srcDiff: A Programmer-Centric Syntactic Differencing Tool

Michael J. Decker
Department of Computer Science
Bowling Green State University
Bowling Green, OH, USA
mdecke@bgsu.edu

Ali F. Al-Ramadan
Department of Computer Science
Kent State University
Kent, Ohio, USA
aalramad@kent.edu

Humphrey B. Borketey
Department of Computer Science
Bowling Green State University
Bowling Green, OH, USA
humphbb@bgsu.edu

Andrew M. Symonds
Department of Computer Science
Bowling Green State University
Bowling Green, OH, USA
asymond@bgsu.edu

Michael L. Collard
Department of Computer Science
The University of Akron
Akron, OH, USA
collard@uakron.edu

Jonathan I. Maletic
Department of Computer Science
Kent State University
Kent, Ohio, USA
jmaletic@kent.edu

***Abstract*—srcDiff, a scalable syntactic differencing tool designed to improve the understandability of source code changes is presented. The tool uses the srcML infrastructure, which encodes source code as XML while preserving all original text. srcDiff extends the srcML markup with additional tags to identify changes. A rule-driven differencing algorithm is applied that integrates matching and edit-script generation during traversal of the abstract syntax tree. The rules capture common developer editing patterns and allow srcDiff to depart from minimal edit distance. This results in a much more human-readable delta. Unlike other syntactic differencing approaches, it incorporates rules based on programming language syntax and domain knowledge of code changes. The resulting srcDiff format supports both automated analysis and generation of human-readable views, including side-by-side, unified, HTML, and terminal outputs with customizable themes and syntax highlighting. srcDiff operates on complete or incomplete code and supports multiple programming languages to provide a practical solution for syntax-aware change analysis.**

Demo Video: https://youtu.be/jLVMqQvJh-g

Keywords—syntactic differencing, source code analysis, srcDiff, srcML

I. Introduction

Source code differencing identifies changes between two versions of a program and is fundamental for a multitude of software development tasks (e.g., code review, merge conflict resolution, program comprehension, impact analysis). It also forms the basis for several software engineering research topics (e.g., bug/feature location, refactoring, clone detection, recommender systems, author attribution). Differencing tools broadly fall into three categories: *textual*, *syntactic*, and *semantic*, based on what information is compared.

Textual differencing treats source code files as sequences of text (e.g., lines, words), with the de facto standard being to treat the text as a sequence of lines. These line-based differencers (e.g., GNU diff and Git diff [1]) are efficient, language-agnostic, and widely used in practice. However, they completely disregard language syntax and operate strictly at the line level. A single logical change can be fragmented across multiple non-contiguous edits and cross line boundaries. That is, a single logical edit does not conform to a line. As a result, the delta produced often fails to clearly reflect the meaning of the change. Changing at a different granularity (e.g., word) does not solve the issue, as textual differencers lack understanding of the source code syntax.

Syntactic differencing tools [2–9] address the logical edit problem by comparing Abstract Syntax Trees (ASTs) rather than raw text, and can better report changes at the appropriate granularity. Because ASTs encode the grammatical structure of the language, syntactic-based approaches preserve syntactic boundaries, detect fine-grained changes within a line (e.g., renaming an identifier), and correctly group related changes that span multiple lines. However, all of these approaches focus on producing a tree-based shortest edit script between ASTs. As shown in Decker et al. [10], the shortest edit script is a poor proxy for understandability and can often obscure meaningful changes. Furthermore, existing syntactic differencers [2–8] discard whitespace, comments, and preprocessor directives since these elements do not appear in the AST, even though developers read and modify them constantly.

Finally, semantic differencing approaches [11–14] go further and reason about behavioral equivalence and dependency relationships. However, these approaches operate at a higher level of abstraction and complement rather than replace syntactic or line-based differencing approaches.

Here, we present srcDiff [10], a scalable syntactic differencing tool built on the srcML infrastructure [15, 16] that preserves the complete source code and augments it with change-markup elements. The resulting srcDiff format [17] encodes these changes directly in srcML to support both automated downstream analysis and human-readable views. srcDiff applies a rule-driven algorithm that integrates matching and edit-script generation during AST traversal, using empirically derived domain-knowledge rules to select appropriate syntactic granularity and scope. This allows srcDiff to depart from the shortest edit script to produce deltas that better match developer expectations.

Our prior work [10], introduced the srcDiff approach and validated it using an early prototype. This paper presents the recently released first opensource version of the srcDiff tool. srcDiff is now available on GitHub[1] with a ready-to-use Docker[2] image. Language support has grown beyond C and C++ to include Java, with C# and Python actively in development. A Git plugin is available, and the tool integrates into any IDE or version control workflow with minimal configuration.

II. Related Work

The most popular differencing tools are textual and line-based, typically implementing a version of Myers' algorithm [18]. Examples include: GNU diff, those built into version control systems such as Git diff [1], GitHub's native diff view, and those built into IDEs such as Visual Studio, Eclipse, and IntelliJ. Several tools extend these textual approaches with additional refinements. Ldiff [19] extends GNU diff with fragment and move detection, spiff [20] highlights token-level changes within lines, Google diff-match-patch [21] operates at the character level, and LHDiff [22] tracks source code lines across versions using simhash-based similarity and heuristics. Despite these refinements, all remain fundamentally textual, lacking any knowledge of program syntax, and therefore do not convey changes beyond line-level granularity.

Syntactic [3] differencing approaches address this by comparing ASTs rather than text. The most widely used is GumTree [2], which uses heuristics to approximate the shortest edit script. Another influential approach, ChangeDistiller [3] derives an approximate shortest edit script via computing similarities. Subsequent enhancements to these tools, such as MTDIFF and IJM, further refine matching, edit script generation, and move detection [5–7]. Diff/TS [4], like GumTree, aims to approximate an optimal tree-edit distance. Difftastic [8] represents differencing as a shortest-path problem over a graph of syntax node positions, using Tree-sitter grammars [24]. However, it still relies on producing a shortest edit script, discards non-significant whitespace, and scales poorly on files with a large number of changes.

Unlike the aforementioned syntactic approaches, srcDiff employs domain-knowledge rules to determine when to deviate from a shortest edit script, producing deltas that better match developer expectations. It is also, to the best of our knowledge, the only syntactic differencer to fully preserve the complete source code, including whitespace, comments, and preprocessor directives. Other approaches lose this information in generating the AST they work upon. Furthermore, srcDiff inherits a unified grammar through srcML [15, 16], which produces a near-identical XML-based AST representation across C, C++, and Java, allowing the same rules and algorithm to apply across languages without modification. Table I summarizes these distinctions, where *Language Knowledge* indicates whether the tool applies rules derived from developer editing patterns and programming language syntax, rather than just matching tree structure. GumTree supports a unified grammar via srcML as a parser, but this remains under testing by its developers. *Per-File* (replicated from [10]) shows the total runtime on 1,307 file pairs (two are removed from the original study due to syntax errors). MTDIFF is excluded as it only supports a web-based view and cannot be timed fairly. However, since it is built on GumTree's infrastructure, it is expected to perform similarly. To further explore the scalability of these tools, a *Stress Test* is conducted using a single large file formed by concatenating all of the per-file original/modified files, respectively. Several tools are noted to fail on the stress test, either by reverting to simple line-based diffs when exceeding byte limits, experiencing timeouts, terminating due to memory exhaustion, or failing because their underlying parsers enforce strict conventional assumptions.

Other related work includes CLDiff [25], which processes changes and presents an alternative output. At the semantic level, Jackson and Ladd's Semantic Diff [13] summarizes the behavioral effects of modifications between program versions. Unrelated to it, despite the similar name, the proprietary tool SemanticDiff [9] highlights refactorings such as moved or repeated code. JDiff [11], LSdiff [12], and Dex [14] also operate at the semantic level. These tools produce output at a different level of abstraction. For more related work, see [10].

TABLE I. Comparison of Syntactic Differencing Tools. Time (Wall Clock) is the Average of Three Runs. Per-File is the Total Time on 1,307 Original/Modified Java File Pairs from the Jenkins Project, With Two Pairs Excluded from the Initial 1,309 Due to Syntax Errors. Stress Test (Highlights Scalability) Consists of Running on All Original/Modified, Each Combined as a Single File.

Tool	Full Source Preservation	Language Knowledge	Unified Grammar	Per-File (Seconds)	Stress Test (Seconds)
GumTree 3.0.0	No	No	Yes	1140.0	Fails
Diff/TS 0.3.6	No	Some	No	498.9	Fails
ChangeDistiller	No	No	No	527.2	Fails
IJM	No	No	No	1317.8	3046.2
MTDIFF	No	No	No	N/A	N/A
Difftastic 0.69.0	No	No	No	86.2	Fails
srcDiff	**Yes**	**Yes**	**Yes**	**82.2**	**104.3**

III. Syntactic Differencing via srcDiff

The srcDiff approach [10] incorporates a set of differencing rules (i.e., srcDiff rules) based on developer/domain knowledge. It is built on top of the srcML infrastructure [15, 16] (www.srcML.org). As such, it inherits the infrastructure's efficiency, robustness, and complete preservation of source code text. srcDiff's primary output is an extension to srcML's XML output format (i.e., srcML format) to encode information on the delta between source-code versions. This is known as the srcDiff format. Figure 1 shows an example of srcDiff on the original and modified versions of the `setImage` function from KOffice revisions 1026809–1026810 (top), which has been simplified to fit. The middle presents the line-based difference (i.e., GNU diff) on the left for comparison and srcDiff's unified view on the right. Deletes are in bold and highlighted in red with a strikethrough, while inserts are bold and highlighted in green. These are added to the GNU Diff and srcDiff format (bottom) to make it easier to compare/illustrate.

[1] GitHub: https://github.com/srcDiff/srcDiff
[2] Docker: https://hub.docker.com/r/srcdiff/ubuntu

[3] Some tools (e.g., Difftastic) use the term *structural* for AST-based differencing; we use the term *syntactic* throughout, following Mens [23].

Original

```
void Settings::setImage( KisImageSP image ) {
    m_widget->m_filter->setImage( image );
}
```

Modified

```
void Settings::setImage( KisImageSP image ) {
    if (m_options) {
        m_options->m_filter->setImage( image );
    }
}
```

GNU Diff Unified View

```
@@ -1,3 +1,5 @@
void Settings::setImage( KisImageSP image ) {
-   m_options->m_filter->setImage( image );
+   if (m_options) {
+       m_options->m_filter->setImage( image );
+   }
}
```

srcDiff Unified View

```
@@ -1 +1 @@
void Settings::setImage( KisImageSP image ) {
    if (m_options) {⏎
        m_widgetm_options->m_filter->setImage( image );
    }⏎
}
```

srcDiff Format

```
<function><type><name>void</name></type> <name><name>KisFilterOpSettings</name><operator>::</operator><name>setImage    \
</name></name><parameter_list>( <parameter><decl><type><name>KisImageSP</name></type> <name>image</name></decl>          \
</parameter> )</parameter_list> <block>{<block_content>
<diff:insert><diff:ws>    </diff:ws><if_stmt><if>if<diff:ws> </diff:ws><condition>(<expr><name>m_options</name></expr>) \
</condition><diff:ws> </diff:ws><block>{<block_content><diff:ws>
    </diff:ws><diff:common>    <expr_stmt><expr><call><name><name><diff:delete type="replace">m_optionsWidget           \
</diff:delete><diff:insert type="replace">m_options</diff:insert></name><operator>&gt;</operator><name>m_filterOption   \
</name><operator>&gt;</operator><name>setImage</name></name><argument_list>(<argument><expr><name>image</name></expr>   \
</argument>)</argument_list></call></expr>;</expr_stmt>
</diff:common><diff:ws>    </diff:ws></block_content>}</block></if></if_stmt><diff:ws>
</diff:ws></diff:insert></block_content>}</block></function>
```

Fig. 1. srcDiff example. The top shows a simplified version of the original and modified source code of the function `setImage` from KOffice revisions 1026809–1026810. As it works at the line-level, the GNU diff View is too coarse-grained and difficult to understand the changes, while, because it is lossless and works at the syntactic level, the srcDiff Unified View is accurate and easier to understand. The srcDiff Format provides an XML representation of the changes, which can be used for exploration, analysis, and manipulation. Deletes are in bold and highlighted in red with a strikethrough, while inserts are in bold and highlighted in green. These are added to the GNU diff and srcDiff Format for illustrative purposes.

In this example, the line-based GNU diff is too coarse-grained to accurately capture and easily understand the changes, whereas srcDiff, as a lossless syntactic differencer, correctly captures the changes and is easier to understand. The bottom section is the difference in the srcDiff format, which can be used for exploration and analysis. In Section III.A, we further discuss the srcML/srcDiff formats, Section III.B presents the algorithm used in srcDiff, Section III.C introduces the srcDiff rules, and III.D briefly summarizes the evaluation from [10].

A. srcML and srcDiff Format

The srcML format (see Figure 1) is an XML-based representation of source code created by wrapping source code text with XML tags containing AST information. The srcDiff format (see Figure 1) extends the srcML format with the addition of three primary XML tags: `diff:common`, `diff:delete`, and `diff:insert`. These tags allow for the markup of unmodified, deleted, and inserted code (including AST information), and collectively represent the complete set of changes (i.e., `delta`) needed to transform the original version into the modified version. A fourth tag `diff:ws` is used in the srcDiff format; however, its purpose is to ease the processing and collection of text in `diff` elements by separating whitespace from source code text.

The three primary srcDiff tags can also contain attributes. This includes the `type` attribute with the values `replace` to indicate code replacement and identifier renames and `convert` to indicate the conversion of one syntactic element to another (e.g., for-loop to while-loop), which is unique to srcDiff. For code moves, both the `diff:delete` and `diff:insert` tags receive a `move` attribute with a shared unique ID, pairing them to indicate the source and destination of the moved code.

Because srcML is a document format, srcDiff views changes as document changes rather than AST changes. Hence, where other tools require moves in addition to deletes and inserts to represent certain changes, srcDiff presents the delta as simple deletes/inserts. When code is physically moved in a document, srcDiff marks those moves. Attribute modifications in srcML are represented by storing both the original and modified values separated by `|`. For example, `filename="f1.cpp|f2.cpp"`, on the unit tag, is a rename of the file from `f1.cpp` to `f2.cpp`.

In this manner, srcDiff fully preserves the complete srcML of both versions and produces a single multi-version document from which either version can be exactly reconstructed. This unified representation supports automated change analysis and the generation of human-readable deltas.

B. srcDiff Algorithm

The srcDiff differencing algorithm compares two versions of source code by performing a preorder traversal of their ASTs in srcML, producing output that fully preserves the structure and ordering of both versions. For each matched node, a sequence differencing procedure, based on Myers' shortest edit script algorithm [18], is applied to the sequences of child elements to identify child-subtrees that are unchanged, deleted, inserted, or modified. Unchanged regions are marked as `diff:common`, while insertions and deletions are marked with `diff:insert` and `diff:delete`, respectively.

When delete(s) and insert(s) occurs at the same relative child position (i.e., a *change*), the srcDiff rules are used to decide how

to handle it. srcDiff considers three scenarios: 1) the delete(s)/ insert(s) are unrelated, 2) the delete(s) are previous versions of the insert(s), and/or 3) the delete(s) or insert(s) contain the other (i.e., nested, or descendants if viewing from AST). This process continues iteratively, refining differences until it reaches the desired granularity. Unlike other AST differencing approaches that separate matching and edit script generation phases, srcDiff integrates these operations while walking the tree structure.

C. srcDiff Rules

The srcDiff approach is driven by a set of empirically derived rules that select the most appropriate syntactic granularity for a given change and determine the scope of changes within the code. Rather than relying purely on tree-edit distance, these rules incorporate domain knowledge of how developers edit code and produce deltas that better reflect developer intent. The rules fall into three primary categories, summarized in Table II: *Match* rules, *Convertibility* rules, and *Nesting* rules.

Match rules determine when two syntactic elements of the same type should be paired for fine-grained differencing rather than treated as a full replacement. For example, the *Name Match* rule pairs a function with the same name across versions regardless of how much its body has changed. The *Condition Match* rule recognizes that a while-loop with the same condition is still the same loop even if its body is completely rewritten, rather than treating it as a deletion and insertion.

Convertibility rules handle the case where a developer changes one syntactic construct into a different type, such as converting a for-loop into a while-loop or a class into a struct. Rather than marking the original as deleted and the replacement as entirely new, srcDiff recognizes the conversion, marks it with a convert attribute in the output, and performs fine-grained differencing on the children of both constructs. Six conversion categories are defined: class-type, access region, logical control flow, else/else-if, statement-level, and cast conversions.

Nesting rules handle the case where existing code is wrapped inside a new construct or extracted from one. A typical example is a developer adding an if-statement around an existing block of statements where the statements themselves are not changed, but they now reside inside a new AST node. srcDiff applies this check iteratively after *Match* and *Convertibility* rules have been applied and selects the candidate that produces the highest textual similarity while covering the largest span of children.

When none of the above rules apply, srcDiff falls back to general syntactic similarity (comparing the number of matching AST child nodes) and textual similarity (comparing the lexical leaf tokens) checks. If none of the checks and rules apply, srcDiff considers them unrelated edits, marking them as completely deleted and inserted. Together, these guarantee that a reasonable delta is always produced. For complete formal definitions of the algorithm and all rules, see [10].

D. Evaluation

The srcDiff approach is evaluated in our prior work via a controlled within-participant user study comparing srcDiff against GumTree, GitHub's diff view, and Mergely across fourteen real-world Java change samples drawn from Elasticsearch, Google Guava, SLF4J, and jEdit. Sixty-nine participants evaluated each tool on accuracy, understandability, readability, and preference. srcDiff (including ties) is rated the most understandable, most readable, and most preferred approach across the sample set. Among expert participants, those who rated themselves as highly skilled with differencing tools, srcDiff is ranked (in all cases) the most accurate, understandable, and readable. srcDiff is also the most preferred in all but one case. No other tool is ranked best more than twice across the combined samples and overall evaluation. The results support the hypothesis that minimal edit distance is a poor proxy for understandability and that domain-knowledge rules yield more meaningful deltas. See [10] for further explanation/details.

TABLE II. SRCDIFF DIFFERENCING RULES ORGANIZED BY CATEGORY.

	Rule	Syntactic Category	Match Condition
Match	Always Match	identifiers, types, operators, literals, blocks, lists, etc.	Always paired for fine-grained differencing
	Any Similarity Match	expressions, expression statements	Any textual similarity without excessive dissimilarity
	Name Match	functions, classes, decls, etc.	Names are identical
	Condition Match	loop and switch statements	Conditions are identical
	If-Statement Match	If-statements	Then-clauses match or conditions match with other checks based on if a block or else-statement is present
Convertibility	Class	classes, structs, unions, enums	Same name or passes similarity rules
	Access Region	public, private, protected	Passes similarity rules
	Logical	for, while, if-statements	Same condition or passes similarity rules
	Else	else, else-if branches	Passes similarity rules
	Statement	expressions, declarations, return statements	Expressions pass similarity rules
	Cast	C++ cast forms	Passes similarity rules
Nesting	Nesting	-	Grammar derived plus other checks. See [10]

IV. RUNNING SRCDIFF

srcDiff is a command-line tool that takes original/modified source-code pairs as input, either as files or directories. By default, output is printed to the terminal; the `-o/--output` option saves it to a file. The following computes the difference between two files and saves the result to `srcdiff.xml`:

```
srcdiff original.cpp modified.cpp -o srcdiff.xml
```

The following does the same with directories (all files/subdirectories) where files are matched via paths:

```
srcdiff source_v1 source_v2 -o srcdiff.xml
```

Human-readable views can be generated. The following outputs a unified view to the terminal via `-u/--unified`:

```
srcdiff original.cpp modified.cpp -u
```

A side-by-side view can be generated with `-y/--side-by-side`, which takes an integer for the tab stop size:

```
srcdiff original.cpp modified.cpp -y 4
```

Both views support customizable syntax highlighting themes. Additionally, both views support output as HTML for integration into web-based workflows. The following outputs the unified view as HTML:

```
srcdiff original.cpp modified.cpp -u --html
        -o diff.html
```

Once two versions of source code are in the srcDiff format, the srcML/srcDiff infrastructure can be used to explore, analyze, and manipulate code changes. Various XML tools such as XPath can quickly query source code and changes. For example, the following XPath finds every function that has been modified:

```
//src:function[.//diff:delete or .//diff:insert]
```

Likewise, the following finds all renamed identifiers:

```
//src:name[.//diff:delete[@type='replace']
                        and not(.//src:*)]
```

The Docker image is provided with code examples, such as the Apache POI[4] project with srcDiff pre-configured as the Git diff driver. The following shows syntactic diffs across commits:

```
git log --ext-diff -p
```

To conduct analysis on the output of srcDiff, we can use more sophisticated XML processing techniques, such as SAX (Simple API for XML), TextReader [26], and DOM. An example application of srcDiff utilizing this kind of analysis can be found in Someya et al. [27], where a SAX parser approach (in Python) is used to count the tokens deleted, inserted, and common to two versions of code, which is then used as the basis for a similarity metric. The SAX parser code is included in the Docker image. Lastly, because srcDiff produces a standard XML document, the format can be easily augmented by external tools (e.g., attaching authorship) or transformed via XSLT into entirely custom representations beyond the native views.

V. Conclusion

srcDiff is a programmer-centric tool for syntactic differencing that preserves complete source code and marks up changes at the syntactic level using a set of empirically derived domain knowledge rules. The tool extends the srcML infrastructure with rule-driven differencing, integrating matching and edit-script generation during AST traversal to produce meaningful deltas. The resulting srcDiff format supports both automated downstream analysis via standard XML tools and human-readable output in unified, side-by-side, and HTML views with customizable syntax highlighting. A Git plugin is already available, and future work includes plugins for major IDEs such as IntelliJ and VS Code. The tool currently supports C, C++, and Java. We are actively adding full support for additional languages (e.g., C# and Python). The tool is opensource and available on GitHub.

Acknowledgment

This work was supported in part by a grant from the US National Science Foundation: CNS 22-32593/32594.

References

[1] "git-diff Documentation" Available: https://git-scm.com/docs/git-diff.

[2] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Montperrus, "Fine-grained and Accurate Source Code Differencing," at the 29th ACM/IEEE International Conference on Automated software engineering (ASE'14), Vasteras, Sweden, 2014.

[3] B. Fluri, M. Wursch, M. Pinzger, and H. C. Gall, "Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction," *IEEE Transactions on Software Engineering*, vol. 33, pp. 725–743, 2007.

[4] M. Hashimoto and A. Mori, "Diff/TS: A Tool for Fine-Grained Structural Change Analysis," at the 15th Working Conference on Reverse Engineering (WCRE'08), 2008, pp. 279–288.

[5] J. Matsumoto, Y. Higo, and S. Kusumoto, "Beyond gumtree: A hybrid approach to generate edit scripts," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 550–554.

[6] V. Frick, T. Grassauer, F. Beck, and M. Pinzger, "Generating accurate and compact edit scripts using tree differencing," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 264–274.

[7] Georg Dotzler and Michael Philippsen, "Move-optimized Source Code Tree Differencing," at the Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, Singapore, Singapore, 2016.

[8] "Difftastic." Available: https://github.com/wilfred/difftastic.

[9] "SemanticDiff." Available: https://github.com/Sysmagine/SemanticDiff.

[10] M. J. Decker, M. L. Collard, L. G. Volkert, and J. I. Maletic, "srcDiff: A Syntactic Differencing Approach to Improve the Understandability of Deltas," *Journal of Software: Evolution and Process*, vol. 32, no. 4, p. e2226, Apr. 2020.

[11] T. Apiwattanapong, A. Orso, and M. Harrold, "JDiff: A differencing technique and tool for object-oriented programs," *Automated Software Engineering*, vol. 14, pp. 3–36, 2007.

[12] Alex Loh and Miryung Kim, "LSdiff: a program differencing tool to identify systematic structural differences," at the Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, Cape Town, South Africa, 2010.

[13] D. Jackson and D. A. Ladd, "Semantic Diff: A Tool for Summarizing the Effects of Modifications," at the IEEE International Conference on Software Maintenance (ICSM'94), 1994, pp. 243–252.

[14] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine, "Dex: A Semantic-Graph Differencing Tool for Studying Changes in Large Code Bases," at the 20th IEEE International Conference on Software Maintenance (ICSM'04), 2004, pp. 188–197.

[15] M. L. Collard, M. J. Decker, and J. I. Maletic, "Lightweight Transformation and Fact Extraction with the srcML Toolkit," in *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, 2011, pp. 173–184.

[16] M. L. Collard, M. J. Decker, and J. I. Maletic, "srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code: A Tool Demonstration," in *29th IEEE International Conference on Software Maintenance (ICSM)*, 2013, pp. 516–519.

[17] J. I. Maletic and M. L. Collard, "Supporting Source Code Difference Analysis," at the 20th IEEE International Conference on Software Maintenance (ICSM'04), 2004, pp. 210–219.

[18] Eugene W. Myers, "An O(ND) Difference Algorithm and its Variations," *Algorithmica*, vol. 1, p. 16, 1986.

[19] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta, "Ldiff: An enhanced line differencing tool," at the 31st International Conference on Software Engineering (ICSE'09), 2009, pp. 595–598.

[20] "spiff." Available: https://github.com/dontcallmedom/spiff.

[21] "google-diff-match-patch." Available: https://github.com/google/diff-match-patch.

[22] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and M. Di Penta, "LHDiff: A Language-Independent Hybrid Approach for Tracking Source Code Lines," at the 29th IEEE International Conference on Software Maintenance (ICSM'13), 2013, pp. 230–239.

[23] T. Mens, "A State-of-the-Art Survey on Software Merging," *IEEE Transactions on Software Engineering*, vol. 28, no. 5, pp. 449–462, Aug. 2002.

[24] "Tree-sitter Grammars." Available: https://github.com/tree-sitter-grammars.

[25] K. Huang *et al.*, "ClDiff: generating concise linked code differences," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 679–690.

[26] "XmlReader." Available: http://veillard.com/XML/xmlreader.html.

[27] K. Someya, L. Chen, M. J. Decker, and S. Hayashi, "How Much Can a Behavior-Preserving Changeset Be Decomposed into Refactoring Operations?," in *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2025, pp. 809–814.

[4] Commit: 03fc1ddea1be6defeca7de71705def32a5ea740b