to evaluate source code comparison using existing source code comparison tools and a new source comparison tool. You will be given a short tutorial on the new source code comparison tool and time to get practise using it. You will be presented with one file in Java and one file in C# and asked to identify potential sources of errors and identify general differences between the two files. You will be asked to participate in this process 6 times using both existing software comparison tools and the new source code comparison tool. Accuracy and the total time to complete these tasks will be measured.

REIMBURSEMENT: $10
TIME COMMITMENT: 30 mins to 1 hour
CONFIDENTIALLY: Your confidentiality will be respected. Information that discloses your identity will not be released without your consent unless required by law”. *(see Behavioural Ethics Guidance Note 8.8.3 for further details on reportable offences). Information will be stored on a secured computer. Subjects will not be identified by name in any reports of the completed study.

Who can you contact if you have questions about the study?
If you have any questions or concerns about what we are asking of you, please contact the study leader or one of the study staff. The names and telephone numbers are listed at the top of the first page of this form.
**Who can you contact if you have complaints or concerns about the study?**
If you have any concerns about your rights as a research subject and/or your experiences while participating in this study, you may contact the Research Subject Information Line in the UBC Office of Research Services at [redacted] or if long distance e-mail [redacted] or call toll free [redacted].

**XII. PARTICIPANT CONSENT AND SIGNATURE PAGE**
Taking part in this study is entirely up to you. You have the right to refuse to participate in this study. If you decide to take part, you may choose to pull out of the study at any time without giving a reason and without any negative impact on your employment, class standing, etc.
- Your signature below indicates that you have received a copy of this consent form for your own records.
- Your signature indicates that you consent to participate in this study.

____________________________________________________
Participant Signature     Date
(or Parent or Guardian Signature)

____________________________________________________
Printed Name of the Participant (or Parent or Guardian) signing above