Spork: Structured Merge for Java With Formatting Preservation

Simon Larsen, Jean-Rémy Falleri, Benoit Baudry, Martin Monperrus

Spork: Structured Merge for Java with Formatting Preservation

Simon Larsén, Jean-Rémy Falleri,
Benoit Baudry, *Member, IEEE*, and Martin Monperrus, *Member, IEEE*,

KTH Royal Institute of Technology, Univ. Bordeaux, Bordeaux INP, CNRS, LaBRI, IUF

Abstract—The highly parallel workflows of modern software development have made merging of source code a common activity for developers. The state of the practice is based on line-based merge, which is ubiquitously used with "git merge". Line-based merge is however a generalized technique for any text that cannot leverage the structured nature of source code, making merge conflicts a common occurrence. As a remedy, research has proposed structured merge tools, which typically operate on abstract syntax trees instead of raw text. Structured merging greatly reduces the prevalence of merge conflicts but suffers from important limitations, the main ones being a tendency to alter the formatting of the merged code and being prone to excessive running times. In this paper, we present SPORK, a novel structured merge tool for JAVA. SPORK is unique as it preserves formatting to a significantly greater degree than comparable state-of-the-art tools. SPORK is also overall faster than the state of the art, in particular significantly reducing worst-case running times in practice. We demonstrate these properties by replaying 1740 real-world file merges collected from 119 open-source projects, and further demonstrate several key differences between SPORK and the state of the art with in-depth case studies.

Index Terms—Version control, structured merge

1 Introduction

Branching development paths is an unavoidable part of modern software engineering [1], and developers spend anywhere from a few hours to several work days each month on integrating changes from others [2]. This activity is known as "merging" code, per the terminology of mainstream version control systems such as GIT. Nearly all developers use *line-based* merge, which operates on lines of text as atomic units. It is often referred to as *textual* or *unstructured* merge [3], [4]. This form of merging is simple and generalizes to any text, but is prone to cause so-called *merge conflicts* when changes on branches under merge affect the same or adjacent lines. Such conflicts can be difficult for developers to resolve, and may even cause them to simply discard a branch that is causing numerous conflicts [5].

Merge conflicts are ubiquitous with line-based merge, with conflicts appearing in about 9% to 19% of merges [6], [7], [8], [9]. However, many such conflicts are spurious, because changes on overlapping lines are not necessarily semantically or syntactically conflicting. This is a fundamental limitation of line-based merge: it does not capture the underlying structure or meaning of the text. For example, if two branches add different methods in the same place in a JAVA file, a line-based merge of said branches yields a conflict, even though the methods can in fact be safely inserted together in any order.

To address this spurious conflict problem, the state of the art is *structured* merge, where the merge process typically acts on *abstract syntax trees* (AST) [3], [4], [10], [11]. Structured merge has two main advantages: first, it is less influenced by formatting differences than line-based merge, and second, it can leverage the syntax and semantics of the considered programming language. For instance, in JAVA, it is useful for a merge tool to know that duplicated statements are allowed, but that duplicated fields are not, or that the order of methods in a class is not important [4].

We observe that the state of the art of structured merge is prone to two main issues. First, any tool that performs AST transformations must conclude with *pretty-printing*, which in this context is the act of turning an AST into its textual representation, i.e. source code. This can fundamentally alter the formatting of the code [12], [13], which is undesirable due to the important role that formatting plays in source code readability [14]. Second, structured merge is known for being slow, with the time complexities of the underlying algorithms often being $O(n^2)$ or worse [4].

In this paper, we address these issues with a new structured merge tool, called SPORK. SPORK is tailored to the JAVA programming language, leveraging both syntax and semantics of important language constructs to avoid or resolve conflicts. A key technical novelty of SPORK is that it builds upon the merge algorithm of the 3DM merge tool for XML documents [15]. In SPORK, we both augment the 3DM algorithm, and demonstrate that the core principles are applicable to the JAVA programming language. As we show in our evaluation, SPORK improves upon the state of the art with respect to the aforementioned problems. First, SPORK reuses source code from the input files when pretty-printing. This improves formatting preservation over the state of the art in more than 90% of merged files, with 4 times better preservation in the median case. Second, SPORK's running time performance slightly improves upon the competition in the median case, but more importantly it significantly reduces the quantities and magnitudes of the

largest running times.

To summarize, our contributions are:

- A novel structured merge approach for JAVA, uniquely based on the 3DM algorithm [15], leveraging domain knowledge of JAVA to detect and resolve conflicts.
- SPORK, a publicly available prototype implementation: https://github.com/KTH/spork
- An evaluation over 890 merge scenarios comprising 1740 file merges, showing that SPORK is fast and accurate enough to be used in practice, and formatting preserving. To our knowledge, we are the first to systematically report on formatting preservation for fully AST-based structured merge.
- A well-documented benchmarking suite for future research, to study and evaluate JAVA merge tools.

This article is based on the master's thesis by the first author done at KTH Royal Institute of Technology [16].

2 Background

2.1 Version Control and Merging

With the rise in popularity of *distributed version control systems* (DVCS) [17], [18], the need for merging in software development has increased [1]. The state of the practice is to use unstructured merge, which operates on raw text, typically using lines of text as atomic units. This is fundamentally limited, as a line of text does not represent the structure of source code. For example, the two lines `int a = 2;` and `int a=2;` are structurally and semantically equivalent, yet the raw text of those lines differ by whitespace. The impedance mismatch between lines and code structure gives rise to needless merge conflicts.

A typical example is if one commit changes the indentation style of some file, while a parallel one changes the actual code. Such changes are structurally and semantically compatible, but merging the commits with a line-based merge results in a merge conflict due to the purely textual differences.

Semistructured merge tools attempt to address some of the problems of unstructured merge by making use of some structural information in the source code [19], [20], [21], [22]. They work by identifying high-level constructs (e.g. fields and methods) in the source code, and then merging the content of these modularized units with line-based merge. Semantic information such as the insignificance of the order of methods within a type can then be utilized to automatically resolve conflicts. However, within type members, most notably methods, semistructured merge still suffers from all of the limitations of unstructured merge.

Structured merge tools go one step further and turn the source code into a fully resolved AST [4], [11], [15], [23], [24]. This allows for a fine-grained merge that respects syntax even within type members, but also creates a new problem: as the AST abstracts away formatting, the conversion back from AST to source code, *pretty-printing*, may impose a completely different formatting style on the merged files. This may be detrimental to source code quality, as formatting is an integral part of maintainability and readability [14]. In addition, developers care about the formatting they put in place, as shown by the sheer amount of style guides that exist[1][2][3] and the existence of formatting enforcers such as CHECKSTYLE[4]. Current structured merge tools do not preserve formatting, and this poses a major obstacle for widespread adoption of structured merge. Another problem with the state of the art in structured merge revolves around running times, which can become excessive for larger merges due to algorithms with time complexities that are quadratic or worse [4].

Left	Base	Right
add(-a,b,1)	add(a,b)	sum(-a,b,c)

TABLE 1: The left, base and right revisions of a line of code

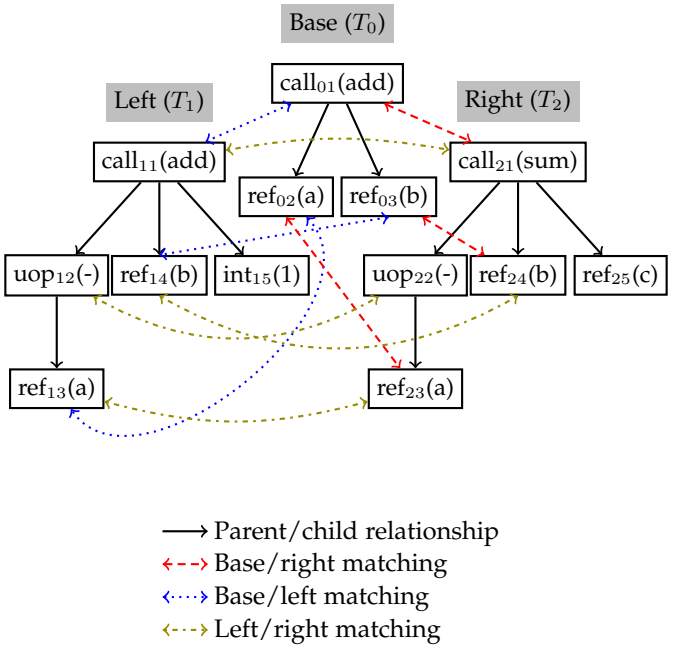

Fig. 1: Pairwise matchings between simplified ASTs of the function calls in Table 1. Each node is subscripted with a unique identifier xy, where x indicates the tree the node belongs to and y is unique within tree x.

2.2 The 3DM Algorithm

3DM is a state of the art merge algorithm created by Lindholm [15]. It performs a *three-way merge* between the two current revisions (*left* and *right*) of some file, and the version from which these are derived (the *base*) [15]. This technique is employed by most merge tools [3], [4], [25]. The most novel part of 3DM is the merge algorithm, which is a generalized merge algorithm for ordered trees[5] that we refer to as 3DM-MERGE. This section presents the theoretical details of 3DM-MERGE that are relevant for our own work.

1. https://google.github.io/styleguide/javaguide.html
2. https://wiki.openjdk.java.net/display/HotSpot/StyleGuide
3. https://www.cs.cornell.edu/courses/JavaAndDS/JavaStyle.html
4. https://checkstyle.sourceforge.io/
5. The child list of each node is an ordered list of nodes

TABLE 2: PCS set of T_0 in Figure 1, ordered into child lists

Node	PCS child list
$\perp$	$(\perp, \dashv, call_{01}),\ (\perp, call_{01}, \vdash)$
$call_{01}$	$(call_{01}, \dashv, ref_{02}),\ (call_{01}, ref_{02}, ref_{03}),\ (call_{01}, ref_{03}, \vdash)$
ref_{02}	$(ref_{02}, \dashv, \vdash)$
ref_{03}	$(ref_{03}, \dashv, \vdash)$

TABLE 3: Class representatives mapping for Figure 1

Node ID	01	02	03	11	12	13	14	15	21	22	23	24	25
Class rep.	01	02	03	01	12	02	03	15	01	12	02	03	25

2.2.1 Data Structures of 3DM

3DM-MERGE does not operate on a traditional tree structure, but on an abstract representation of an ordered tree, called a *change set* [15]. This is composed of two primary data types. The first of these is the *parent-child-successor* (PCS) triple, which encodes the structure of the tree. A PCS triple is written in the form $pcs(parent, pred, succ)$, where $parent$ is an arbitrary tree node, $pred$[6] is any of $parent$'s children, and $succ$ is the node in $parent$'s child list that directly succeeds $pred$. For a given tree, the set of triples with $parent = x$ therefore encode the child list of x.

There are also three kinds of *virtual* nodes: a virtual root $\perp$, a virtual start of a child list $\dashv$ and a virtual end of a child list $\vdash$ [15]. These nodes mark the boundaries of the tree's structure. As an example of applying the PCS concepts, consider the base revision function call in Table 1 and its corresponding AST T_0 in Figure 1, and how the syntactical structure is fully encoded by the PCS set in Table 2. Note that nodes are identified by ID, and not by content. For example, a variable reference $ref_{01}(a)$ is different from another reference $ref_{02}(a)$, even though they are identical apart from ID. We often refer to nodes by their IDs alone to reduce the verbosity of figures and tables. For example, the PCS set in Table 2 is equivalent to the base revision PCS set in Table 4.

The second data type is the *content* tuple, written $c(v, m)$, where v is any concrete node and m is v's content, the exact form of which is domain dependent. In general, the content m of a tree node v is all data related to v that does not impact the structure of the tree. We express m as a set of values. As a concrete example, the content tuples of T_0 in Figure 1 is $\{c(01, \text{"}add\text{"}), c(02, \text{"}a\text{"}), c(03, \text{"}b\text{"})\}$. Note that m is a set. If for example the base and left revision of `add` had modifiers `public` and `private`, respectively, then the content tuples would be $c(01, \{public, \text{"}add\text{"}\})$ and $c(11, \{private, \text{"}add\text{"}\})$. The change set is simply the union of the content tuples and PCS triples of a tree.

A change set is said to be *consistent* if each node v has at most 1) one parent x, 2) one predecessor y, 3) one successor z and 4) one content set m [15]. A consistent change set is unambiguous, and a tree always encodes a consistent change set. As the consistency criteria allow a node to have less than one parent, predecessor, successor and content set, a consistent change set does not necessarily encode a well-formed tree.

6. In the original paper, this node is called *child*

2.2.2 Matchings and Class Representatives

3DM-MERGE makes use of tree matchings to determine where trees to be merged are similar [15]. We define a tree matching as a symmetric relation between the nodes of two trees T_i and T_j, where each node $v \in T_i$ can be matched to at most one node $w \in T_j$. The details of how a match between nodes v and w is computed varies greatly between matching algorithms. The most powerful algorithms consider many factors, including the nodes' relative positions, their contents, as well as the similarities of their subtrees [26].

For a three-way merge, three pairwise matchings are typically required: base/left, base/right, and left/right. Figure 1 illustrates this for a simple merge scenario. Note for example that the base/left matching contains a node matching between root nodes 01 and 11, which is reasonable as the root nodes of the base and left revisions are identical and in the same position. Note also that the base/right matching contains a node matching between root nodes 01 and 21 even though the method names differ, as their positions and subtrees are similar enough.

The tree matchings are then used to create a *class representatives* mapping [15]. Each node v is mapped to precisely one class representative w, which we denote with $(v \to w)$ and refer to as a *classmapping*. All nodes assigned to the same class representative are considered equivalent by 3DM-MERGE. Formally, let T_0, T_1 and T_2 be the base, left and right revisions, respectively. A node $v \in T_i$ is classmapped to $w \in T_j$, i.e. $(v \to w)$, if the following three criteria are met: 1) v is matched to w, 2) $j \leq i$ and 3) there is no other node $u \in T_k$ where v is matched to u and $k < j$. Note that a classmapping is directional, so $(v \to w) \implies (w \to v)$.

Intuitively, a node without any matches in other trees is classmapped to itself, and the base revision takes precedence over the left revision, which in turn takes precedence over the right. This is evident from Table 3, which shows the class representatives mapping produced from the matchings in Figure 1. For example, all nodes in the base revision are classmapped to themselves, and $(21 \to 01)$ even though there is also a node matching between 11 and 21, showing the base revision's precedence. Similarly, $(22 \to 12)$ instead of the other way around, showing the left revisions precedence over the right.

2.2.3 Merging in 3DM

3DM-MERGE operates in two distinct phases. First, it converts the AST revisions into change sets, with each node mapped to its class representative, and initializes the *raw merge* as the set union of these change sets. Unless all input revisions are identical, the raw merge always contains violations of the consistency criteria presented in Section 2.2.1, so-called *inconsistencies*. For example, the two PCS triples $pcs(x, y, z)$ and $pcs(x', y, z)$ violate the criterion that each node has a unique parent. The second and most important phase of 3DM-MERGE is dedicated to finding and removing inconsistencies, with the end-goal of turning the raw merge into a consistent change set.

Consider Table 4, which shows an example merge of the trees in Figure 1 in terms of PCS elements only. Note how the identical elements of the revisions are merged simply by the nature of a set union, such as $(01, \dashv, 12)$ that is

TABLE 4: PCS sets of the trees in Figure 1, the raw merge of these and the finished merge. All nodes are presented as their class representative IDs.

Revision	PCS set
Left	$(\bot, \dashv, 01), (\bot, 01, \vdash),$ $(01, \dashv, 12), (01, 12, 03), (01, 03, 15), (01, 15, \vdash),$ $(12, \dashv, 02), (12, 02, \vdash), (02, \dashv, \vdash), (03, \dashv, \vdash), (15, \dashv, \vdash)$
Base	$(\bot, \dashv, 01), (\bot, 01, \vdash),$ $(01, \dashv, 02), (01, 02, 03), (01, 03, \vdash), (02, \dashv, \vdash), (03, \dashv, \vdash)$
Right	$(\bot, \dashv, 01), (\bot, 01, \vdash),$ $(01, \dashv, 12), (01, 12, 03), (01, 03, 25), (01, 25, \vdash),$ $(12, \dashv, 02), (12, 02, \vdash), (02, \dashv, \vdash), (03, \dashv, \vdash), (25, \dashv, \vdash)$
Raw merge	$(\bot, \dashv, 01), (\bot, 01, \vdash),$ $(01, \dashv, 02), (01, 02, 03), (01, 03, \vdash), (01, \dashv, 12),$ $(01, 12, 03), (01, 03, 15), (01, 15, \vdash),$ $(01, 03, 25), (01, 25, \vdash), (12, \dashv, 02), (12, 02, \vdash),$ $(02, \dashv, \vdash), (03, \dashv, \vdash), (15, \dashv, \vdash), (25, \dashv, \vdash)$
Merge	$(\bot, \dashv, 01), (\bot, 01, \vdash),$ $(01, \dashv, 12), (01, 12, 03), (01, 03, 15), (01, 15, \vdash),$ $(01, 03, 25), (01, 25, \vdash), (12, \dashv, 02), (12, 02, \vdash),$ $(02, \dashv, \vdash), (03, \dashv, \vdash), (15, \dashv, \vdash), (25, \dashv, \vdash)$

present in both left and right revisions, yet only appears once in the raw merge. The raw merge also contains numerous inconsistencies that need to be processed. 3DM-MERGE identifies these by iterating over each element δ of the change set, and searching for another element δ' such that δ and δ' are inconsistent. For example, given $\delta = (01, \dashv, 02)$, then $\delta' = (01, \dashv, 12)$ is found to be inconsistent. Note that δ is present in the base revision, while δ' is not. The inconsistency can therefore be resolved by removing δ, thus preserving the change represented by δ'. We refer to such an inconsistency as *soft*.

However, now consider the inconsistent pair $\delta = (01, 03, 15)$ and $\delta' = (01, 03, 25)$. Neither element is present in the base revision, and therefore removing either would cause change information to be lost. We refer to this as a *hard* inconsistency, and these are always caused by a conflict on the AST level. In this case, the conflict is caused by the left and right revisions inserting the nodes $int_{15}(1)$ and $ref_{25}(c)$ in the same place. Note the terminology used; *conflict* refers to incompatible changes to the ASTs, and *inconsistency* refers to a violation of the consistency criteria in the change set.

The same principles apply to content inconsistencies. For example, there is a content inconsistency between $c(01, "add")$ and $c(01, "sum")$[7]. This is a soft inconsistency as $c(01, "add")$ is present in the base revision, which can therefore be removed. Hard content inconsistencies are analogous to hard PCS inconsistencies, and occur when neither of the inconsistent elements are present in the base revision.

3 Technical contribution: SPORK

SPORK performs structured merging of JAVA files in 5 distinct phases, which are illustrated schematically in Figure 2. The first phase consists of parsing source files into ASTs, as described in Section 3.1. It is followed by a matching phase, in which tree matchings are computed as described in Section 3.2. These matchings are used in SPORK's variation of 3DM-MERGE, as described in Section 3.3. SPORK then enters a composite phase in which it handles conflicts and builds a merged AST, as described in Section 3.4. The final phase is responsible for pretty-printing the merged AST, as described in Section 3.5.

3.1 Parsing

SPORK uses the SPOON library [27] to parse JAVA source files into ASTs. In our example, Spoon is responsible for going from the raw source code in Table 1 to their corresponding ASTs in Figure 1 At this point, SPORK identifies and stores the style of indentation in the source file as the amount of tabs or spaces that precede top-level type members. This is necessary to later be able to print the merged file with the correct indentation. SPOON itself also stores the original source code of each parsed file, which SPORK in certain cases directly reuses to respect arbitrary formatting styles. This is further detailed in Section 3.5.

3.2 Tree Matching

SPORK uses GUMTREE[8,9] [26] to compute the pairwise base/left, base/right and left/right matchings between the ASTs. GUMTREE for example produces the matchings shown in Figure 1.

The base/left and base/right matchings allow SPORK to align the two changed revisions with the base. The left/right matching is primarily used to merge identical or near-identical additions in the left and right revisions. These matchings ground the set properties utilized in the raw merge, as shown in Table 4.

3.3 Merging Approach

Merging is the primary technical contribution in SPORK, as most of the functionality is implemented directly in the tool itself. The implementation is based on 3DM-MERGE, which is described in Section 2.2.

3.3.1 Mapping to class representatives

The mapping to class representatives largely follows the theoretical ideas presented in Section 2.2.2. First, nodes are classmapped to themselves if they are unmatched, or to at most one node in another revision according to the matching prioritization presented in Section 2.2.2. Ultimately, this results in a class representatives mapping like that of Table 3.

The left/right matching is however prone to contain spurious matchings, and carelessly adding these to the class representatives mapping can cause unnecessary conflicts. For example, if the left and right revisions add identical method parameters to different methods, the parameters may be matched due to their similarity, even though they are entirely unrelated. To reduce the effect of spurious matchings, SPORK implements two heuristics to decide whether or not to classmap $(v_{right} \rightarrow v_{left})$ given a match between v_{right} in the right revision and v_{left} in the left revision. First, the matching is ignored if any of the nodes are already classmapped to a node v_{base} in the base revision. This prevents the classmappings $(v_{left} \rightarrow v_{base})$ and

7. Note that nodes are mapped to class representatives

8. https://github.com/gumtreediff/gumtree
9. https://github.com/spoonlabs/gumtree-spoon-ast-diff

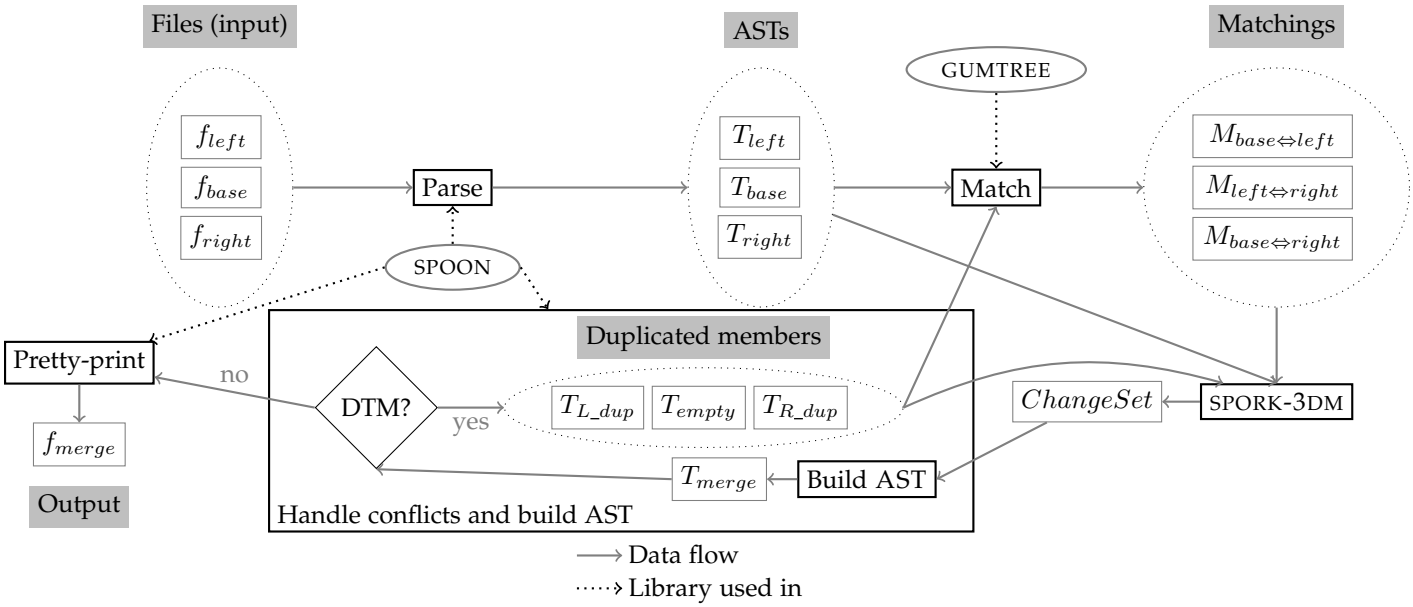

Fig. 2: Schematic drawing of SPORK's phases. Thin-lined rectangles represent data, filled rectangles are labels describing the closest data, and related data is grouped within dotted outlines. Thick-lined rectangles represent phases and ellipses represent libraries. "DTM"=duplicated type member.

$(v_{right} \rightarrow v_{left})$ from coexisting, which avoids a strange situation in which a node from the left revision appears in the right revision's change set, but not the left revision's. Second, the parents of v_{left} and v_{right} must already be classmapped to the same class representative. This prevents unrelated matchings, such as matchings between method parameters of different methods, from making their way into the class representatives mapping.

Adding eligible left/right matches to the class representatives is performed with a top-down scan of the left tree. The fact that the scan is top-down is important, as it allows arbitrarily complex subtrees to be incrementally mapped as long as their roots have parents that are already mapped to the same class representative.

3.3.2 Converting an AST to a change set

A tree can be converted into a change set by traversing it top-down and creating a PCS child list for each node, as well as extracting each node's content. This corresponds to the process of going from the base tree in Figure 1 to the PCS triples in Table 2 and the associated content set shown in Section 2.2.1.

For some complex nodes, naively building a PCS child list of their direct children is insufficient to achieve appropriate separation between distinct syntactical elements. For example, consider the method declaration in Listing 1. In SPOON, parameters and thrown types are considered direct children of the method node, and so appear in its child list. Figure 3 shows a schematic AST with a naively built child list for the method node in Listing 1. As the end of the list of parameters is adjacent to the beginning of the list of thrown types, structural modifications to the former may conflict with structural modifications to the latter.

To avoid collections of elements of different types within a child list to conflict with each other, SPORK inserts *intermediate virtual nodes* when building the PCS structure. A

Listing 1: A JAVA method declaration

```
int div(int lhs, int rhs) throws ArithmeticException
    { ... }
```

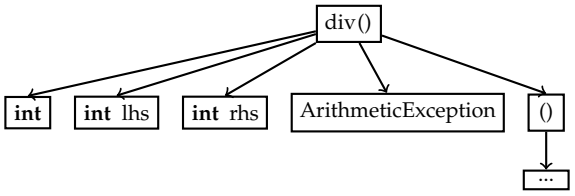

Fig. 3: Schematic drawing of naively built child list for the method declaration in Listing 1

schematic example of this is shown in Figure 4, where the `parameters` and `thrown` virtual nodes separate the previously adjacent parameters and thrown types. It is important that all applicable intermediate virtual nodes are inserted even if the parent node has no children of the corresponding types, as otherwise conflicts can occur due to insertions and deletions of the virtual nodes themselves.

3.3.3 Core SPORK Algorithm

SPORK implements a non-trivial variation of 3DM-MERGE, called SPORK-3DM. The key concepts of the algorithm are presented with pseudocode in Figure 5. To ease understanding, non-obvious helper functions are described in Figure 6.

The merge function of Figure 5 shows the SPORK-3DM algorithm at a high level of abstraction. It starts out with converting the input trees to change sets, all nodes being mapped to their class representatives. The union of these change sets forms the initial raw merge mergeCS, which as noted in Section 2.2.3 may contain inconsistencies for any

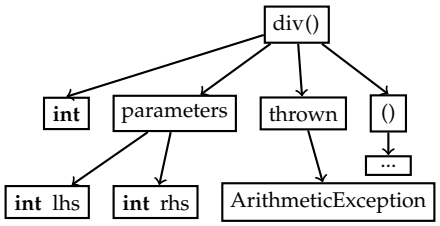

Fig. 4: Schematic drawing of the method declaration in Listing 1 with intermediate virtual nodes for the parameters and thrown types

non-trivial merge. The loop starting on line 6 is concerned with making mergeCS consistent by removing soft inconsistencies, and recording any hard inconsistencies. Acting on hard inconsistencies, known as *conflict handling*, is not part of 3DM-MERGE nor SPORK-3DM and SPORK's conflict handling is described in Section 3.4.

The removeSoftPcsInconsistencies function is the heart of the algorithm: if two PCS are found to be inconsistent, SPORK removes any that is in the base revision. If neither is in the base revision, they are in a hard inconsistency. Note that each invocation of the function is concerned only with PCS that are inconsistent with the input variable pcs. Therefore, if the input pcs is found to be in the base revision, all inconsistencies related to it are resolved by removing it from the change set, hence the early return on line 16. Table 5 shows the effects of a series of invocations of this function on the raw merge of Table 4, until the final merge is achieved.

The removeSoftContentInconsistencies function follows the same principles as removeSoftPcsInconsistencies. It is however simplified due to each content tuple belonging to precisely one tree node, and each tree node having at most one content tuple from each revision. A hard content inconsistency is therefore local to the node in which it occurs, and is identified by there being more than one non-base content tuple. In a three-way merge, the only possibility is that there are precisely two non-base content tuples: one from the left revision, and one from the right.

SPORK-3DM differs from 3DM-MERGE in two key aspects. First, 3DM-MERGE iterates over all elements of the change set and intermingles the activities of processing content and PCS triples. SPORK-3DM on the other hand iterates only over the PCS triples and separates the processing of content tuples and PCS triples, which makes it possible to reason about the merging of structure and content separately. Due to how far removed the merging of content is from the merging of PCS triples, it is perfectly viable for an implementation of SPORK-3DM to defer the merging of content until building an AST from the merged change set. Second, there is a key functional difference in how the algorithms discover inconsistencies. 3DM-MERGE finds at most one inconsistent element per iteration of the primary loop, while SPORK-3DM finds all of them. With the original algorithm, hard inconsistency detection sometimes becomes non-deterministic when the same PCS triple is involved in inconsistencies with many other triples.

Algorithm 1: Spork-3dm

```
1  Function merge is
       Data: base, left, right: Tree, classReps: Map[Tree,
             Tree]
       Result: ChangeSet
2      baseCS = toChangeSet(base, classReps)
3      leftCS = toChangeSet(left, classReps)
4      rightCS = toChangeSet(right, classReps)
5      mergeCS = unionOf(baseCS, leftCS, rightCS)
6      for pcs ∈ copyOf(mergeCS.pcsSet) do
7          removeSoftPcsInconsistencies(pcs,
             mergeCS, baseCS)
8          handleContent(pcs, mergeCS, baseCS)
9      return mergeCS
10 Function removeSoftPcsInconsistencies is
       Data: pcs: PCS, mergeCS, baseCS: ChangeSet
11     inconsistencies =
         getAllInconsistentPcs(pcs, mergeCS)
12     if size(inconsistencies) is 0 then
13         return
14     if pcs ∈ baseCS then
15         removePcs(mergeCS, pcs)
16         return
17     for otherPcs ∈ inconsistencies do
18         if otherPcs ∈ baseCS then
19             removePcs(mergeCS, otherPcs)
20         else
21             hardPcsInconsistency(pcs,
                 otherPcs)
22 Function handleContent is
       Data: pcs: PCS, mergeCS, baseCS: ChangeSet
23     for tree ∈ {pcs.parent, pcs.pred, pcs.succ } do
24         removeSoftContentInconsistencies(tree,
             mergeCS, baseCS)
25 Function removeSoftContentInconsistencies
     is
       Data: tree: Tree, mergeCS, baseCS: ChangeSet
26     cts = getContentTuples(tree, mergeCS)
27     if size(cts) ≤ 1 then
28         return
29     nonBaseCts = {ct | ct ∈ cts, ct ∉ baseCS }
30     setContentTuples(tree, nonBaseCts,
         mergeCS)
31     if size(nonBaseCts) > 1 then
32         hardContentInconsistency(nonBaseCts)
```

Fig. 5: Pseudocode for SPORK-3DM

3.4 Building the AST and Handling Conflicts

When the change set has been merged, it must be converted back into an AST. This is achieved by traversing the PCS set and inserting visited nodes and their contents into an AST. Note that excess structural information is discarded when building the AST, such as the virtual root ($\perp$), start ($\dashv$) and end ($\vdash$), as well as the intermediate virtual nodes discussed

TABLE 5: The effect of consecutive invocations of the removeSoftPcsInconsistencies function in the core loop of the merge function starting from the raw merge from Table 4. For brevity, the table illustrates the invocations until all hard inconsistencies have been discovered and the final merge is achieved, as subsequent invocations have no effect.

pcs (input)	inconsistencies	removals	hard inconsistencies
$(01, \dashv, 02)$	$(01, \dashv, 12), (12, \dashv, 02), (12, 02, \vdash)$	$(01, \dashv, 02)$	-
$(12, \dashv, 02)$	$(01, 02, 03)$	$(01, 02, 03)$	-
$(01, 03, 15)$	$(01, 03, \vdash), (01, 03, 25)$	$(01, 03, \vdash)$	$(01, 03, 25)$
$(01, 15, \vdash)$	$(01, 25, \vdash)$	-	$(01, 25, \vdash)$

Algorithm 2: Helpers for spork-3dm

```
1  Function toChangeSet is
       Data: tree: Tree, classReps: Map[Tree, Tree]
       Result: ChangeSet
2      Convert tree to a change set with nodes mapped
         to their class representatives.
3  Function getAllInconsistentPcs is
       Data: pcs: PCS, cs: ChangeSet
       Result: List[PCS]
4      Get all parent, predecessor and successor
         inconsistencies for pcs in cs.
5  Function getContentTuples is
       Data: tree: Tree, changeSet: ChangeSet
       Result: Set[ContentTuple]
6      Get all content tuples related to the tree
         according to the change set.
7  Function setContentTuples is
       Data: tree: Tree, contents: Set[ContentTuple],
             changeSet: ChangeSet
8      Set the content tuples associated with the tree in
         the change set.
9  Function hardPcsInconsistency is
       Data: pcs: PCS, other: PCS
10     Register pcs and other as a hard inconsistency.
11 Function hardContentInconsistency is
       Data: contentTuples: Set[ContentTuple]
12     Register the provided content tuples as a hard
         inconsistency.
```

Fig. 6: Helper function definitions for SPORK-3DM

in Section 3.3.2.

Given a PCS, traversal left and right within the child list amounts to finding another PCS with the same parent and one matching child node, but where said child node's position is different. The child list of any node x starts with a PCS where x is the parent, and $\dashv$ is the predecessor. These traversal rules are summarized in Table 6. Also recall that the content of an arbitrary node v is represented by all content tuples $c(v, m)$.

In a child list without conflicts, there is always precisely one PCS matching each traversal pattern from some starting point, and each node of that PCS has precisely one content tuple. Consider again the merged PCS set in Table 4. A traversal always begins from the start of the virtual root's child list, which according to the traversal rules is $(\perp, \dashv, 01)$. It is a simple matter to traverse this child list to find that 01 is the concrete root of the tree, and similarly that the first two children of 01 are 12 and 03. Upon encountering each of these nodes in the traversal, their contents can be extracted from the merged content set of Table 7. However, the PCS set is clearly not consistent, as there are two PCS triples matching the right traversal pattern from $(01, 12, 03)$. This hard inconsistency indicates a conflict in the merge, the handling of which is described in Section 3.4.1.

TABLE 6: Traversal rules for the PCS structure, starting from an arbitrary initial PCS (x, y, z). Traversing in a given direction amounts to finding a PCS matching a specific pattern, where $?$ is an unknown node.

Direction	PCS pattern
Left	$(x, ?, y)$
Right	$(x, z, ?)$
Into y's child list	$(y, \dashv, ?)$
Into z's child list	$(z, \dashv, ?)$

TABLE 7: Content merge of the merge scenario of Figure 1

$c(01, "sum"), c(02, "a"), c(03, "b"), c(12, -), c(15, 1), c(25, "c")$

3.4.1 Insert/insert conflicts

An insert/insert conflict occurs when both revisions insert one or more nodes in the same position in the AST. The hard inconsistencies in Table 4 are caused by such a conflict, namely the insertion of $int_{15}(1)$ in the left revision and $ref_{25}(c)$ in the right revision (see Figure 1). The inconsistent elements show a typical pattern for an insert/insert conflict, namely that the conflict starts with the successor inconsistency between $(01, 03, 15)$ and $(01, 03, 25)$, and ends with the predecessor inconsistency between $(01, 15, \vdash)$ and $(01, 25, \vdash)$. This yields two possible paths from node 03 to the virtual end $\vdash$, either through node 15 or through node 25.

SPORK handles insert/insert conflicts by traversing both paths through the child list and collecting the nodes of both sides of the conflict. These are inserted into a conflict node, which in this case contains 15 from the left revision and 25 from the right revision. The full AST represented by the merge in Table 4 can then be built, resulting in the tree shown in Figure 7.

3.4.2 Delete/delete conflicts

A delete/delete conflict occurs when the left and right revisions delete adjacent nodes. For example, consider that the base revision has a node p with a child list ab, and that the left revision deletes b while the right revision deletes a. Omitting the parent node, the base revision's PCS child list is then $(\dashv, a), (a, b), (b, \vdash)$, the left revision's is $(\dashv, a), (a, \vdash)$, and the right revision's is $(\dashv, b), (b, \vdash)$. When merging these

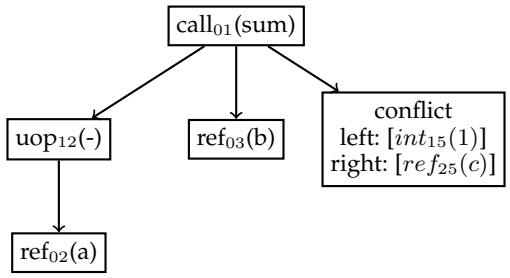

Fig. 7: AST built from the PCS merge in Table 4 and content set in Table 7

TABLE 8: The left, base and right revisions of a line with a delete/delete conflict, and the textual representations of the subtrees with conflicting child lists

	Left	Base	Right
Line	abs(sum(a))	abs(sum(a,b))	abs(sum(b))
Conflict subtree	sum(a)	sum(a,b)	sum(b)

child lists, the first and last elements are inconsistent across the left and right revisions. However, as $(\dashv, a)$ and $(b, \vdash)$ are both in the base revision, both inconsistencies are soft and can be eliminated, resulting in the consistent but clearly disjoint child list $(\dashv, b), (a, \vdash)$. This shows that the consistency criteria are not strong enough to guarantee that a consistent change set encodes a well-formed tree.

Disjoint child lists are trivial to detect, as they always result in one PCS with a successor (e.g. b in $pcs(p, \dashv, b)$) that never appears as a predecessor in the same child list. However, SPORK currently cannot recover all conflicting AST nodes due to the fact that parts of the conflict have already been removed from the change set. Instead, it falls back on a line-based merge of the textual representations of the subtrees rooted in parent node p. We refer to this as the *local fallback*. Table 8 shows an example merge scenario of a code snippet where the delete/delete conflict already discussed occurs in the argument list of a method call. The resulting merge conflict can be seen in Figure 8, along with the fully line-based merge for comparison. SPORK expresses conflicts in the same way that GIT's default merge tool does, with so-called *conflict hunks*. Each hunk starts with left-facing arrows ($<$) followed by the left revision's part and ends with right-facing arrows ($>$) preceded by the right revision's part, the two parts being demarcated by a line of equals signs ($=$). We use the terms conflict and conflict hunk interchangeably.

Note that the lines merged by the local fallback are not necessarily the exact lines of the original source files, but the textual representation of the conflicting subtrees. Thus, although less granular than structured conflict handling, the local fallback is still significantly more granular than a line-based merge of the entire file. A conflict node is then inserted into the AST containing the line-based merge, which is printed as-is during pretty-printing.

3.4.3 Insert/delete conflicts

An insert/delete conflict occurs when one revision inserts a node where another revision deletes a node. For example,

```
abs(
<<<<<<< left
sum(a)
=======
sum(b)
>>>>>>> right
)
```

(a) Structured merge with local fallback

```
<<<<<<< left
abs(sum(a))
=======
abs(sum(b))
>>>>>>> right
```

(b) Fully line-based merge

Fig. 8: Merge produced by SPORK's local fallback activating on the merge in Table 8 and the merge of a traditional line-based merge tool

assume that a node p has a child list a in the base revision, that the left revision deletes a, and the right revision inserts b after a. Omitting p, the base child list is then $(\dashv, a), (a, \vdash)$, the left is $(\dashv, \vdash)$ and the right is $(\dashv, a), (a, b), (b, \vdash)$. Removing all soft inconsistencies from the raw merge results in the child list $(\dashv, \vdash), (a, b), (b, \vdash)$. The first and last elements are in a hard predecessor inconsistency, and the middle element is disjoint from the start of the child list.

As with the delete/delete conflict discussed in Section 3.4.2, it is difficult to retrieve the left and right sides of the conflict due to the fact that part of the conflict is not present in the final change set. SPORK therefore resorts to the local fallback described in Section 3.4.2 when discovering an insert/delete conflict.

3.4.4 Move conflicts

Move conflicts are theoretically and practically troublesome. Intuitively, a move conflict occurs when the left and right revisions both manipulate the same node or the context around it such that the node ends up in two different positions in the computed merge. A pure move conflict involves the left and right revisions moving the same node to two different locations. Moves may also conflict with insertions (move/insert) or deletions (move/delete) at the source or destination sites. For example, assume that a node p has a child list a in the base revision, that the left revision moves a to another child list, and the right revision inserts b after a. This is a move/insert conflict, and in fact causes the child list of p to take exactly the same form as with the insert/delete conflict discussed in Section 3.4.3. The difference in this case is that a also exists in some other child list.

There are two high-level variations of move conflicts that present differing difficulties to handle. An *inter-parent* move conflict occurs when a node v is involved in a hard parent inconsistency, that is to say, has two different parents. Parent inconsistencies are easy to detect, and can only be caused by move conflicts, such as the one described above where a ends up in two child lists. SPORK handles such conflicts by recursively classmapping all nodes in the subtrees rooted in v to themselves, and then restarting the merge. As the nodes in the revisions of v are no longer classmapped to each other, they are considered different nodes by SPORK-3DM, which effectively turns moves into insertions and deletions. This may resolve the conflict entirely, or result in non-move conflicts that are easier to deal with.

An *intra-parent* move conflict occurs when a node v appears in two places in the same child list. These conflicts

are hard to identify as move conflicts as there is no single feature that distinguishes the resulting inconsistencies from those caused by other conflicts, delete conflicts in particular. Therefore, determining which nodes to classmap to themselves is difficult, and so SPORK resorts to the local fallback instead.

3.4.5 Conflict handlers

In some cases, conflicts can be automatically resolved. In the case of structural conflicts where SPORK successfully extracts all AST nodes that take part in a conflict, SPORK invokes two *structural conflict handlers*. The first of these resolves conflicts involving ambiguous ordering of method declarations[10], by inserting them in sorted order. The second handler resolves conflicts where one of the conflicting sides is empty by picking the non-empty side, optimistically: if a handler can resolve the conflict, the conflict node is replaced by a concrete node.

SPORK also defines *content conflict handlers*. When building the AST and attempting to set the content of a node for which multiple content tuples are found, SPORK invokes the content conflict handlers one by one until the conflict is resolved, or there are no more conflict handlers. Most content conflict handlers are highly dependent on SPOON implementation details, and therefore fall outside of the scope of this paper, and we refer the reader to the implementation for details[11].

3.4.6 Duplicated Type Member Elimination

The built SPOON AST is then subjected to *duplicated type member elimination*. In cases where both the left and right revisions add non-identical versions of the same type member, a failure to match these results in type member duplication in the merge process. For example, a class may end up with two methods with the same signature, or two fields with the same name, making for semantic conflicts. As 3DM-MERGE knows nothing of the semantics of JAVA, a duplicated type member will not seem problematic to it. To address such issues, SPORK searches the merged SPOON AST for duplicated type members, and re-executes the entire merge process for any pair it can find, using an empty node as the base revision[12]. The reason why duplicated type member elimination is performed at such a late stage is a matter of implementation convenience; it is trivial to find duplicated type members in the merged SPOON tree, while doing so in any of the earlier stages is much harder.

3.5 Pretty-printing

SPORK uses SPOON's default pretty-printer to produce the final result of the merge, namely a JAVA source file. There are however two significant extensions to the printer in SPORK.

The first extension is reusing the original source code for subtrees that originate from a single revision, in effect performing a copy-paste of a subtree's original source code. We refer to this as *high-fidelity pretty-printing*, and it allows SPORK to retain most of the formatting from the file revisions. High-fidelity pretty-printing of a merged AST is currently limited to type members and comments due to complications at more granular levels when adjacent elements stem from different revisions. For example, SPORK can directly reuse the source code of a method declaration that it determines stems from a single revision. However, if SPORK finds that a method declaration is composed of elements from multiple revisions, it currently cannot perform high-fidelity pretty-printing of individual child elements of that method, such as method parameters and statements. Either the entire method is high-fidelity pretty-printed, or none of it is. For printing of more granular elements whose parent elements cannot be printed with high-fidelity pretty-printing, SPORK uses SPOON's default pretty-printer, only taking the original indentation into account. All other formatting is fixed for each type of AST node. For example, the last method parameter in a parameter list is always immediately followed by a closing parenthesis, while all non-last parameters are followed by a comma and a space. We refer to this as *low-fidelity pretty-printing*.

```
sum(-a, b,
<<<<<<< left
1
=======
b
>>>>>>> right
)
```

Fig. 9: Pretty-printed output of the AST in Figure 7

The second extension is printing of conflicts. SPORK uses high-fidelity pretty-printing to print both sides of a structural conflict, or directly prints the contents of a conflict node produced by the local fallback (see Section 3.4.2). Note that the limitation of what elements can be printed with high-fidelity pretty-printing do not apply here, as the nodes of any given side of a conflict by definition stem from the same revision. The pretty-printed output of the running example is shown in Figure 9, containing such a conflict.

4 Experiment Methodology

This section presents the methodology we use for the evaluation of SPORK. We compare SPORK against JDIME [28], a state-of-the-art structured merge tool for JAVA, and AUTOMERGEPTM [24], a merge tool that builds upon JDIME with an enhanced tree matching algorithm. We assess SPORK in regards to conflicts, running time and formatting preservation.

4.1 State-of-the-art of Structured Merge for Java

We select the state-of-the-art tools for structured merge in Java as follows. First, the state-of-the-art tool does fully-structured merge on ASTs, which allows for an apples-to-apples comparison with SPORK. Second, there is a publicly available code base, which is crucial for reproducibility. Third, it works on real-world and non-trivial merge scenarios.

We have evaluated the tools in Table 9 according to these criteria. Only two tools fulfill all criteria: JDIME [28]

10. So-called *ordering conflicts*
11. https://github.com/KTH/spork
12. This effectively makes it a two-way merge of the type members

TABLE 9: Potential Merge Tools for Evaluation

Tool	Key features	Comments
JDIME [28][1]	Structured, AST-based	Included
AUTOMERGEPTM [24][2]	Structured, built on JDIME with improved tree matching	Included
JFSTMERGE [22][3]	Semistructured	Excluded
INTELLIMERGE [25][4]	Semistructured, graph-based	Excluded
Envision IDE [32][5]	Line-based merge on textual AST	Excluded

[1] https://github.com/se-sic/jdime
[2] https://github.com/thufv/automerge-ptm
[3] https://github.com/guilhermejccavalcanti/jFSTMerge
[4] https://github.com/symbolk/intellimerge
[5] https://github.com/dimitar-asenov/Envision

and AUTOMERGEPTM [24]. JDIME is fully structured and built on top of the EXTENDJ compiler framework and is able to merge real-world merge scenarios. JDIME has been extensively used in subsequent research, incl. [4], [24], [29], [30], [31]. AUTOMERGEPTM is mostly the same tool as JDIME, but with an enhanced tree matching algorithm.

4.2 Research Questions

The evaluation is structured around the following research questions.

- **RQ1: How does SPORK compare to JDIME and AUTOMERGEPTM in terms of amounts of conflicts and amounts of conflicting lines?** Merge conflict prevalence is a key aspect of a merge tool. The reduction of the number of merge conflicts is a primary advantage of structured merge. Furthermore, conflict size is an indicator that developers use to estimate the conflict difficulty, increasing size being associated with increasing difficulty [33].
- **RQ2: How does SPORK compare to JDIME and AUTOMERGEPTM in terms of running time?** Running time is an important aspect for any software engineering tool that is used in an interactive computing environment. The user has to wait for the merge tool to run to completion before being able to proceed, meaning that all else being equal, a faster tool is preferable to a slower one.
- **RQ3: How does SPORK compare to JDIME and AUTOMERGEPTM in terms of preserving source code formatting?** Due to operating on an abstract representation of source code, a structured merge always concludes in a pretty-printing step. If the printer does not attempt to recreate the formatting of the input, it may fundamentally alter it [12], [13]. We argue that a merge tool should not alter formatting at all.

4.3 Dataset

We select projects for the experiments from the REAPER dataset of GITHUB repositories [34]. This dataset consists of 1.8 million GITHUB projects that are scored with respect to a variety of indicators of a well-engineered project. These indicators include the use of CI and unit tests, the amount of documentation and amount of core contributors. The REAPER dataset has been used in other merge studies [30], [35].

4.3.1 Filtering projects

We select projects in REAPER that use JAVA, are classified as well-engineered, have more than 50 stars and a minimum of 2 core contributors. A total of 1174 projects fulfill all of the criteria. We further filter out projects that are forks.

We perform a last filtering to find projects that build using MAVEN in our test environment. We look for a `pom.xml` file in the latest commit[13] of the default branch, indicating the use of MAVEN. If there is such a file, the project is cloned and built with MAVEN. If the build succeeds within at most 5 minutes, the project is added to a list of candidate projects. We use two of these projects for testing purposes during the development of SPORK, and we exclude them from the evaluation.

This selection process leads to a list of 359 candidate projects that fulfill all criteria. This list is available in the online appendix[14].

4.3.2 Filtering merge commits

Our experiments require the base commit of each merge scenario to be located. We use GIT-LOG to find merge commits, and GIT-MERGE-BASE to find the merge base. In cases where a commit history has a criss-cross pattern[15], there are multiple possible merge bases. The merge base is then said to be *ambiguous*. Merge commits with ambiguous merge bases are excluded from the dataset as they complicate merge replay.

As noted by Cavalcanti et al. [30], most merges have no overlap between the files edited by the left and right revisions, making the merge resolution trivial: simply pick the edited file. GIT only invokes a merge tool when the same file has been edited in both revisions. The merge commits are therefore filtered to include only commits where at least one JAVA source file is edited in both the left and right revisions.

We also filter out merge commits for which at least one of the revisions do not build using MAVEN. If the project builds, we can be certain that it is syntactically valid, which is important as syntactically invalid files can cause unexpected behavior in structured merge tools. Finally, as some projects have thousands of merge commits, while others have as little as 1, we limit the amount of merge commits per project to 15 to avoid the larger projects being overrepresented.

We extract a total of 890 real-world merge scenarios from 119 different projects, consisting of a total of 1740 file merges. We observe a great deal of variety in project domains, such as the MAGE game engine, the CORENLP natural language processing library, the ASSERTJ assertions library, the SINGULARITY platform-as-a-service and the CHRONICLE-MAP in-memory database. The popularity and sizes of projects also vary greatly. Project sizes range from 1106 to 1782052 lines of code, with a median of 24306. The amount of GitHub stars ranges from 58 to 21720, with

13. As of the 10th of May 2020
14. https://github.com/slarse/spork-experiments
15. https://git-scm.com/docs/git-merge-base

a median of 341. The amount of core contributors ranges from 1 to 40, with a median of 6. Finally, the amount of merge scenarios extracted from each project ranges from 1 to 15, with a median of 7, and the amount of file merges ranges from 1 to 122, with a median of 12. Note that the project metadata was collected on August 12th 2020 while the REAPER dataset is from 2017, meaning that there are some discrepancies between the two. The full list of project statistics is available in the online appendix[16].

4.4 Experiment Protocol

We design an experiment protocol similar to that of Shen et al. [25]. For each merge scenario, the individual file merges are extracted. This involves finding all revisions of a merged file, including the one actually committed by the developer, which we refer to as the *expected* revision of the file merge. We use GIT's merge functionality to locate the revisions.

SPORK and JDIME are then applied in turn to the base, left and right revisions of each file merge. We refer to the merged file produced by a merge tool as the *replayed* revision for that file merge and merge tool.

In order to assess RQ1, we scan each replayed revision for conflict hunks[17], and record the amount of hunks as well as the total amount of lines contained in them. In general, it is easier for developers to deal with conflicts if they are as few and small as possible [5], [36], [37], meaning that minimizing conflict quantities and sizes is desirable. To make sure that we only analyze merge conflicts produced by the tools under test, any file merges in which the base, left or right revisions contain conflict markers are excluded from the experiment. For all comparisons, file merges where at least one tool fails to produce a non-empty merged file are excluded. This also applies to subsequent research questions.

To address RQ2, the execution of each file merge is timed 10x per file file, with the running time measured as the wall time from the moment of invoking the merge tool to the moment it exits. Each execution is a cold start, meaning that the JVM is not allowed to warm up. A timeout is set to 300 seconds per file merge, after which the merge is forcibly aborted.

To address RQ3, we measure formatting preservation with the expected revision as the ground truth for correctly formatted output. In order to determine how closely the replayed revision resembles the expected revision, we compare them with two metrics: a *line diff* computed with GIT[18], and a *character diff* computed with the PYTHON standard library module DIFFLIB [19]. We refer to the sum of insertions and deletions of lines and characters as the *line diff size* and *character diff size*, respectively. Poor formatting preservation increases the diff size.

For all RQs, we illustrate the behavior of SPORK with case studies. Those case studies are real merge scenarios taken from our dataset. They are selected manually, with the goal of highlighting advantages and drawbacks of SPORK.

16. https://github.com/slarse/spork-experiments
17. Recall the definition of conflict hunks from Section 3.4.2
18. https://git-scm.com/docs/git-diff
19. https://docs.python.org/3.8/library/difflib.html

4.5 Statistical Tests

All measurements provide us with paired ordinal data (conflict sizes, conflict quantities, diff sizes and running times) for each merge scenario (one measure for JDIME, one for SPORK, and one for AUTOMERGEPTM). As the tools might fail, we exclude all measures from merge scenarios where at least one tool fails to produce a merge.

To statistically assess the differences between the tools, we first start by a Friedman test with the null hypothesis that the measures for all tools are the same. In case of a significant p-value on the significance level of $\alpha = 0.05$, the null hypothesis is rejected and we then proceed to two post-hoc tests to compare the groups SPORK vs JDIME and SPORK vs AUTOMERGEPTM. We use a two-sided Wilcoxon signed-rank test, along with the matched-pairs rank-biserial correlation (RBC) as effect size [38], [39] for each post-hoc test. The resulting p-values are then corrected using a Holm-Bonferroni correction. Finally, we assess their significance using $\alpha = 0.05$. We use the implementation provided by the PYTHON package PINGOUIN[20] version 0.4.0 to perform the tests and calculate effect sizes. In our results, a negative RBC indicates that SPORK's values in the given test are smaller than JDIME's or AUTOMERGEPTM, while a positive RBC indicates the opposite.

4.6 Experiment Environment

The test environment hardware consists of a Ryzen 5900X, 32 GiB of RAM @3600 MHz and a SATA SSD with read and write speeds of 500 MB/s. The test environment runs ARCHLINUX with kernel 5.13.9, OPENJDK 1.8.0u292 and CPYTHON 3.8.2. We build JDIME from source using commit 100aeece. We build AUTOMERGEPTM from source using commit e73038b5. We use SPORK release v0.5.1. For further information, we refer the reader to the online appendix[21].

5 Experiment Results

This section presents the results from the experiments. Section 5.1 presents results on sizes and quantities of conflicts, Section 5.2 presents results on running times and Section 5.3 presents results on formatting preservation.

5.1 RQ1: Quantity and Size of Conflicts

5.1.1 Amount of conflict hunks

We first measure and compare the amount of conflict hunks per file. As noted in Section 4.4, fewer and smaller conflicts is generally better. However, there are cases where fewer or smaller conflicts are due to a poor merge, such as when truly conflicting edits are not detected as such, or when a conflict is not intuitively represented. This is discussed in the illustrative case studies below.

As noted in Section 4.4, we filter out file merges where at least one merge tool fails to produce a non-empty file merge, as it is then not possible to fairly compare the results. There are three separate cases that can occur: the merge tool can crash, exceed the time limit of 300 seconds or produce an empty merged file. An empty or non-existing file cannot be

20. https://pingouin-stats.org/
21. https://github.com/slarse/spork-experiments

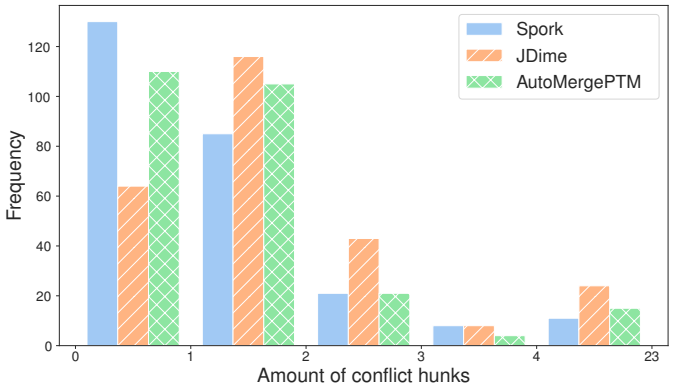

Fig. 10: RQ1: Histogram of conflict hunk quantities per file for SPORK, JDIME and AUTOMERGEPTM. Lower is better. Each histogram bin contains the frequency of values in the range $[L, R)$, where L and R are the values to the left and right of the bin, respectively.

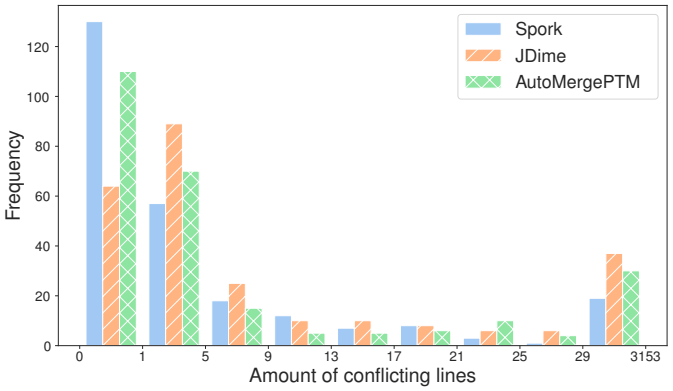

Fig. 11: Histogram of conflict sizes for SPORK, JDIME and AUTOMERGEPTM per file merge. Lower is better. Each histogram bin contains the frequency of values in the range $[L, R)$, where L and R are the values to the left and right of the bin, respectively.

Tool	Timeouts	Crashes	Empty file	Total
SPORK	0	34	0	34
JDIME	16	7	0	23
AUTOMERGEPTM	7	11	19	37

TABLE 10: Summary of merge failures.

used for making comparisons of our chosen metrics, and must therefore be excluded. Table 10 shows a breakdown of merge failures across the tools. SPORK has the most amount of crashes, but exhibits none of the other kinds of failures. JDIME has the smallest amount of failures in total, but it also exhibits the largest amount of timeouts. AUTOMERGEPTM has the largest amount of failures in total. It is the only tool to occasionally produce an empty merged file, accounting for most of its failures, but it also exhibits both the other kinds of failures. There is little overlap between the file merges where the tools fail, with the 94 merge failures occurring across 83 unique file merges, constituting 4.77% of the total of 1740 file merges.

Out of the 1740 file merges in the benchmark, there are 255 file merges for which all of SPORK, JDIME and AUTOMERGEPTM produce a non-empty merge file, and at least one tool encounters a conflict. SPORK signals conflicts in 125 of the merges and produces a total of 227 conflict hunks. JDIME signals conflicts in 191 of the merges and produces a total of 376 conflict hunks. AUTOMERGEPTM signals conflicts in 145 of the merges and produces a total of 245 conflict hunks. Overall, SPORK produces 151 conflict hunks fewer than JDIME (40% reduction), and 18 fewer than AUTOMERGEPTM (7% reduction).

Figure 10 shows the histogram of the distribution of conflict hunk quantities for SPORK, JDIME and AUTOMERGEPTM. While JDIME is clearly at a disadvantage (more files with 2 or more conflict hunks), the distributions for SPORK and AUTOMERGEPTM look largely similar.

We use a Friedman test to determine if further analysis of the results is relevant, with the null hypothesis that the results from the different tools are the same. The test yields a p-value of 2.40e-13, so we reject the null hypothesis and proceed with further analyses.

We use a two-sided Wilcoxon signed-rank test to test the following hypothesis:

H_0^1: SPORK and JDIME produce the same amounts of conflict hunks
H_a^1: SPORK and JDIME do not produce the same amounts of conflicts

The test yields a p-value of 7.98e-8, and we therefore accept the alternative hypothesis that the tools produce differing amounts of conflict hunks. The RBC is -0.423, indicating that SPORK produces fewer hunks than JDIME.

We use a two-sided Wilcoxon signed-rank test to test the following hypothesis:

H_0^2: SPORK and AUTOMERGEPTM produce the same amounts of conflict hunks
H_a^2: SPORK and AUTOMERGEPTM do not produce the same amounts of conflicts

The test yields a p-value of 0.269, so we cannot reject the null hypothesis that the tools produce the same amounts of conflicts.

5.1.2 *Amount of conflicting lines*

We now consider the amount of conflicting lines per file merge, which we refer to as the *conflict size*. We consider here the same 255 file merges as in Section 5.1.1; file merges where all tools produce a non-empty merged file and at least one tool produces a conflict hunk. We measure the conflict size of a file as the sum of all lines in all conflict hunks (see Section 3.4.2), which is a proxy to the effort spent by developers to resolve conflicts [5], [36], [37].

Figure 11 shows a histogram of conflict sizes for SPORK, JDIME and AUTOMERGEPTM. The first bin refers to cases where the merge is fully successful, containing no conflict[22] In this bin, SPORK outperforms both JDIME and AUTOMERGEPTM. Looking at the rightmost bin of the figure, JDIME and AUTOMERGEPTM produce more files with large

22. Note that it is identical to that of the conflict quantity histogram in Figure 10.

conflict sizes. For all tools, there are outliers, meaning that a small amount of conflicts account for the majority of conflicting lines. This is the primary explanation of SPORK's improvement: AUTOMERGEPTM and JDIME produce more abnormally large conflicts than SPORK. In particular, SPORK produces files with conflict sizes at or above 20 lines of code in 24 merges, making for a reduction by 54% compared to JDIME's 52 cases, and by 47% compared to AUTOMERGEPTM's 45 cases. In the middle of the distribution, the interpretation is not clear-cut, but accounts for significantly fewer data points than the extrema.

In the case of JDIME and AUTOMERGEPTM, abnormally large conflicts are often caused by failures to match renamed elements to each other, which is exemplified with a method rename in case study *C4* discussed below. Regarding SPORK's large conflicts, they are often caused by the local-fallback activating on the body of a class, causing most of the file to be merged with a line-based merge.

Let us now aggregate these results. Over the 255 file merges, SPORK produces a total of 2446 conflicting lines, JDIME produces a total of 13975 conflicting lines, and AUTOMERGEPTM produces a total of 6635 conflicting lines. SPORK improves upon AUTOMERGEPTM, second best by this metric, by 63%.

We use a Friedman test to determine if further analysis is necessary, with the null hypothesis that the results from the different tools are the same. The test yields a p-value of 2.23e-13, so we reject the null hypothesis and proceed with further analyses.

We use a two-sided Wilcoxon signed-rank test to test the following hypothesis:

H_0^3: SPORK and JDIME produce equal amounts of conflicting lines

H_a^3: SPORK and JDIME do not produce equal amounts of conflicting lines

The test yields a p-value of 4.47e-4, and we therefore accept the alternative hypothesis that the tools do not produce equally large conflicts. The effect size RBC is -0.280, indicating that SPORK produces fewer conflicting lines than JDIME.

We also use a two-sided Wilcoxon signed-rank test to test the following hypothesis:

H_0^4: SPORK and AUTOMERGEPTM produce equal amounts of conflicting lines

H_a^4: SPORK and AUTOMERGEPTM do not produce equal amounts of conflicting lines

The test yields a p-value of 0.441, so we cannot reject the null hypothesis that the tools produce equal amounts of conflicting lines.

Conflict case studies

It is important to note that conflict quantities and sizes alone do not fully describe the conflict behavior of a merge tool. While in general, merge tools should strive for as few and as small conflicts as possible, the presence of a conflict is positive when there is no best conflict handling decision to be made. Similarly, a smaller conflict is not always easier to interpret, as it may be small by virtue of failing to include relevant information. We now illustrate this important point with examples[23]. Each case study is provided with an identifier on the form *Cx*, where *x* is an integer. This identifier can be used to find the complete file merge along with all metadata in our online appendix[24].

As a first concrete example, consider the merge of the file header comment in Figure 12, stemming from file merge *C1*. SPORK correctly produces a conflict as the changes across revisions are incompatible, while both JDIME and AUTOMERGEPTM simply discard the file header comment, producing no conflict. In this case, the presence of a conflict is good, and SPORK produces the most informative output for the developer.

The opposite is also prominent, i.e. that some non-conflicting edits are detected as conflicts. Figure 13 shows

23. Note that the presence of ... in a source code snippet indicates that it has been truncated to fit the paper format.

24. https://github.com/slarse/spork-experiments

```
/**
 * Copyright 2009-2019 ...
 *
```

(a) Left revision

```
/**
 * Copyright 2009-2016 ...
 *
```

(b) Base revision

```
/**
 * Copyright 2009-2020 ...
 *
```

(c) Right revision

```
/**
<<<<<<< LEFT
 *     Copyright 2009-2019 ...
=======
 *     Copyright 2009-2020 ...
>>>>>>> RIGHT
 *
```

(d) SPORK's merge

Fig. 12: The left, base and right revisions of the license header from file merge *C1*, along with SPORK's merge. JDIME and AUTOMERGEPTM do not merge comments, and discard file headers completely.

```
<<<<<<< LEFT
!transport.isSuccessful()) parseAndThrowException(result);
=======
!transport.isSuccessful()) parseAndThrowException(result,
    jobInfo.getContentType());
>>>>>>> RIGHT
```

(a) Conflict from SPORK's merge of *C2* caused by too conservative left/right matching. JDIME and AUTOMERGEPTM produce the right revision's contribution as the merged output.

```
<<<<<<< LEFT
long idleThreadKeepAliveMillis = 60000;
=======
private static final String DEFAULT_EXCHANGE_NAME = "";
>>>>>>> RIGHT
```

(b) Conflict between two unrelated and textually far removed fields from JDIME's/AUTOMERGEPTM's merge of *C3*, caused by too aggressive left/right matching. SPORK correctly adds both fields at their respective points of insertion.

Fig. 13: Snippets showing drawbacks of too conservative and too aggressive left/right matchings

```
    @Test
    public void
      testNonNullNativeIgnoreingDocumentationParameterMatcher() {
      context.checking(new Expectations() {{
-       exactly(1).of (mock).withBoolean(with(any(Boolean.class)));
-       exactly(1).of (mock).withByte(with(any(Byte.class)));
...
+       exactly(1).of(mock).withBoolean(with.booleanIs(anything()));
+       exactly(1).of(mock).withByte(with.byteIs(anything()));
...
    }});
```

(a) Base/left line-based diff. Lines preceded by – and + indicate deletions and additions, respectively.

```
    @Test
-   public void testNonNullNativeIgnoreing...
+   public void testNonNullNativeIgnoring...
```

(b) Base/right line-based diff. Lines preceded by – and + indicate deletions and additions, respectively.

```
<<<<<<< LEFT
...
 @Test public void testNonNullNativeIgnoreingDocumentationParameterMatcher() {
  context.checking(new Expectations() {
   {
    exactly(1).of(mock).withBoolean(with.booleanIs(anything()));
...
  }
=======
>>>>>>> RIGHT

...
 @Test public void testNonNullNativeIgnoringDocumentationParameterMatcher() {
  context.checking(new Expectations() {
   {
    exactly(1).of(mock).withBoolean(with(any(Boolean.class)));
...
  }
```

(c) JDIME's/AUTOMERGEPTM's merge, with the left revision's version of the method in a conflict, followed by the right revision's version of the method outside the conflict hunk

Fig. 14: Line-based base/left and base/right diffs from file merge *C4*. The left revision edits the body of a test method, and the right revision fixes a typo in the method's name. JDIME and AUTOMERGEPTM do not detect the rename, and produce merge conflict. SPORK correctly merges the name change in the right revision with the body changes in the left, producing no conflict.

the effects of too conservative and too aggressive left/right matchings from file merges *C2* and *C3*, respectively. In Figure 13a, SPORK's conservative left/right matching causes it to fail to match near-identical subtrees inserted in the left and right revisions, thus producing a coarse conflict where the right revision's part is a strict superset of the left. This can be automatically resolved, and a reasonable resolution to the conflict is the right revision, which is what JDIME and AUTOMERGEPTM produce. However, too aggressive left/right matching also causes problems with conflicts. In Figure 13b, JDIME and AUTOMERGEPTM match two completely unrelated fields that are added some 100 lines away from each other in the left and right revisions, respectively, and therefore produce a nonsensical conflict. SPORK on the other hand inserts the fields appropriately, without conflict.

We now provide evidence of SPORK 's move and update detection capability being beneficial. Figure 14 shows parts of the base/left and base/right diffs from file merge *C4*, where the left revision edits the body of a test method, and the right revision renames said method. JDIME and AUTOMERGEPTM both treat the rename in the right revision as a deletion of the original method, and an insertion of an entirely new method. The deletion interferes with the edit in the method's body in the left revision This results in a delete/edit conflict containing the left revision's version of the method. As the right revision's renamed method is seen as an insertion, it is printed outside the conflict hunk. Thus, in failing to match the renamed method of the right revision to the edited method in the left revision, the merge conflict produced is not only unnecessary, but it also fails to include the right revision's version of the method in the conflict hunk. Thus, a smaller conflict hunk is not always easier to understand. SPORK on the other hand performs the merge without conflict, as it detects the right revision's rename as an update of the method node's content, which is unrelated to the left revisions edits in its subtree (see the description of SPORK-3DM in Section 3.3.3 for the separation of content and structure).

The takeaway of these illustrative case studies is that SPORK exhibits differing and desirable merge properties from JDIME and AUTOMERGEPTM. This experiment also recalls that conflict quantity and size are indicative but not perfect metrics [30], as there may be some degenerate cases. For example, when conflicts occur inside comments or formatting, a merge tool which does not support comments or formatting preservation may produce zero conflict while missing an essential part of the merge.

Answer to RQ1. SPORK produces fewer and smaller conflicts than JDIME, and is on par with AUTOMERGEPTM. All assessed merge tools sometimes produce abnormally large conflicts (Figure 11) but SPORK to a lesser extent.

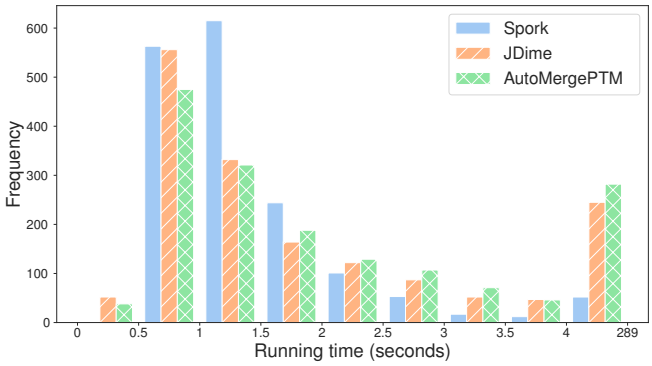

Fig. 15: Histogram of file merge running times for SPORK, JDIME and AUTOMERGEPTM. Lower is better. Each histogram bin contains the frequency of values in the range $[L, R)$, where L and R are the values to the left and right of the bin, respectively.

5.2 RQ2: Running Time

The running time of a tool on a given file merge is computed as the median wall time of 10 executions. We only consider the 1657 file merges where all of SPORK, JDIME and AUTOMERGEPTM produce a non-empty merged file. It is noteworthy that there are cases where JDIME and AUTOMERGEPTM fail due to timing out at 300 seconds. JDIME suffers 16 timeouts and AUTOMERGEPTM suffers 7, while SPORK exhibits no timeouts. The exclusion of these timeouts is conservative, as it clearly benefits JDIME and AUTOMERGEPTM.

In the median case, SPORK has a running time of 1.17 seconds, JDIME has a running time of 1.32 seconds and AUTOMERGEPTM has a running time of 1.48 seconds. Per this median value, SPORK is the fastest out of the three. SPORK being a faster tool is further reinforced by the sum of running times: SPORK's total running time is 2415 seconds, which is 51% faster than JDIME's 4912 seconds, and 55% faster than AUTOMERGEPTM's 5360 seconds. The histogram of running times in Figure 15 further exposes performance differences. JDIME and AUTOMERGEPTM have more of the smallest running times (leftmost bin), with 52 and 38 running times respectively that are less than 0.5 seconds, while SPORK has none. In terms of the largest running times (rightmost bin), JDIME and AUTOMERGEPTM have 245 and 282 running times respectively that are larger than or equal to 4 seconds, whereas SPORK only has 52. Furthermore, SPORK's maximum running time is 11.9 seconds, while JDIME and AUTOMERGEPTM top out at 287.9 and 287.7 seconds, respectively. Compared to JDIME, spork is faster in 963 cases and slower in the remaining 694 cases. Compared to AUTOMERGEPTM, SPORK is faster in 1126 cases and slower in the remaining 531 cases. While SPORK is not as fast as either JDIME or AUTOMERGEPTM in the best case, it is faster in the median case, and significantly reduces the amount and magnitudes of excessive running times larger than 4 seconds.

We use a Friedman test to determine if further analysis of the results is relevant, with the null hypothesis that the results from the different tools are the same. The test yields a p-value of 7.88e-247, so we reject the null hypothesis and proceed with further analyses.

We use a two-sided Wilcoxon signed-rank test to test the following hypothesis:

H_0^5: There is no difference between SPORK's and JDIME's running times
H_a^5: There is a difference between SPORK's and JDIME's running times

The test yields a p-value of 1.80e-54, and we therefore accept the alternative hypothesis. The effect size RBC is -0.441, indicating that SPORK's running times are smaller than JDIME's.

We use a two-sided Wilcoxon signed-rank test to test the following hypothesis:

H_0^6: There is no difference between SPORK's and AUTOMERGEPTM's running times
H_a^6: There is a difference between SPORK's and AUTOMERGEPTM's running times

The test yields a p-value of 1.74e-121, and we therefore accept the alternative hypothesis. The effect size RBC is -0.666, indicating that SPORK's running times are smaller than AUTOMERGEPTM's.

Answer to RQ2. SPORK is a faster merge tool than the state of the art. In particular, it has fewer exceedingly long running times which makes it more useful in practice for the developer.

5.3 RQ3: Formatting Preservation

Formatting preservation is measured as the diff size (the sum of insertions and deletions in a diff) between the replayed merge produced by the merge tool and the expected revision committed by the developer, considered as ground truth. We use two metrics at different levels of granularity: a line diff as well as a character diff. The results can be interpreted as the amount of lines and the amount of characters by which the produced and expected revisions differ. We consider the 1402 file merges in which all of SPORK, JDIME and AUTOMERGEPTM produce conflict-free merges.

SPORK produces file merges with a median line diff size of 65, which represents a 78% reduction compared to JDIME's median of 308.5, and a 79% reduction compared to AUTOMERGEPTM's median of 314.5. This clearly shows that SPORK preserves more formatting than the other tools. The histogram in Figure 16 shows SPORK's clear advantage over JDIME and AUTOMERGEPTM. Compared to JDIME, SPORK produces smaller line diff sizes for 1336 cases, of equal size in 3 cases, and larger ones in the remaining 63. Compared to AUTOMERGEPTM, SPORK produces smaller line diff sizes in 1341 cases, of equal size in 3 cases, and larger ones in the remaining 58.

The trend set in the line diff comparison carries over to the character diff measurements. SPORK produces file merges with a median character diff size of 528, which represents a 75% reduction compared to JDIME's median of 2181, and a 78% reduction compared to AUTOMERGEPTM's median of 2430. The histogram in Figure 17 shows SPORK's clear advantage over JDIME and AUTOMERGEPTM. Compared to JDIME, SPORK produces smaller character diff sizes

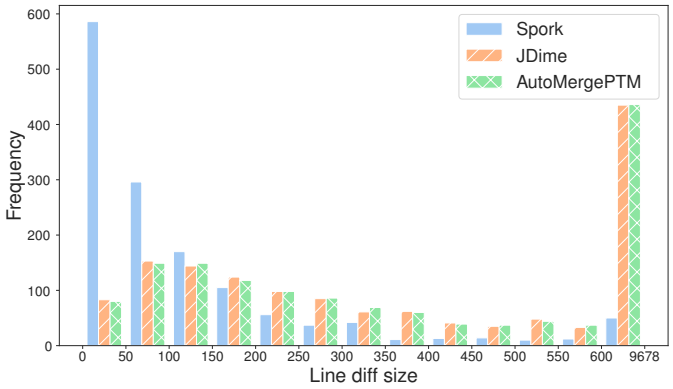

Fig. 16: Histogram of line diff sizes for SPORK, JDIME and AUTOMERGEPTM. Lower is better. Each histogram bin contains the frequency of values in the range $[L, R)$, where L and R are the values to the left and right of the bin, respectively.

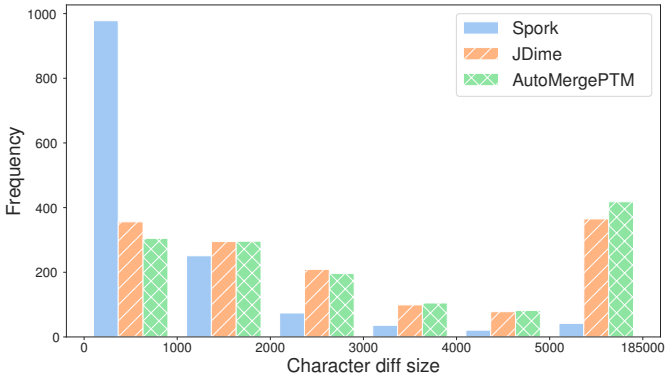

Fig. 17: Histogram of character diff sizes for SPORK, JDIME and AUTOMERGEPTM. Lower is better. Each histogram bin contains the frequency of values in the range $[L, R)$, where L and R are the values to the left and right of the bin, respectively.

in 1286 cases, and larger ones in the remaining 116. Compared to AUTOMERGEPTM, SPORK produces smaller character diff sizes in 1301 cases, and larger ones in the remaining 101. These numbers correspond well with the line-based line diff, indicating that it is a good approximation for the overall textual similarity of two files.

To illustrate SPORK's improvements, we present a final case study. Figure 18 shows a complex conditional expression from file merge *C5*. The condition of the if-statement is complex both with respect to the number of clauses and with respect to formatting (lots of ad hoc indentation and line breaks). Through high-fidelity pretty-printing of the method containing this if-statement, SPORK precisely reproduces said formatting. In contrast, JDIME's pretty-printer both changes the indentation from 4 spaces to 2, and collapses the entire first condition into a single line of 280 characters, completely ruining readability. This also applies to AUTOMERGEPTM by virtue of using JDIME's pretty-printer.

Our manual analysis confirms that the small diff sizes for SPORK's merges can be attributed to the SPORK's high-

```
if (parentContext != null
    && parentContext.object != null
    && ("java.util.ArrayList".equals(parentName)
     || "java.util.List".equals(parentName)
     || "java.util.Collection".equals(parentName)
     || "java.util.Map".equals(parentName)
     || "java.util.HashMap".equals(parentName))) {
  parentName = parentContext.object.getClass().
    getName();
  if (parentName.equals(parentClassName)) {
    param = parentContext.object;
  }
}
```

(a) SPORK's output, identical to the developer merge

```
if (parentContext != null && parentContenxt...
 parentName = parentContext.object.getClass().
    getName();
 if (parentName.equals(parentClassName)) {
  param = parentContext.object;
 }
}
```

(b) JDIME's/AUTOMERGEPTM's output. The entire condition has been written out on a single 280 characters long line (note truncation: ...), which would not be acceptable for the developer.

Fig. 18: Comparison between SPORK' and JDIME's formatting preservation on part of file merge *C5*

fidelity pretty-printing that preserves the original indentation, style and formatting, as explained in Section 3.5. SPORK is able to copy the original source code of certain elements involved in a merge and print it as-is into the output file. This is in contrast to JDIME and AUTOMERGEPTM, which only perform low-fidelity pretty-printing with its own formatting style.

We use a Friedman test to determine if further analysis of the line diff sizes is relevant, with the null hypothesis that the results from the different tools are the same. The test yields a p-value of 0 with machine precision, so we reject the null hypothesis and proceed with further analyses.

We use a two-sided Wilcoxon signed-rank test to test the following hypothesis:

H_0^7: There is no difference between the line diff sizes of file merges produced by SPORK and JDIME
H_a^7: There is a difference between the line diff sizes of file merges produced by SPORK and JDIME

The test yields a p-value of 1.85e-213, and we therefore accept the alternative hypothesis that there is a difference between the line diff sizes of merges produced by the tools. The RBC is -0.963, indicating that SPORK produces merges with lesser line diff sizes than JDIME.

We use a two-sided Wilcoxon signed-rank test to test the following hypothesis:

H_0^8: There is no difference between the line diff sizes of file merges produced by SPORK and AUTOMERGEPTM
H_a^8: There is a difference between the line diff sizes of file merges produced by SPORK and AUTOMERGEPTM

The test yields a p-value of 1.85e-213, and we therefore accept the alternative hypothesis that there is a difference

between the line diff sizes of merges produced by the tools. The RBC is -0.963, indicating that SPORK produces merges with lesser line diff sizes than AUTOMERGEPTM.

We use a Friedman test to determine if further analysis of the character diff sizes is relevant, with the null hypothesis that the results from the different tools are the same. The test yields a p-value of 0 with machine precision, so we reject the null hypothesis and proceed with further analyses.

We use a two-sided Wilcoxon signed-rank test to test the following hypothesis:

H_0^9: There is no difference between the character diff sizes of file merges produced by SPORK and JDIME

H_a^9: There is a difference between the character diff sizes of file merges produced by SPORK and JDIME

The test yields a p-value of 1.29e-199, and we therefore accept the alternative hypothesis that there is a difference between the character diff sizes of merges produced by the tools. The RBC is -0.929, indicating that SPORK produces merges with lesser character diff than JDIME.

We use a two-sided Wilcoxon signed-rank test to test the following hypothesis:

H_0^10: There is no difference between the character diff of file merges produced by SPORK and AUTOMERGEPTM

H_a^10: There is a difference between the character diff of file merges produced by SPORK and AUTOMERGEPTM

The test yields a p-value of 2.26e-205, and we therefore accept the alternative hypothesis that there is a difference between the character sizes of merges produced by the tools. The RBC is -0.944, indicating that SPORK produces merges with lesser line diff sizes than AUTOMERGEPTM.

Answer to RQ3. SPORK preserves formatting to a greater extent than JDIME and AUTOMERGEPTM. SPORK produces smaller line and character diffs in more than 90% of cases with median diff size reductions of 75% and above.

5.4 Recapitulation

In our experiments, we have answered three research questions targeting different facets of structured merge: conflicts, running times and formatting preservation. We have systematically and quantitatively compared our contribution, SPORK, against the relevant state-of-the-art, JDIME and AUTOMERGEPTM. We summarize the quantitative results in Table 11. Regarding conflicts (RQ1), SPORK performs better than JDIME and on par with AUTOMERGEPTM. Regarding running times (RQ2), SPORK is slightly faster in the median case, but more importantly reduces both amounts and magnitudes of excessive running times. Regarding formatting preservation, which is our main contribution, SPORK decreases the formatting changes by an order of magnitude. According to this evaluation, SPORK can be considered to be pushing the state of the art of software merging.

6 Discussion

The results of our experiments indicate that SPORK performs well overall. In this section, we discuss the limitations we identified, as well as the threats to the validity of our experiment.

		SPORK	JDIME	APTM
RQ1: Conflicts	# considered merges	255	255	255
	# files with conflicts	125	191	145
	# conflict hunks	227	376	245
	# conflicting lines (total)	2446	13975	6635
	# conflict sizes $\geq$ 20 LOC	24	52	45
RQ2: Running times	# considered merges	1667	1667	1667
	median running time	1.18s	1.32s	1.48s
	total running time	2435s	4937s	5388s
	# running times $<$ 0.5s	0	52	38
	# running times $\geq$ 4s	53	248	285
RQ3: Formatting	# considered merges	1402	1402	1402
	median line diff size	65	308.5	314.5
	median char diff size	528	2181	2430

TABLE 11: Summary of our quantitative results.

6.1 Limitations of SPORK

SPORK has two limitations when it comes to handling conflicts. The first one is the problem with move and delete conflicts, which are currently handled with textual representations of the subtrees involved. Move conflicts in particular are difficult to handle, and pose a problem that is introduced solely due to SPORK being move-enabled. While there are file merges in the results that SPORK can merge due to being move-enabled, such as method renaming, it is unclear whether the benefits outweigh the drawbacks. Therefore, a future study to compare move-enabled merge to non-move-enabled merge is called for.

The second conflict-related limitation is that SPORK ignores so-called delete/edit conflicts, which occur when one revision deletes a subtree where the other revision performs edits. In 3DM-MERGE, such a deletion silently overrides any edits in the subtree [15], and SPORK has no additional measure in place to detect such conflicts. This limitation is thus inherited from 3DM-MERGE. While detecting a delete/edit conflict in 3DM-MERGE is possible through post-processing of the change set [15], finding the correct way to represent it in the merged AST is less straightforward and requires non-trivial extensions of SPORK. Combined with the findings presented in the case studies in Section 5.1, more in-depth analysis of conflict behavior along the lines of those conducted by Cavalcanti et al. [30] and Tavares et al. [40] is therefore necessary to draw accurate conclusions about conflict handling.

Furthermore, there are limitations in SPORK's high-fidelity pretty-printing, which is a fundamentally hard problem [12]. While high-fidelity pretty-printing is one of SPORK's primary advantages over the other structured merge tools, it is not perfect. In the current implementation, high-fidelity pretty-printing is only enabled for type members and comments that stem from a single revision.

More granular elements are printed with low-fidelity pretty-printing. This often causes SPORK to alter formatting in undesirable ways, such as by printing redundant parentheses not present in the original source code [41], or by failing to reproduce ad-hoc indentation like JDIME does in Figure 18. To sum up, while SPORK greatly improves over the related work with respect to formatting and readability of merges, the difficulty of the problem calls for future research and engineering about formatting preservation.

SPORK also exhibited 34 crashes in the experiments, indicating unhandled corner cases. It should be noted that 15 of these errors were caused by parse errors in SPOON, and were thus completely outside of SPORK's control.

6.2 Threats to Validity

The primary threats to external validity are the diversity and representativeness of the dataset, as defined by Nagappan et al. [42]. The diversity of the dataset is a critical aspect enabling the results to generalize. Our dataset consists of open-source JAVA projects from the GITHUB platform, which means that the results do not necessarily generalize to other platforms or closed-source projects. Discarding projects and merge scenarios that failed to build with MAVEN also limits the diversity of the dataset, both by honing in on projects using MAVEN and by enforcing that the projects build.

A threat to representativeness is the fact that our methodology can only discover merge scenarios that are explicitly present in the commit history, which notably excludes merges that have been squashed or occurred during rebasing [1], [43]. Furthermore, as no project was allowed to contribute more than 15 merge scenarios, the dataset is not representative of the population of merge scenarios in terms of proportions. This is however necessary, as trial runs of the experiments without this restriction had a few of the largest projects completely determine the outcome. By limiting the amount of merge scenarios per project, smaller projects with fewer merge scenarios are also allowed to meaningfully impact the results. This makes the results more representative of the population of projects rather than the population of merge scenarios.

There are three primary threats to internal validity, all of which are related to the execution of the experiments. First, running time measurements are not perfectly reliable because of the underlying variance of the system, even with 10 repetitions of each merge. Second, the experiment scripts are relatively complex, and there is a possibility that they contain errors. To mitigate such risks, all our benchmark scripts are made publicly available in our online appendix[25]. Third, the results are only valid for one set of tuning parameters, and these are not necessarily optimal for any of the tested tools. Notably, the experiments were executed with JDIME's default settings. This for example means that its lookahead heuristics for identifying renamed methods and shifted code were not enabled, which if enabled could have helped avoid some conflicts at the cost of increased running time [23].

The 83 file merges excluded on the basis of at least one tool exhibiting a merge failure also pose a threat to validity. As the overlap in failing file merges is small between the tools, there is a possibility that these exclusions are more advantageous for some tools than others. For example, excluding a file merge where tool A times out benefits the running time results of tool A. Similarly, excluding a file merge where tool B crashes or produces an empty file can mask poor formatting preservation or large amounts and sizes of conflicts, potentially benefiting tool B.

25. https://github.com/slarse/spork-experiments

7 Related Work

Merging of source code is an active research field. This section presents the most closely related work on merge tools in Section 7.1, and other approaches to assist in the merging of code in Section 7.2.

7.1 Structured and semistructured merge

This section outlines related work on structured and semistructured merging. Section 7.1.1 presents structured diff algorithms, Section 7.1.2 presents complete structured merge tools and Section 7.1.3 presents related work on semistructured merge.

7.1.1 Structured diff algorithms

The distinction between an unstructured diff algorithm and a structured one is that the former operates on raw text, while the latter operates on some form of structure that the text encodes [3]. Most often, that entails some form of tree structure, ranging from ordered trees to represent structured text documents [44] to fully resolved ASTs [26]. More generalized graph representations can also be utilized [3], [45].

LADIFF represents one of the earliest structured diff algorithms that can deal with insertions, deletions, updates and moves [44]. It targets structured text documents, such as LaTeX and HTML. The algorithm relies heavily on an assumption that each leaf node in a tree T_1 has at most one highly similar leaf node in another tree T_2. This makes it unsuitable for source code differencing.

CHANGEDISTILLER improves upon LADIFF by removing the assumption of unique matchings for leaf nodes [46], making it more suitable for source code differencing. Leaf nodes are however represented as text, meaning that there is still room for increased granularity.

GUMTREE is a structured diff algorithm that like LADIFF and CHANGEDISTILLER can operate on insertions, deletions, updates and moves [26]. However, it operates on a fully resolved AST, making it more granular. We make use of GUMTREE in our own work.

CALCDIFF is another structured diff algorithm that operates on a control flow graph instead of an AST [45]. It is specifically designed to target object-oriented languages, and in particular with static code analysis in mind, such as being able to predict test coverage changes based on changes to the production source code.

7.1.2 Structured merge tools

Structured merge tools typically make use of a structured diff algorithm to identify changes across revisions, and based on that information use varying strategies for computing a merge. The topic was first studied in the early 1990s [10].

JDIME is a three-way structured merge tool for JAVA that implements its own tree differencing and merging algorithms [4], [11]. The matching step is simplistic and can only detect insertions and deletions. A heuristic lookahead mechanism built on top of the matching does however allow for limited move and update detection [23]. The work on JDIME is closely related to our own work, and we have drawn a great deal of inspiration from it. What sets our work apart is more powerful tree matching, a focus on providing minimal textual diffs with high-fidelity pretty-printing, and overall more modern components allowing support for newer versions of JAVA.

Another approach to structured merge is to use a generic, textual representation of ASTs, and then merge with a standard line-based merge algorithm [32]. The proposed algorithm can work either with unique identifiers stored across revisions to avoid the need for tree differencing, or use a differencing algorithm such as GUMTREE to compute matchings.

3DM is a move-enabled three-way merge tool designed for XML documents, with a novel merge algorithm that is applicable to any form of ordered tree [15]. It operates on units of small node contexts of three nodes; a parent node, and two of its children in the order they appear in its child list. This makes the merge granular, and it is also efficient with a time complexity of $\mathcal{O}(n * log(n))$. We implement the merge algorithm from 3DM in our own work.

Another approach for merging XML documents is to apply diffs computed on one version of a document to another version of it [47], [48]. This approach has the benefit of not requiring all three revisions to be present on the same machine, which may prove useful in situations where bandwidth is highly limited. It is however by nature less precise than a traditional three-way merge, such as the one implemented by 3DM.

7.1.3 Semistructured merge tools

Semistructured merge tools represent an attempt to find a middle-ground between structured and unstructured merging in terms of accuracy and running time performance [20]. The idea is to merge high-level elements such as method headers structurally, and use unstructured merge within fine-grained code elements such as method bodies.

FSTMERGE is the earliest example of semistructured merge [19], and provides a framework for implementing semistructured merge tools. Merge tools built on FSTMERGE have been shown to improve upon unstructured merge for JAVA, PYTHON and C# [20], [21], [22]. An implementation for JAVASCRIPT also exists, but the approach of semistructured merge yields significantly smaller improvements for JAVASCRIPT than it does for a language like JAVA [40].

INTELLIMERGE presents a different approach to semistructured merge for JAVA [25]. It uses a lightweight graph to represent the overall structure of a program, while keeping method bodies in textual form. While graph-based merging techniques typically suffer from excessive running times [25], [26], INTELLIMERGE is shown to be even faster than a comparable specialization of FSTMERGE.

7.2 Other Approaches

Orthogonally to the development of better merge tools, there are two other major approaches to assisting the merging of code. The first of these is *conflict resolution helpers*. The most straightforward of such tools are simple visualizers of conflicts, such as KDIFF3, MELD and WINMERGE. More involved tools may provide collaborative online environments for solving conflicts [49], automated suggestions for which developers are best equipped to solve some given conflict [50], replaying of individual edits [51] and even synthesizing of solutions to conflicts [31].

The second major approach is to avoid conflicts by predicting them before they occur. Workspace awareness tools such as SYDE [52], PALANTIR [53], CASSANDRA [9] and CRYSTAL [8] monitor the workspaces of individual developers and try to predict where conflicts may occur with other developers. This is typically done by preemptively merging developers' branches with each other, with some variations in the exact mechanisms, the merge tools used and the amount of validation of the merged systems. A more recent trend is to do lightweight feature analysis in order to predict conflicts [6], [29], [35], [54], or predict the difficulty of resolving a conflict that has already manifested [55]. This can potentially enhance workspace awareness tool accuracy while also reducing computational cost.

8 Conclusion

In this paper, we have presented a novel structured merge system for JAVA, called SPORK. SPORK, uniquely based on the 3DM algorithm, embeds essential domain knowledge of the JAVA programming language in order to minimize the amount of conflicts and the impact on formatting. We have presented a systematic and large scale empirical evaluation, showing that SPORK makes significant improvements to key metrics of merging, including running times and preservation of source code formatting.

We observe that formatting is an important aspect of source code that developers care deeply about, and plays a prominent role in readability and maintenance. As such, merge tools that do not preserve the formatting that developers have put in place are unlikely to be widely adopted. While SPORK presents a major improvement over comparable tools in terms of preserving formatting, it still in part makes use of low-fidelity pretty-printing that alters formatting. We believe that future research on structured merge should focus on improving formatting preservation even further, as without near perfect preservation of formatting, real-world applicability of structured merge remains limited.

References

[1] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, "The promises and perils of mining git," in *2009 6th IEEE International Working Conference on Mining Software Repositories*. IEEE, may 2009.

[2] C. Bird and T. Zimmermann, "Assessing the value of branches with what-if analysis," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12*. ACM Press, 2012.

[3] T. Mens, "A state-of-the-art survey on software merging," *IEEE Transactions on Software Engineering*, vol. 28, no. 5, pp. 449–462, may 2002.

[4] O. Leßenich, S. Apel, and C. Lengauer, "Balancing precision and performance in structured merge," *Automated Software Engineering*, vol. 22, no. 3, pp. 367–397, may 2014.

[5] N. Nelson, C. Brindescu, S. McKee, A. Sarma, and D. Dig, "The life-cycle of merge conflicts: processes, barriers, and strategies," *Empirical Software Engineering*, vol. 24, no. 5, pp. 2863–2906, feb 2019.

[6] K. Dias, P. Borba, and M. Barreto, "Understanding predictive factors for merge conflicts," *Information and Software Technology*, p. 106256, jan 2020.

[7] P. Accioly, P. Borba, and G. Cavalcanti, "Understanding semi-structured merge conflict characteristics in open-source java projects," *Empirical Software Engineering*, vol. 23, no. 4, pp. 2051–2085, dec 2017.

[8] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Early detection of collaboration conflicts and risks," *IEEE Transactions on Software Engineering*, vol. 39, no. 10, pp. 1358–1375, oct 2013.

[9] B. K. Kasi and A. Sarma, "Cassandra: Proactive conflict minimization through optimized task scheduling," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. IEEE Press, 2013, p. 732–741.

[10] B. Westfechtel, "Structure-oriented merging of revisions of software documents," in *Proceedings of the 3rd International Workshop on Software Configuration Management*. ACM Press, 1991.

[11] O. Leßenich, "Adjustable syntactic merge of java programs," Master's thesis, Universität Passau, 2012.

[12] D. Waddington and B. Yao, "High-fidelity c/c++ code transformation," *Science of Computer Programming*, vol. 68, no. 2, pp. 64–78, sep 2007.

[13] V. Savchenko, K. Sorokin, G. Pankratenko, S. Markov, A. Spiridonov, I. Alexandrov, A. Volkov, and K. Sun, "Nobrainer: An example-driven framework for c/c++ code transformations," in *Lecture Notes in Computer Science*. Springer International Publishing, 2019, pp. 140–155.

[14] R. P. Buse and W. R. Weimer, "A metric for software readability," in *Proceedings of the 2008 international symposium on Software testing and analysis - ISSTA '08*. ACM Press, 2008.

[15] T. Lindholm, "A three-way merge for XML documents," in *Proceedings of the 2004 ACM symposium on Document engineering - DocEng '04*. ACM Press, 2004.

[16] S. Larsén, "Spork: Move-enabled structured merge for java with gumtree and 3dm," Master's thesis, KTH Royal Institute of Technology, 2020. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-281960

[17] C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig, "How do centralized and distributed version control systems impact software changes?" in *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*. ACM Press, 2014.

[18] K. Muşlu, C. Bird, N. Nagappan, and J. Czerwonka, "Transition from centralized to decentralized version control systems: a case study on reasons, barriers, and outcomes," in *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*. ACM Press, 2014.

[19] S. Apel, J. Liebig, C. Lengauer, C. Kästner, and W. R. Cook, "Semistructured merge in revision control systems." in *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, 2010, pp. 13–19.

[20] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner, "Semistructured merge," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE '11*. ACM Press, 2011.

[21] G. Cavalcanti, P. Accioly, and P. Borba, "Assessing semistructured merge in version control systems: A replicated experiment," in *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, oct 2015.

[22] G. Cavalcanti, P. Borba, and P. Accioly, "Evaluating and improving semistructured merge," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–27, oct 2017.

[23] O. Leßenich, S. Apel, C. Kästner, G. Seibt, and J. Siegmund, "Renaming and shifted code in structured merging: Looking ahead for precision and performance," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. Piscataway, NJ, USA: IEEE Press, 2017, pp. 543–553. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155631

[24] F. Zhu, F. He, and Q. Yu, "Enhancing precision of structured merge by proper tree matching," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, may 2019.

[25] B. Shen, W. Zhang, H. Zhao, G. Liang, Z. Jin, and Q. Wang, "IntelliMerge: a refactoring-aware software merging technique," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–28, oct 2019.

[26] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Montperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE '14*. ACM Press, 2014.

[27] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "SPOON: A library for implementing analyses and transformations of java source code," *Software: Practice and Experience*, vol. 46, no. 9, pp. 1155–1179, aug 2015.

[28] S. Apel, O. Leßenich, and C. Lengauer, "Structured merge with auto-tuning: balancing precision and performance," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012, pp. 120–129.

[29] O. Leßenich, J. Siegmund, S. Apel, C. Kästner, and C. Hunsen, "Indicators for merge conflicts in the wild: survey and empirical study," *Automated Software Engineering*, vol. 25, no. 2, pp. 279–313, sep 2017.

[30] G. Cavalcanti, P. Borba, G. Seibt, and S. Apel, "The impact of structure on software merging: Semistructured versus structured merge," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1002–1013.

[31] F. Zhu and F. He, "Conflict resolution for structured merge via version space algebra," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–25, Oct. 2018. [Online]. Available: https://doi.org/10.1145/3276536

[32] D. Asenov, B. Guenat, P. Müller, and M. Otth, "Precise version control of trees with line-based version control systems," in *Fundamental Approaches to Software Engineering*. Springer Berlin Heidelberg, 2017, pp. 152–169.

[33] S. McKee, N. Nelson, A. Sarma, and D. Dig, "Software practitioner perspectives on merge conflicts and resolutions," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, sep 2017.

[34] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating GitHub for engineered software projects," *Empirical Software Engineering*, vol. 22, no. 6, pp. 3219–3253, apr 2017.

[35] M. Owhadi-Kareshk, S. Nadi, and J. Rubin, "Predicting merge conflicts in collaborative software development," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, sep 2019.

[36] G. G. L. Menezes, L. G. P. Murta, M. O. Barros, and A. V. D. Hoek, "On the nature of merge conflicts: a study of 2,731 open source java projects hosted by GitHub," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018.

[37] G. Vale, C. Hunsen, E. Figueiredo, and S. Apel, "Challenges of resolving merge conflicts: A mining and survey study," *IEEE Transactions on Software Engineering*, 2021.

[38] D. S. Kerby, "The simple difference formula: An approach to teaching nonparametric correlation," *Comprehensive Psychology*, vol. 3, p. 11.IT.3.1, jan 2014.

[39] M. Tomczak and E. Tomczak, "The need to report effect size estimates revisited. an overview of some recommended measures of effect size," *Trends in sport sciences*, vol. 1, no. 21, pp. 19–25, 2014.

[40] A. T. Tavares, P. Borba, G. Cavalcanti, and S. Soares, "Semistructured merge in JavaScript systems," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, nov 2019.

[41] H. Adzemovic, "A template-based approach to automatic program repair of sonarqube static warnings," Master's thesis, KTH, School of Electrical Engineering and Computer Science (EECS), 2020.

[42] M. Nagappan, T. Zimmermann, and C. Bird, "Diversity in software engineering research," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*. ACM Press, 2013.

[43] T. Ji, L. Chen, X. Yi, and X. Mao, "Understanding Merge Conflicts and Resolutions in Git Rebases," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, 2020.

[44] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change detection in hierarchically structured information," in *Proceedings of the 1996 ACM SIGMOD international conference on Management of data - SIGMOD '96*. ACM Press, 1996.

[45] T. Apiwattanapong, A. Orso, and M. Harrold, "A differencing algorithm for object-oriented programs," in *Proceedings. 19th International Conference on Automated Software Engineering, 2004.* IEEE, 2004.

[46] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change distilling:tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, nov 2007.

[47] S. Rönnau, C. Pauli, and U. M. Borghoff, "Merging changes in XML documents using reliable context fingerprints," in *Proceeding of the eighth ACM symposium on Document engineering - DocEng '08*. ACM Press, 2008.

[48] S. Rönnau, G. Philipp, and U. M. Borghoff, "Efficient and reliable merging of XML documents," in *Proceeding of the 18th ACM conference on Information and knowledge management - CIKM '09*. ACM Press, 2009.

[49] A. Nieminen, "Real-time collaborative resolving of merge conflicts," in *Proceedings of the 8th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing*. IEEE, 2012.

[50] C. Costa, J. Figueiredo, L. Murta, and A. Sarma, "TIPMerge: recommending experts for integrating changes across branches," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*. ACM Press, 2016.

[51] Y. Nishimura and K. Maruyama, "Supporting merge conflict resolution by using fine-grained code change history," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, mar 2016.

[52] L. Hattori and M. Lanza, "Syde: A tool for collaborative software development," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*. ACM Press, 2010.

[53] A. Sarma, D. F. Redmiles, and A. van der Hoek, "Palantir: Early detection of development conflicts arising from parallel code changes," *IEEE Transactions on Software Engineering*, vol. 38, no. 4, pp. 889–908, jul 2012.

[54] P. Accioly, P. Borba, L. Silva, and G. Cavalcanti, "Analyzing conflict predictors in open-source java projects," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. New York, NY, USA: ACM, 2018, pp. 576–586. [Online]. Available: http://doi.acm.org/10.1145/3196398.3196437

[55] C. Brindescu, I. Ahmed, R. Leano, and A. Sarma, "Planning for untangling," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ACM, jun 2020.