Conflict Resolution for Structured Merge via Version Space Algebra

FENGMIN ZHU, School of Software, Tsinghua University, Key Laboratory for Information System Security, MoE, and Beijing National Research Center for Information Science and Technology, China
FEI HE*, School of Software, Tsinghua University, Key Laboratory for Information System Security, MoE, and Beijing National Research Center for Information Science and Technology, China

Resolving conflicts is the main challenge for software merging. The existing merge tools usually rely on the developer to manually resolve conflicts. This is of course inefficient. We propose an interactive approach for resolving merge conflicts. To the best of our knowledge, this is the first attempt for conflict resolution of structured merge. To represent the possibly very large set of candidate programs, we propose an expressive and efficient representation by version space algebra. We also design a simple mechanism for ranking resolutions in the program space, such that the top-ranked resolution is very likely to meet the developer's expectation. We prototype our approach as a merge tool `AutoMerge`, and evaluate it on 244 real-world conflicts arising from 10 open-source projects. Results show great practicality of our approach.

CCS Concepts: • **Software and its engineering → Software version control**; **Context specific languages**;

Additional Key Words and Phrases: Revision control systems, software merge, conflict resolution, version space algebra

ACM Reference Format:
Fengmin Zhu and Fei He. 2018. Conflict Resolution for Structured Merge via Version Space Algebra. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 166 (November 2018), 25 pages. https://doi.org/10.1145/3276536

1 INTRODUCTION

Cooperation is an essence of contemporary software development. Revision control systems are invented to simplify the management of revisions and evolutions of software systems. A common workflow under revision control can be described as follows: two developers create their own versions by deriving it from a *base* version; then these two versions, named the *left* and *right* versions, evolve independently by either adding new functionalities or fixing previous issues; finally the left and right versions are merged again with the base version. A major problem here is that conflicts occur when concurrent changes contradict each other. Zimmermann [2007] found that between 22.75% and 46.62% of all merge integrations in four large projects exhibit conflicts.

Software merging techniques can be classified into unstructured approaches and structured approaches. Unstructured merge regards programs as sequences of plain text and perform the

*Corresponding Author

Authors' addresses: Fengmin Zhu, School of Software, Tsinghua University , Key Laboratory for Information System Security, MoE , Beijing National Research Center for Information Science and Technology, Beijing, China, zfm17@mails.tsinghua.edu.cn; Fei He, School of Software, Tsinghua University , Key Laboratory for Information System Security, MoE , Beijing National Research Center for Information Science and Technology, Beijing, China, hefei@tsinghua.edu.cn.

2475-1421/2018/11-ART166
https://doi.org/10.1145/3276536

merge line by line. It is quite simple and efficient, and has been adopted in many practical version control systems, such as `CVS`, `subversion`, and `git`. However, since it neglects all knowledge of the language, it reveals weaknesses in expressing differences and handling merge conflicts. Structured merge treats programs as abstract syntax trees. It is able to exploit the language-specific knowledge and hence is more precise than unstructured approaches [Mens 2002]. Both approaches follow the three-way merge rules [Westfechtel 1991] to detect merge conflicts. In case of conflicts, both approaches cannot give any resolution.

The main reason that existing merge tools do not attempt to resolve conflicts is *safety*: when conflicts present, the resolution is usually ambiguous, it is thus dangerous to directly guess a resolution and then apply it to the conflict. Existing tools rely on the developer to manually resolve merge conflicts. We argue that providing the developer with a variety of useful candidate resolutions is also valuable. We thus propose an interactive mechanism to present the developer with candidate resolutions.

Our approach is motivated by the following observation: when a developer is reported of a conflict, it is usually unnecessary to compare the whole program. Instead, one usually compares the conflicting sections merely, and form a resolution by "combining" them in a rearranged way. These combinations form a program space. Since the program space is possibly very large, directly representing it in an explicit manner is impractical. Instead, we rely on *version space algebra* (VSA) to implicitly represent the program space. We also need a ranking mechanism to find a resolution in the program space that is very likely to meet the developer's expectation. A proper ranking function reduces the interaction rounds and accelerates the convergence of our approach. The design of our ranking function is problem-specific, and motivated by the three-way merge rules.

We prototype our VSA-based approach as a merge tool `AutoMerge`. To demonstrate the practicality of our approach, we evaluate `AutoMerge` on 10 real-world software projects selected from Github[1]. We choose 95 merge scenarios from the repositories of these projects, and let the actual merge commits be the expected versions. We then run a state-of-the-art structured merger `JDime`[2] on these merge scenarios and get 244 conflicts. Note that all these conflicts cannot be resolved by `JDime`. With our `AutoMerge`, 232 of 244 (95.1%) conflicts are successfully resolved. Among the 232 successful cases, `AutoMerge` needs only to try 1.79 candidates on average until the expected version is hit.

The main technical contributions of this paper are summarized as follows:

- We propose an interactive approach for resolving merge conflicts. To the best of our knowledge, this is the first attempt on conflict resolution of structured merge.
- We identify the expressiveness and efficiency of version space algebra, and elaborate an algorithm to automatically construct the VSA representation of the program space.
- We design a simple but effective mechanism for ranking resolutions in the program space.
- We prototype our approach as `AutoMerge`, and conduct experiments on real-world software projects. Results show great practicality of our approach.

The rest of the paper is organized as follows: In Section 2, we introduce preliminaries on structured merge and version space algebra. In Section 3, we illustrate our approach with a motivating example. In Section 4, we first give an overview of our approach, and then formulate relevant concepts and explain details of our VSA-based approach. In Section 5, we present additional details on implementing our `AutoMerge`. In Section 6, we propose research questions on evaluation, and discuss the results. In Section 7, we survey related studies on software merge, program synthesis

[1] https://github.com
[2] https://github.com/se-passau/jdime

Table 1. Basic rules of three-way merge.

	Type	Base B	Left L	Right R	Target T
1	Node	e	e	e'	e'
2	Node	e	e_L	e_R	conflict
3	List	$e \in B$	$e \in L$	$e \notin R$	$e \notin T$
4	List	$e \notin B$	$e \in L$	$e \notin R$	$e \in T$ or conflict

and program transformation, then we compare our work with theirs. Finally, we conclude the paper in Section 8.

2 PRELIMINARIES

2.1 Software Merge

Two-way merge and three-way merge are two distinct methods for software merge: the former attempts to directly join two versions without using any inherited information from the version control system, whereas the latter traces the common ancestor version where these two versions are derived from, and thus has more information to determine where a change comes from and whether it is a conflict [Mens 2002]. In the context of three-way merge, the two versions to be merged are called the *left* version and the *right* version, and their common ancestor is called the *base* version. The base, left and right versions form a *merge scenario*. The merged version is called the *target* version.

Widely-used three-way merge tools like `diff` and `merge` follow the traditional unstructured manner, i.e. line-based strategy. Unstructured merge is applicable to a wide range of real-world software projects while being efficient. Source code written in any programming language is regarded as plain text. This has the benefit of language-independence, as well as the weakness of low precision. To alleviate the issue of conflict resolution in low precision, Westfechtel [1991] employed structural information such as the context-free and context-sensitive syntax during the merge process. Later, researchers have proposed variates of structural diff and merge tools.

Structured merge and unstructured merge both follow the basic rules of three-way merge shown in Table 1. If a node is modified in exactly one of the opposing versions, then it is included in the merged version (row 1). In case it is concurrently modified in both versions, then a conflict is created (row 2). If a node which appears in the base version is missing in exactly one of the opposing versions, then it is regarded as removed and thus will be excluded in the merged version (row 3). In the contract, if a new node is introduced in exactly one of the opposing versions, then it is regarded as inserted (row 4). However, in this case, the position where the new node has to be inserted is critical, and we have to concern if the insertion position is ambiguous. If it is, a conflict has to be reported as well.

The underlying idea of structured merge is to represent programs as trees which reflect their context-free syntax. Abstract syntax tree (AST) is a natural and structured way to represent programs. In this paper, we focus on the following three kinds of AST nodes:

$$\begin{array}{lcll} \text{AST } N & ::= & V & \text{(leaf)} \\ & | & F(N_1, N_2, \ldots, N_k) & \text{(constructed)} \\ & | & \texttt{List}(N_1, N_2, \ldots, N_k) & \text{(list)} \end{array}$$

A *leaf* node has no children. A *constructed* node (internal node) has a fixed number of children $N_1, N_2, \ldots, N_k$. It is constructed with a k-ary constructor F. For instance, in Java language, a try-statement is constructed by a 2-ary constructor `Try`, where the first argument is the try-block,

and the second argument is the catch-block. A *list* is a special internal node that has an arbitrary number (possibly empty) of children $N_1, N_2, \ldots, N_k$.

For lists, extra information on whether the siblings can be safely permuted has been proved useful in structured merge [Apel et al. 2011]. Those that cannot be safely permuted are called *ordered*, otherwise *unordered*. For instance, in Java, a sequence of statements inside a block are *ordered*. Conversely, a number of method declarations inside a class are *unordered*. We use the notation $\texttt{List}^{\text{ord}}(N_1, N_2, \ldots, N_k)$ or $\texttt{List}^{\text{unord}}(N_1, N_2, \ldots, N_k)$ to emphasize that the elements of a list are ordered or unordered. Two unordered lists are *equivalent* if they have the identical elements, regardless of their order, i.e. $\texttt{List}^{\text{unord}}(N_1, N_2, \ldots, N_k) = \texttt{List}^{\text{unord}}(N_{i_1}, N_{i_2}, \ldots, N_{i_k})$ if $\{1, 2, \ldots, k\} = \{i_1, i_2, \ldots, i_k\}$. We ignore the superscript annotation and simply write $\texttt{List}(N_1, N_2, \ldots, N_k)$ for convenience when we can clearly determine if it is ordered or unordered.

We distinguish if elements are ordered or unordered in the last rule (row 4) shown in Table 1. In unordered merge, the positions of the elements are unimportant. Therefore, no conflicts will be reported in this case. Nevertheless, in ordered merge, the insertion position is sometimes ambiguous and conflicts must be reported.

While structured merge augments the merge precision, the time complexity of the internal tree matching algorithm is extremely high. Unfortunately, the computation of the *largest common embedded subtrees*, a frequently-used algorithm for matching across different levels, has been proved to be APX-hard [Zhang and Jiang 1994]. Recently, a semistructured approach [Apel et al. 2011] that combines structured merge and unstructured merge gains the benefit of both: the generality of unstructured merge and the expressiveness of structured merge.

Despite the fact that structured merge is more precise than unstructured merge, conflicts are unavoidable when conditions of row 2 and 4 in Table 1 are encountered. Our approach is capable of resolving conflicts under these circumstances.

2.2 Version Space Algebra

Version space algebra is a succinct way to represent a large number of programs in polynomial space. It is defined by Mitchell [1982] and later expanded upon specific applications by Lau et al. [2000] and Polozov and Gulwani [2015]. From an intuitive aspect, a VSA can be viewed as a directed graph where each node represents a set of programs. A leaf node uses explicit representation: it is directly annotated with a set of programs. An internal node uses implicit representation: it is a composition of several VSA nodes. There are two kinds of internal nodes – *union nodes* and *join nodes*. A *union node* is a set-theoretic union of the program sets represented by its children. A *join node* is a cross product over all possible applications of a k-ary constructor F over k arguments, each individually chosen from its corresponding child node.

Formally, the grammar of the three kinds of VSA nodes is:

$$\begin{array}{rrll} \text{VSA } \widetilde{N} & ::= & \{P_1, P_2, \ldots, P_k\} & \text{(explicit)} \\ & | & \widetilde{N_1} \cup \widetilde{N_2} \cup \cdots \cup \widetilde{N_k} & \text{(union)} \\ & | & F_{\bowtie}(\widetilde{N_1}, \widetilde{N_2}, \ldots, \widetilde{N_k}) & \text{(join)} \end{array}$$

where $P_1, P_2, \ldots, P_k$ are programs and F is a k-ary constructor. For each kind of VSA $\cdot$, we denote the set of programs it represents as $[\![\cdot]\!]$, named the *program space*:

$$\begin{aligned} [\![\{P_1, P_2, \ldots, P_k\}]\!] &= \{P_1, P_2, \ldots, P_k\} \\ [\![\widetilde{N_1} \cup \widetilde{N_2} \cup \cdots \cup \widetilde{N_k}]\!] &= [\![\widetilde{N_1}]\!] \cup [\![\widetilde{N_2}]\!] \cup \cdots \cup [\![\widetilde{N_k}]\!] \\ [\![F_{\bowtie}(\widetilde{N_1}, \widetilde{N_2}, \ldots, \widetilde{N_k})]\!] &= \{F(P_1, P_2, \ldots, P_k) \mid P_1 \in [\![\widetilde{N_1}]\!], P_2 \in [\![\widetilde{N_2}]\!], \ldots, P_k \in [\![\widetilde{N_k}]\!]\} \end{aligned}$$

```
...
for (...) {
  try {
    s1;
  } catch {
    // empty
  }
}
...
```

(a) *Base*

```
...
for (...) {
  s3;
  if (...) {
    try {
      s4;
    } catch {
      // empty
    }
  }
}
...
```

(b) *Left*

```
...
for (...) {
  try {
    s1;
  } catch {
    s2;
  }
}
...
```

(c) *Right*

```
...
for (...) {
  s3;
  if (...) {
    try {
      s4;
    } catch {
      s2;
    }
  }
}
...
```

(d) *Merged*

Fig. 1. The *base*, *left*, *right* and *merged* versions of the motivating example. Changes are highlighted.

Specially, an *empty VSA*, which represents an empty set of programs, is denoted by $\varnothing$. We say that a program P is *included in* a VSA $\widetilde{N}$, denoted by $P \in \widetilde{N}$, if and only if $P \in [\![\widetilde{N}]\!]$. No program is included in an empty VSA. Since a program can be represented by its AST, a VSA also represents a set of ASTs.

The *size* of a VSA $\widetilde{N}$ is the number of programs in its program space $[\![\widetilde{N}]\!]$, denoted by $\left|\widetilde{N}\right| = \left|[\![\widetilde{N}]\!]\right|$. Equivalently, the size of a VSA can be obtained without explicitly computing the program space:

$$
\begin{aligned}
|\{P_1, P_2, \ldots, P_k\}| &= k \\
\left|\widetilde{N_1} \cup \widetilde{N_2} \cup \cdots \cup \widetilde{N_k}\right| &= \left|\widetilde{N_1}\right| + \left|\widetilde{N_2}\right| + \cdots + \left|\widetilde{N_k}\right| \\
\left|F_{\bowtie}(\widetilde{N_1}, \widetilde{N_2}, \ldots, \widetilde{N_k})\right| &= \left|\widetilde{N_1}\right| \times \left|\widetilde{N_2}\right| \times \cdots \times \left|\widetilde{N_k}\right|
\end{aligned}
$$

VSA has been adopted in programming by demonstration [Lau et al. 2000] and programming by examples [Gulwani 2011; Polozov and Gulwani 2015]. In those applications, the program space is huge: a practical program space defined by a domain-specific language may have up to 10^{30} programs that are consistent with the given specification [Singh and Gulwani 2012].

3 A MOTIVATING EXAMPLE

In this section, we use a simple program to motivate our approach. Figure 1a to 1c show three versions (*base*, *left* and *right*) of the program. They form a merge scenario. We are required to merge *left* and *right*. Figure 1d shows a possible *merged* result.

Note that both *left* and *right* modify the for-loop body of *base* (highlighted in Fig. 1). The line-based unstructured approaches treat programs as sequences of plain text, and will recognize the highlighted segments of *left* and *right* as different text lines. Similarly, the structured approaches treat programs as abstract syntax trees, and will conclude that the try-statement of *base* are modified by both *left* and *right*. For either kind of the approaches, this merge scenario will be concluded as a "conflict" (according to three-way merge rules), and no resolution will be given by the existing approaches.

We intend to develop an approach to resolve conflicts, and form the merged program automatically. Looking at the highlighted segments of *left* and *right*, a few changes are made in each version. In *left*, the try-statement has been embedded into an if-statement, and the try-block of the try-statement has been changed from the statement `s1` to `s4`. Additionally, the statement `s3` is

inserted before the if-statement. In *right*, the catch-block of the try-statement has been implemented as a single statement `s2`. All these modifications reflect the developer's intentions to some extent. A resolution can be considered as a *merge* of these modifications. The merge here is not simply to put all the modifications together. Instead, each modification is possible to be kept or not kept in the resolution. Moreover, the merge procedure may yield new statements that do belong to neither version. For example, the try-statement in Figure 1d is a new statement, composed of the try-block taken from *left* and the catch-block taken from *right*.

Obviously, there are possibly many resolutions for a conflict, each yields a merged program. We expect a compact representation to express all merged programs. It should be expressive enough to include the program that meets the developer's expectation, and also restricted enough, so that the expected program can be found efficiently. As discussed in Section 2.2, VSA is an appropriate representation to cooperate program versions. First, they are capable of expressing cross products of components. Second, the node sharing mechanism of VSA makes it a very compact representation. We will discuss in Section 4.2 how to represent a possibly large program space by VSA in an expressive and restricted way.

In the last step, we intend to figure out a merged result from the program space. To make our approach practical, we specify ranking rules (see Section 4.3 for details) and select the top-ranked programs as recommendations to the developer.

4 APPROACH

This section presents our VSA-based approach for resolving conflicts in structured merge. A three-way *merge scenario* is a triple (B, L, R), where B, L and R represents the base, left and right version, respectively. Our algorithm (Algorithm 1) takes a merge scenario (B, L, R) as input, and generates a target T as the merged result. The process is mainly summarized as the following three steps:

(1) Conflict detection. A structured merger is applied on the merge scenario (B, L, R) to generate a target program T_H with a set of holes H (line 4). In case of conflicts, the structured merger inserts a *hole* in the target program, and records the relevant segments in B, L and R that cause the conflict. In the motivating example, the highlighted statements in *base*, *left* and *right* (Figure 1) form a hole.
(2) Program space representation. For each hole $h \in H$, we construct a VSA $\widetilde{S_h}$ (line 7), which represents all possible resolutions that can instantiate the hole h.
(3) Resolution ranking. We rank the candidate resolutions in $\widetilde{S_h}$ with a ranking function (line 8). We first instantiate the hole h with the top-ranked resolution and generate a candidate program, which is then presented to the developer. If the developer rejects this candidate program, then we instantiate the hole with the next ranked resolution and generate the next candidate program. We count the number of presented candidate programs until the developer accepts a result. Finally, we instantiate the hole h with the accepted resolution P (line 9). Our approach fails if the developer rejects all candidate programs defined by the constructed VSA $\widetilde{S_h}$.

When all the holes are successfully instantiated, the algorithm terminates and returns the merged result T (line 10).

4.1 Conflict Detection

We rely on a structured merger to detect the conflicts. Given a merge scenario (B, L, R), a typical structured merge algorithm has two passes: tree matching and amalgamation. In the tree matching pass, the algorithm finds the *largest common embedded subtree* of two trees (T_1, T_2), and generates a set of node pairs. Each node pair (n_1, n_2), where tree node n_1 is taken from T_1 and n_2 is taken

Algorithm 1 Merge

```
1:  procedure MERGE(B, L, R)
2:      Input: Base version B, left version L and right version R
3:      Output: Merged result T
4:      perform a structured merge on (B, L, R) and generate T_H;
5:      T ← T_H;
6:      for all hole h ∈ H do
7:          S̃_h ← CONSTRUCTVSA(h);
8:          rank S̃_h;
9:          T ← T[h ↦ P] where P ∈ S̃_h is the accepted resolution;
10:     return T;
```

from T_2, indicates that n_1 and n_2 are matched with each other. For two-way merge, tree matching is simply performed once on (L, R). For three-way merge, tree matching is performed on (B, L), (B, R) and (L, R). In the tree amalgamation pass, the node pairs are compared and merged in a top-down, recursive fashion. For leaf nodes, the basic rules of three-way merge are applied. For constructed nodes, tree amalgamation is level-wised: its argument nodes are recursively processed in order. For lists, distinct algorithms are designed to be applied on ordered and unordered nodes accordingly. Finally, a target tree T is returned as the result with respect to the merge scenario (B, L, R).

If a conflict is detected in structured merge, then the algorithm produces a special *conflict node*. A conflict node is actually a forest with two trees – tree l for the conflicting subtree taken from L, and tree r for the conflicting subtree taken from R. These two subtrees, together with the corresponding subtree b taken from the base version B, form a *conflicting scenario*. We represent a conflicting scenario using a *hole*, denoted by $\texttt{Hole}(b, l, r)$. Therefore, when a structured merge is completed, a target tree T_H is created and it possibly has a number of holes H.

Remark that tree amalgamation is performed from top to bottom in a level-wise, recursive manner. As a result, the hole $h \in H$ is generated as deep as possible in the tree T_H. The subtrees that form the conflicting scenario are lifted as low as possible until the leaf level. This gives us the benefit that the generated VSA, without loss of expressiveness, is restricted to the full.

4.2 Program Space Representation

For every conflicting scenario, there may be a substantial number of resolutions. VSA is an appropriate representation to cooperate different program versions. It is capable of expressing cross products of components, and its memory sharing mechanism also makes it a very compact representation. We thus adopt VSA for representing the program space of the candidate instantiations of a hole.

The conventional VSA is incapable of representing the list structure, which, however, is necessary for composing the resolutions. We discuss in Section 4.2.1 an extension of VSA with a *list join* node. Given a hole $\texttt{Hole}(b, l, r)$, where b, l, r are represented in ASTs, to construct a VSA that represents all candidate instantiations, we first need the conversion rules of AST to VSA (Section 4.2.2), and then the merge rules of VSAs (Section 4.2.3). The VSA construction algorithm, as a formulation of the mechanisms stated in Section 4.2.2 and Section 4.2.3, is presented in Section 4.2.4.

4.2.1 Extending VSA with List Join. List structures, such as a block of statements, are common in real-world programs. However, the conventional VSA (Section 2.2) is unable to express this kind of structures.

Considering the motivating example, we want a statement block that can be used to instantiate the for-loop body. Given that this merged result can only be a combination of *left* and *right*, only

$$\frac{}{\alpha(V) = \{V\}}\ \text{A-EXP}$$

$$\frac{\widetilde{N_1} = \alpha(N_1), \widetilde{N_2} = \alpha(N_2), \ldots, \widetilde{N_k} = \alpha(N_k)}{\alpha(F(N_1, N_2, \ldots, N_k)) = F_{\bowtie}(\widetilde{N_1}, \widetilde{N_2}, \ldots, \widetilde{N_k})}\ \text{A-JOIN}$$

$$\frac{\widetilde{N} = \alpha(N_1) \cup \alpha(N_2) \cup \cdots \cup \alpha(N_k)}{\alpha(\texttt{List}(N_1, N_2, \ldots, N_k)) = \texttt{List}_{\bowtie}(\widetilde{N})}\ \text{A-LISTJOIN}$$

Fig. 2. Conversion rules for AST to VSA, where α is the conversion operation.

three kinds of statements need be considered: the statement `s3` (occurs in *left*), the if-statement (occurs in *left*), and the try-statement (occurs in both *left* and *right*). Other kinds of statements are neglected since they do not occur in either *left* or *right*. For the for-loop body, we expect a sequence of statements, which has a list structure. Note that the join node is unable to express a list. To succinctly represent such list structures, we introduce *list join* nodes.

Let $\widetilde{N}$ be a VSA node. We denote by $\texttt{List}_{\bowtie}^{\text{ord}}(\widetilde{N})$ and $\texttt{List}_{\bowtie}^{\text{unord}}(\widetilde{N})$ the *list join* node for ordered and unordered lists, respectively. Both of them represent several lists, each element of which is taken from the program space $[\![\widetilde{N}]\!]$:

$$[\![\texttt{List}_{\bowtie}^{\text{ord}}(\widetilde{N})]\!] = \{\texttt{List}^{\text{ord}}(N_1, N_2, \ldots, N_k) \mid k \geq 0, N_1, N_2, \ldots, N_k \in [\![\widetilde{N}]\!] \text{ are distinct}\}$$

$$[\![\texttt{List}_{\bowtie}^{\text{unord}}(\widetilde{N})]\!] = \{\texttt{List}^{\text{unord}}(N_1, N_2, \ldots, N_k) \mid k \geq 0, N_1, N_2, \ldots, N_k \in [\![\widetilde{N}]\!] \text{ are distinct}\}$$

Recall that $\texttt{List}^{\text{unord}}(N_1, N_2, \ldots, N_k) = \texttt{List}^{\text{unord}}(N_{i_1}, N_{i_2}, \ldots, N_{i_k})$ if $\{1, 2, \ldots, k\} = \{i_1, i_2, \ldots, i_k\}$. Apparently, the program space of an ordered list join node $\texttt{List}_{\bowtie}^{\text{ord}}(\widetilde{N})$ over $\widetilde{N}$ is much larger than that of an unordered list join node $\texttt{List}_{\bowtie}^{\text{unord}}(\widetilde{N})$ over the same $\widetilde{N}$. In the remainder of this paper, we neglect the superscript annotation and simply write $\texttt{List}_{\bowtie}(\widetilde{N})$ if we can clearly determine if the list join node is ordered or unordered.

The graph representation of an ordered list join node is available in Figure 4, where the root nodes $\widetilde{S}, \widetilde{S^L}, \widetilde{S^R}$ are ordered list join nodes.

In the motivating example, let $\texttt{List}_{\bowtie}(\widetilde{A})$ be the program space of the expected for-loop body. Assume that the expected for-loop body contains 2 statements, then $\texttt{List}_{\bowtie}(\widetilde{A})$ is a set of the following: (1) `s3` followed by an if-statement, (2) `s3` followed by a try-statement, (3) an if-statement followed by `s3`, (4) an if-statement followed by a try-statement, (5) a try-statement followed by `s3`, and (6) a try-statement followed by an if-statement.

After extension, the grammar of VSA becomes:

$$\begin{array}{rcll} \text{VSA } \widetilde{N} & ::= & \{P_1, P_2, \ldots, P_k\} & \text{(explicit)} \\ & \mid & \widetilde{N_1} \cup \widetilde{N_2} \cup \cdots \cup \widetilde{N_k} & \text{(union)} \\ & \mid & F_{\bowtie}(\widetilde{N_1}, \widetilde{N_2}, \ldots, \widetilde{N_k}) & \text{(join)} \\ & \mid & \texttt{List}_{\bowtie}(\widetilde{N}) & \text{(list join)} \end{array}$$

4.2.2 AST to VSA Conversion. The conversion of AST to VSA is performed node by node. Let α : AST $\rightarrow$ VSA be the conversion operation that maps from the set of AST nodes to the set of VSA nodes. There are three kinds of AST nodes, i.e., *leaf*, *constructed* and *list* nodes, then we define conversion rules for them respectively.

As shown in Figure 2, the conversion of a leaf node is trivially the VSA containing itself (A-EXP). The conversion of a constructed node is a join node representing programs generated by a cross

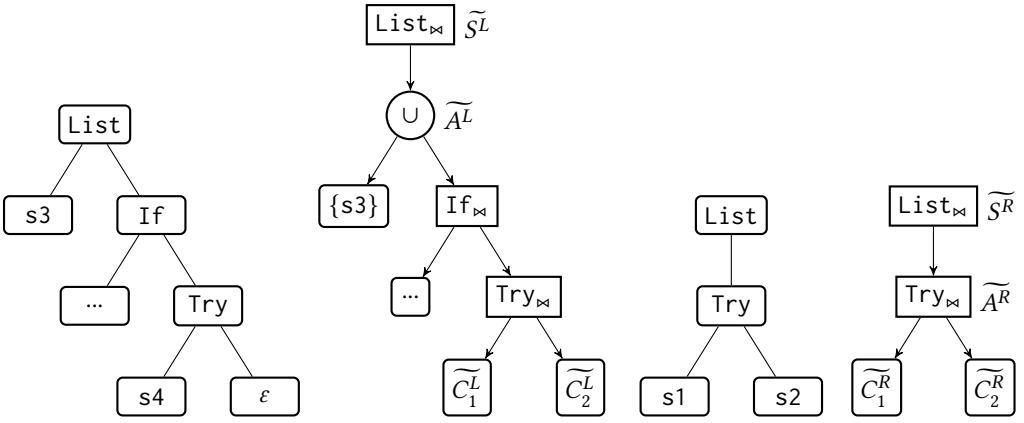


Fig. 3. Applying conversion to the motivating example: (a) and (c) represent the ASTs l and r, (b) and (d) represent the VSAs $\alpha(l)$ and $\alpha(r)$.

product over the k arguments (A-JOIN). The conversion of a list is a list join node, describing several lists of the elements taken from the union of its children VSAs (A-LISTJOIN).

For the motivating example, after applying the conversion operation α to the left conflicting subtree l and the right conflicting subtree r, we get $\alpha(l) = \widetilde{S^L}$, $\alpha(r) = \widetilde{S^R}$, and

$$\widetilde{S^L} = \texttt{List}_{\bowtie}(\widetilde{A_L}) \qquad \widetilde{S^R} = \texttt{List}_{\bowtie}(\widetilde{A_R})$$
$$\widetilde{A^L} = \{\texttt{s3}\} \cup \texttt{If}_{\bowtie}(\_, \widetilde{B}) \qquad \widetilde{A^R} = \texttt{Try}_{\bowtie}(\widetilde{C_1^R}, \widetilde{C_2^R})$$
$$\widetilde{B} = \texttt{Try}_{\bowtie}(\widetilde{C_1^L}, \widetilde{C_2^L})$$
$$\widetilde{C_1^L} = \{\texttt{s4}\} \qquad \widetilde{C_1^R} = \{\texttt{s1}\}$$
$$\widetilde{C_2^L} = \{\varepsilon\} \qquad \widetilde{C_2^R} = \{\texttt{s2}\}$$

where ε denotes an empty statement. The ASTs l, r, and their corresponding VSAs $\alpha(l)$, $\alpha(r)$ are depicted in Figure 3.

4.2.3 VSA Merge. Given two VSAs, each contains a single program, we now define the merge operation on VSAs. The merged VSA represents an enlarged program space, which contains not only the original programs which we can obtain from the input VSAs, but also some fresh programs, which do not belong to any of the input VSA.

Especially, we specify a merge operation on two VSA nodes by *sharing* some of their subnodes: union them and regard them as a whole. Details of the merge operation is discussed in Section 4.2.4. Here we simply explain the rationale behind the merge operation. Considering the two VSAs $\alpha(l)$, $\alpha(r)$ in Figure 3. If we share $\widetilde{C_1^L}$ with $\widetilde{C_1^R}$ by creating a new VSA node $\widetilde{C}_1 = \widetilde{C_1^L} \cup \widetilde{C_1^R}$, and substituting $\widetilde{C_1^L}$ and $\widetilde{C_1^R}$ with $\widetilde{C}_1$, then the try-block of a try-statement now has two options: `s4` and `s1`. Note that the necessary condition for sharing these two VSAs is that `s4` and `s1` are both try-blocks of some try-statement. Originally, $\left|\widetilde{C_1^L}\right| = \left|\widetilde{C_1^R}\right| = 1$. After applying the merge operation, $\left|\widetilde{C}_1\right|$ increases

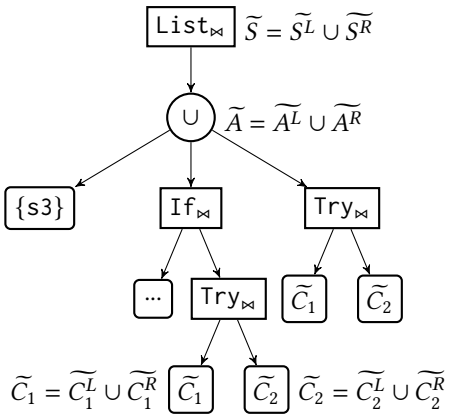


Fig. 4. Merged VSA of $\widetilde{S^L}$ and $\widetilde{S^R}$ in Figure 3.

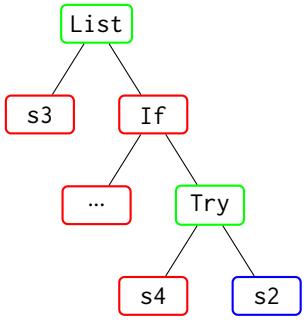


Fig. 5. AST of the program in Figure 1d. Node color represents from which version it is derived: red for *left*, blue for *right*, and green for both.

to 2. Similar merge operations are performed and the merged VSA $\widetilde{S}$ becomes:

$$\begin{aligned}
\widetilde{S} &= \widetilde{S^L} \cup \widetilde{S^R} = \texttt{List}_{\bowtie}(\widetilde{A}) \\
\widetilde{A} &= \widetilde{A^L} \cup \widetilde{A^R} = \{\texttt{s3}\} \cup \texttt{If}_{\bowtie}(_, \widetilde{B}) \cup \texttt{Try}_{\bowtie}(\widetilde{C_1}, \widetilde{C_2}) \\
\widetilde{B} &= \texttt{Try}_{\bowtie}(\widetilde{C_1}, \widetilde{C_2}) \\
\widetilde{C_1} &= \widetilde{C_1^L} \cup \widetilde{C_1^R} = \{\texttt{s4}\} \cup \{\texttt{s1}\} \\
\widetilde{C_2} &= \widetilde{C_2^L} \cup \widetilde{C_2^R} = \{\varepsilon\} \cup \{\texttt{s2}\}
\end{aligned}$$

Figure 4 shows the VSA $\widetilde{S}$ by merging $\widetilde{C_1^L}$ with $\widetilde{C_1^R}$. We observe that the program in Figure 1d is included in $\widetilde{S}$. Figure 5 illustrates how this program is linked to l and r. We draw nodes in different colors to distinguish their sources. A node in red, blue or green color means this node is derived from *left*, *right* or both, respectively. Remark that the program in Figure 1d is composed of nodes from l and r, but itself is a distinct program from l and r.

4.2.4 Algorithm. With the above mechanisms and results, we now formally present our algorithm for constructing the VSA that represents all candidate programs.

The algorithm (Algorithm 2) consists of two procedures. Procedure ConstructVSA takes a hole $\texttt{Hole}(b, l, r)$ as input, and builds the target VSA $\widetilde{S}$ by recursively visiting the left version l and the right version r. To represent the program space of $\widetilde{S}$ succinctly, we maintain intermediate VSAs (including $\widetilde{S}$ itself), which are necessary to construct $\widetilde{S}$, in a map σ : Identifier $\rightarrow$ VSA. Every VSA is labeled with an *identifier* and is accessed by this identifier. As a convention, we denote the VSA labeled with identifier N as $\widetilde{N}$, i.e., $\widetilde{N} = \sigma(N)$. This convention yields a property that two VSAs $\widetilde{N_1}$ and $\widetilde{N_2}$ are equivalent if their identifies N_1 and N_2 are identical.

Procedure Visit takes three inputs: an AST t that is currently being visited, the search depth d, and an identifier N such that the corresponding intermediate VSA $\widetilde{N}$ is yet to be grown. If the identifier N is not yet occurred in the map σ, then we add it and initialize $\widetilde{N}$ with $\varnothing$. We name the identifier N as the *context* of visiting AST t. We set a threshold D denoting the maximal search depth. When D is reached (line 6), AST t is regarded as a leaf node, which will not be recursively visited any more.

Algorithm 2 VSA Construction

```
1:  procedure ConstructVSA(Hole(b, l, r))
2:      Visit(l, 1, S);
3:      Visit(r, 1, S);
4:      return S̃;
5:  procedure Visit(t, d, N)
6:      if d ≥ D then Ñ ← Ñ ∪ {t};
7:      else
8:          match t
9:              case V then Ñ ← Ñ ∪ {V};                          ▷ t is a leaf
10:             case F(N_1, N_2, ..., N_k) then                   ▷ t is a constructed node
11:                 for i = 1 to k do
12:                     V_i ← f(F, i, N);                         ▷ mapper f returns an identifier
13:                     Visit(N_i, d + 1, V_i);
14:                 Ñ ← Ñ ∪ F_⋈(Ṽ_1, Ṽ_2, ..., Ṽ_k);
15:             case List(N_1, N_2, ..., N_k) then                ▷ t is an (ordered or unordered) list
16:                 for i = 1 to k do Visit(N_i, d, V_N);
17:                 Ñ ← Ñ ∪ List_⋈(Ṽ_N);
```

Algorithm 2 combines the mechanisms we have proposed in Section 4.2.2 and Section 4.2.3, by observing the following two facts:

(1) Let VSA $\widetilde{N} = \varnothing$. After executing Visit$(t, 1, N)$ for an AST t, without the limitation of the maximal search depth, $\widetilde{N} = \alpha(t)$.
(2) Let l and r be two ASTs. ConstructVSA(Hole(_, l, r)) returns a VSA $\widetilde{S}$ that is the merged VSA of $\{l\}$ and $\{r\}$.

Fact (1) indicates that the procedure Visit implements the conversion operation α we have defined in Section 4.2.2. It is immediately shown by comparing line 9 with A-EXP, line 14 with A-JOIN and lines 16 – 17 with A-LISTJOIN. Note that in the third case (lines 15 – 17), an ordered (unordered) list join node will be created if the `List` node is ordered (unordered).

Fact (2) points out that by recursively visiting l (line 2) and r (line 3) in the same context S, we retrieve a VSA $\widetilde{S}$ that merges $\{l\}$ with $\{r\}$. As the example shown in 4.2.3, we merge $\widetilde{C_1^L} = \{\texttt{s4}\}$ with $\widetilde{C_1^R} = \{\texttt{s1}\}$ and the merged result is denoted by a new VSA node $\widetilde{C_1} = \widetilde{C_1^L} \cup \widetilde{C_1^R} = \{\texttt{s4}, \texttt{s1}\}$. This can be done by specifying the same context C_1 for both $\widetilde{C_1^L}$ and $\widetilde{C_1^R}$. In this way, when Visit$(l, 1, S)$ is done, `s4` is appended to $\widetilde{C_1}$. When Visit$(r, 1, S)$ is done, `s1` is also appended to $\widetilde{C_1}$. Therefore, $\widetilde{C_1}$ becomes $\{\texttt{s4}, \texttt{s1}\}$, as expected.

A merge operation must concern about which VSAs – usually those that are corresponding to the argument nodes (i.e. the nodes $N_1, N_2, \ldots, N_k$ listed at line 10) – shall be shared. We thus define a special *mapper* f that aims to specify the sharing "rules". A mapper is a function that maps an argument node to an identifier. We share two VSAs, each of which is corresponding to an argument node, by mapping these two argument nodes to the same identifier. If this is the case, we say that the two argument nodes are *indistinguishable*, otherwise *distinguishable*. The argument node N_i is depicted using the following three parameters:

- the type of the constructor (F),
- the index of the argument (i), and

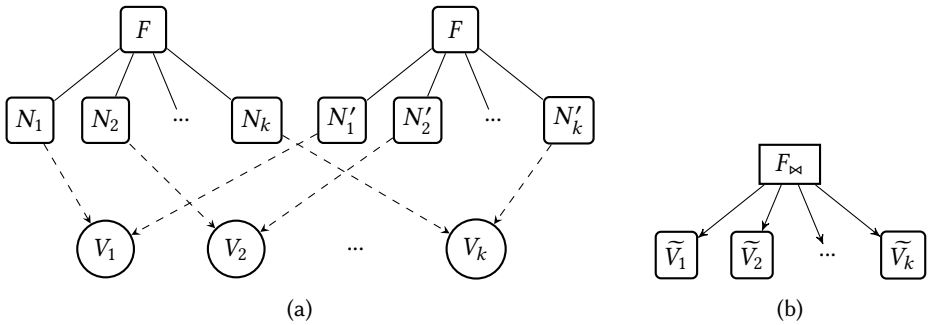


Fig. 6. Direct mapper. (a) For two constructed nodes $F(N_1, N_2, \ldots, N_k), F(N'_1, N'_2, \ldots, N'_k)$ with a common constructor F, the direct mapper maps N_i and N'_i to the same identifier V_i, for $i = 1, 2, \ldots, k$, as shown by the dashed arrows. (b) The join node composed of VSAs $\widetilde{V_1}, \widetilde{V_2}, \cdots, \widetilde{V_k}$.

- the context (N) in which the mapper is invoked.

We allow the users to employ both pre-defined mappers (introduced later) and customized mappers (created by the users), or a composition of them.

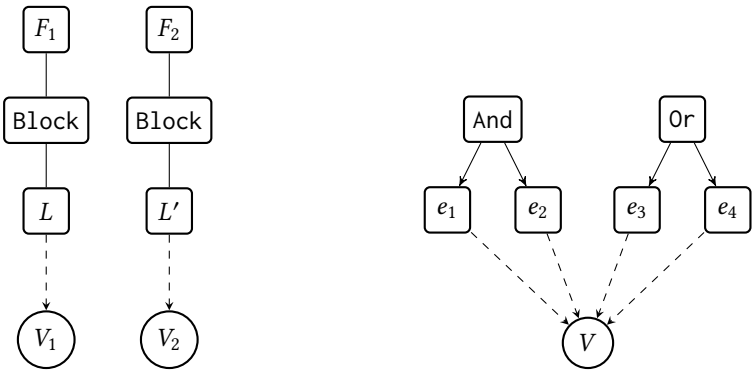
Recall that the condition of sharing $\widetilde{C_1^L} = \{\texttt{s4}\}$ with $\widetilde{C_1^R} = \{\texttt{s1}\}$ is that both `s4` and `s1` are try-blocks. We generalize it as a sharing "rule": two argument nodes shall be shared if they have the identical constructor and index. This "rule" can be formally defined as a *direct mapper* f_1 such that $f_1(F_1, i_1, N_1) = f_1(F_2, i_2, N_2)$ if and only if $F_1 = F_2 \wedge i_1 = i_2$. As described in Figure 6a, any two argument nodes N_i and N'_i are mapped to the identical identifier V_i, if they have the same constructor F and index i. Figure 6b shows the join node $F_{\bowtie}(\widetilde{V_1}, \widetilde{V_2}, \ldots, \widetilde{V_k})$ composed of k VSAs $\widetilde{V_i}$ where $\widetilde{V_i} = \sigma(V_i)$.

With the direct mapper, our VSA construction algorithm is capable of merging VSAs from l and r, as $\widetilde{C_1}$ consists of both `s4` (from l) and `s1` (from r). We take the catch-block of the try-statement as another example. Suppose that the direct mapper returns identifier C_2 for all argument nodes whose constructor $F = \texttt{Try}$ and index $i = 2$ (catch-block is the 2nd argument of a `Try` node). When $\texttt{Try}(\texttt{s4}, \varepsilon)$ (from l) is visited, $\textsc{Visit}(\varepsilon, 2, C_1)$ is invoked (at line 13) and ε will be appended to $\widetilde{C_2}$. When $\texttt{Try}(\texttt{s1}, \texttt{s2})$ (from r) is visited, $\textsc{Visit}(\texttt{s2}, 3, C_1)$ is invoked (at line 13) and `s2` will be appended to $\widetilde{C_2}$. Therefore, $\widetilde{C_2} = \{\varepsilon, \texttt{s2}\}$. Combined with $\widetilde{C_1} = \{\texttt{s4}, \texttt{s1}\}$, the program space of a try-statement is represented by a join node $\widetilde{B} = \texttt{Try}_{\bowtie}(\widetilde{C_1}, \widetilde{C_2})$ (at line 14), with $\left|\widetilde{B}\right| = 2 \times 2 = 4$. We see that the program space is enlarged.

After completely applying Algorithm 2 to the motivating example, we finally get a VSA $\widetilde{S}$ as:

$$\widetilde{S} = \texttt{List}_{\bowtie}(A) \tag{1}$$
$$\widetilde{A} = \{\texttt{s3}\} \cup \texttt{If}_{\bowtie}(\_, \widetilde{B}) \cup \texttt{Try}_{\bowtie}(\widetilde{C_1}, \widetilde{C_2}) \tag{2}$$
$$\widetilde{B} = \texttt{Try}_{\bowtie}(\widetilde{C_1}, \widetilde{C_2}) \tag{3}$$
$$\widetilde{C_1} = \{\texttt{s4}\} \cup \{\texttt{s1}\} \tag{4}$$
$$\widetilde{C_2} = \{\varepsilon\} \cup \{\texttt{s2}\} \tag{5}$$

Note that $\widetilde{S}$ is identical to the one created in Section 4.2.3.



(a) Block mapper (b) Expression mapper

Fig. 7. Block mapper and expression mapper, where dashed arrows show the mapping function.

Other Mappers. Selection of distinct mappers has an impact on the constructed VSA and thus effects the expressiveness of the generated program space. We now introduce two other pre-defined mappers.

Sometimes, argument nodes constructed by an identical constructor but occur in different levels of the AST may be distinguishable. For example, to distinguish blocks nested in distinct constructors (for instance, the for-loop and the while-loop), we define a *block mapper* f_2 which maps these blocks to different identifiers. Formally, $f_2(\texttt{Block}, i_1, N_1) = f_2(\texttt{Block}, i_2, N_2)$ if and only if $i_1 = i_2 \wedge N_1 = N_2$. As shown in Figure 7a, two lists L and L' are mapped to two distinct identifiers V_1 and V_2, as the constructors F_1 and F_2 are distinct (e.g. $F_1 = \texttt{For}$ but $F_2 = \texttt{While}$), i.e. the context N_1 and N_2 are distinct. Using the block mapper, the constructed VSA is restricted.

On the contrary, nodes with distinct types and indexes may be indistinguishable. For a logical expression $\texttt{And}(e_1, e_2)$ appeared in the left version, and another expression $\texttt{Or}(e_3, e_4)$ appeared in the right version, the expression $\texttt{Or}(e_4, e_1)$ is not included in the program space using the direct mapper. To include this expression, we attempt not to distinguish between And and Or, as well as their indexes. As illustrated in Figure 7b, all of these arguments are mapped to a unified identifier V. Now, we can represent any logical expression connected with And or Or by using atomic expressions $\{e_1, e_2, e_3, e_4\}$. We thus define an *expression mapper* f_3 which maps a number of different expression constructors with all arguments to a unified identifier. Formally, if $t_1, t_2 \in \{\texttt{And}, \texttt{Or}\}$, then $f_3(t_1, \_, \_) = f_3(t_2, \_, \_)$ is always held. In this way, the constructed VSA is expanded.

4.3 Resolution Ranking

A VSA represents a large set of programs. Enumerating the entire set of programs is apparently impractical. From the viewpoint of developers, they expect the target result to be figured out as early as possible. A ranking function for the candidate resolutions is thus desired. To some extent, this ranking function determines the usability of our VSA-based approach.

Generally speaking, a *ranking function* assigns a score (the value is usually a real number) to each candidate program. Explicitly specifying a score for each program is tedious. In fact, a *prior to* relation between any two programs is sufficient for program ranking. We say that P_1 is *prior to* P_2, denoted by $P_1 \prec P_2$, if and only if P_1 is more likely to be the target result than P_2.

In particular, we lift the $\prec$ relation to VSAs: $\widetilde{N_1}$ is prior to $\widetilde{N_2}$ if all programs in $\widetilde{N_1}$ are prior to those in $\widetilde{N_2}$. Formally, $\widetilde{N_1} \prec \widetilde{N_2}$ if $\forall P_1 \in \widetilde{N_1}, P_2 \in \widetilde{N_2} : P_1 \prec P_2$. The technique above implies that

the priority over programs are induced from the priority over VSAs. Supposing that $\widetilde{A} = \widetilde{N_1} \cup \widetilde{N_2}$ and $\widetilde{N_1}$ is prior to $\widetilde{N_2}$, then programs in $\widetilde{N_1}$ have higher priority to those in $\widetilde{N_2}$. When enumerating programs from $\widetilde{A}$, we first consider those that come from $\widetilde{N_1}$ and then $\widetilde{N_2}$.

This ranking method has been widely adopted in the area of programming by examples [Gulwani 2011; Le and Gulwani 2014; Rolim et al. 2017]. For instance, Gulwani [2011] specifies a string processing domain-specific language. It identifies the position inside a string by either the absolute (constant) index, or the relative index specified with regular expressions, denoted by the grammar: position $P ::= \texttt{CPos}(k) \mid \texttt{Pos}(r_1, r_2, c)$, where k is an integer constant, c is an integer expression, r_1 and r_2 are regular expressions. Here, CPos is the absolute position constructor, and Pos is the relative position constructor. Let $\widetilde{P_{\text{const}}}$ denote all programs constructed with CPos and $\widetilde{P_{\text{relative}}}$ denote all programs constructed with Pos. Since Pos is more general and expressive than the CPos, then we let $\widetilde{P_{\text{relative}}}$ be prior to $\widetilde{P_{\text{const}}}$. Therefore, programs that construct positions with Pos have higher priority than those with CPos.

In our VSA construction algorithm, VSAs are distinguished by their identifiers. For convenience, we define the *prior to* relation on VSA identifiers. Given two identifiers V_1, V_2, we define the partial relation motivated by the basic rules of three-way merge. Suppose that V_1 occurs in both the base and the left version, but not in the right version, then by 3rd row of Table 1, it is deleted in the target version. Besides, suppose that V_2 only presents in left version, but not in the base and the right version, then by 4th row of Table 1, it is inserted in the target version. Since V_1 is deleted and V_2 is inserted in the target version, we conclude that V_2 is preferred to V_1.

Given an identifier V, we use $S_V \subseteq \{\mathsf{B}, \mathsf{L}, \mathsf{R}\}$ to express the *source set* of V. $\mathsf{B} \in S_V$ if and only if V occurs in the base version. The same case for $\mathsf{L} \in S_V$ and $\mathsf{R} \in S_V$. Given two identifiers V_1, V_2, we define $V_1 \prec V_2$ if

$$(S_{V_1} = \{\mathsf{L}\} \wedge \mathsf{B} \in S_{V_2} \wedge \mathsf{R} \in S_{V_2}) \vee (S_{V_1} = \{\mathsf{R}\} \wedge \mathsf{B} \in S_{V_2} \wedge \mathsf{L} \in S_{V_2}) \vee (S_{V_1} = \{\mathsf{L}, \mathsf{R}\} \wedge \mathsf{B} \in S_{V_2}).$$

Consider the VSA $\widetilde{S}$ of the motivating example (Section 4.2.4). In equation (2), statement s3 and the if-statement only appear in *left*, while the try-block occur in both *base* and *right*. Therefore, the statement s3 and the if-statement are prior to the try-block. In equation (4), the statement s4 is only presented in the *left*, whereas the statement s1 is presented in both *base* and *right*. For the same reason, the statement s4 is prior to s1. In equation (5), the statement s4 only appears in *right*, while the empty statement occur in both *base* and *left*. Consequently, the statement s2 is prior to the empty statement. In this way, the program in Figure 1d (or equivalently, AST in Figure 5) is ranked to the top.

4.4 Discussion

With Algorithm 2, the merged VSA is composed of nodes merely from the left VSA and the right VSA. This implies an assumption by our approach that the resolution is always a combination of the left and the right versions. This assumption is reasonable and holds for most of the practical cases, as evidenced by our evaluation results (in Section 6).

If a conflict cannot be resolved by merging the left and right versions, the developer rejects all candidate programs in the merged VSA. Then we need to consider more program constructors and form more candidate programs. One possible solution is to "enlarge" the hole, i.e., treating the hole as a triple (b, l, r), where b, l, r are subtrees of ASTs, we lift the root node of these subtrees to their parents. In this way, we get three larger subtrees, with which much more candidate programs can be composed. The hole enlarge procedure continues until the expected program is returned, or the AST root of the program is reached.

5 IMPLEMENTATION

`AutoMerge` is based on the open-source structured merge tool `JDime`, release version 0.4.3. It is written in a combination of Java and Scala, and consists of a merge module and a conflict resolution module. The merge module is built on the top of `JDime` and we reuse the Java parser (built with `extendj`), AST node definitions, tree matching algorithms and merge operators of `JDime`. To better integrate with the conflict resolution module, we have made the following improvements:

- When creating a conflict, the base version with respect to the left and right versions is also recorded, so that our ranking strategy based on three-way merge is possible to be applied.
- We have implemented a more precise ordered merge algorithm, which can generate conflicting sections with more than one statement, whereas the original implementation produces conflicts with only one statement.

The conflict resolution module is built from bottom:

- First, we construct a VSA based on Algorithm 2. Additionally, when we visit ASTs, we label VSA identifiers with sources, say `L` (from left version) or `R` (from right version). The AST of the base version is also visited to label identifiers which appear in base version with `B` (from base version). The source sets are later used by the ranking function. As for the mappers, by default we apply the direct mapper and the block mapper.
- Second, we rank VSAs following the method proposed in Section 4.3, with the help of source sets.
- Finally, we enumerate candidates from the top-ranked to the bottom-ranked. We achieve this in an efficient way by presenting candidate programs as a lazy stream (supported by the stream collection) and each time we take the head element.

6 EVALUATION

To measure the performance of `AutoMerge` on real-world software projects, we conduct experiments that are designed to answer the following two research questions:

- RQ1: How effective and efficient is `AutoMerge` at resolving conflicts?
- RQ2: How do various mappers affect the effectiveness and efficiency of our approach?

We first describe our experimental setup in Section 6.1. Then, we answer these two research questions by inspecting evaluation results in Section 6.2 and Section 6.3. Finally, we discuss some major threats to the validity of the results in Section 6.4.

6.1 Experimental Setup

Merge Scenarios Extraction. Extracting a variety of proper merge scenarios from software projects is an important issue. To avoid the severe problem that manually created cases are biased, we prepare our benchmark suite by extracting merge scenarios from a substantial number of open-source projects developed for real-world applications.

We describe our merge scenarios extraction method in detail:

(1) We select 10 open-source projects with high stars from Github and analyze their commit histories (git logs).
(2) Based on the histories, we are able to obtain the merge commits that have been performed by the developers, together with the base commit and two parent commits. We treat the base commit as the base version, and two parent commits as the left and right versions. In fact, the three versions form a merge scenario. Additionally, the merge commit itself is marked as the *expected* version with respect to the merge scenario.

Table 2. Summary of extracted merge scenarios. Conf. commits: number of conflicting merge commits.

Project	Conf. commits	Description
auto	1	A collection of source code generators for Java.
drjava	2	A lightweight programming environment for Java.
error-prone	6	Catch common Java mistakes as compile-time errors.
fastjson	6	A fast JSON parser/generator for Java.
freecol	4	A turn-based strategy game.
itextpdf	47	Core Java Library + PDF/A, xtra and XML Worker.
jsoup	2	Java HTML Parser, with best of DOM, CSS, and jquery.
junit4	21	A programmer-oriented testing framework for Java.
RxJava	1	Reactive Extensions for the JVM.
vert.x	5	A tool-kit for building reactive applications on the JVM.

(3) Not all of these merge commits are interesting. Some of them can be fully merged by the structured merger `JDime`, and thus it is unnecessary to apply our conflict resolution approach on these merge scenarios. Instead, we only concentrate on merge commits that cannot be fully merged by `JDime`, say at least one conflict exhibits. Our benchmark suite comprises these conflicting merge commits, cataloged in Table 2. There are totally 95 conflicting merge commits, each regarded as a merge scenario.

Correctness Checking. Lines of merged code is a common criterion [Apel et al. 2011; Leßenich et al. 2017, 2015] to measure the performance of a merge tool. However, the criterion is inappropriate for our VSA-based approach, for the reason that every conflict can be resolved by arbitrarily choosing one candidate resolution as the merged result. A more rational criterion is to regard the merged version, which is performed manually by the developers, as the "ground truth". Noting that we have extracted the merged versions as the expected versions, we check if the expected version is included in the candidate program space. To be more precise, we use k to measure the performance of `AutoMerge` at resolving conflicts, where k is the smallest number such that the expected result is ranked within the top k programs.

Environment. All of our experiments are conducted on an Intel(R) Core(TM) computer with i7-7700 CPU (3.60 GHz) and 16 GB memory on Ubuntu 16.04.

6.2 Evaluation on Conflict Resolution (RQ1)

We evaluate `AutoMerge` on 10 open-source projects cataloged in Table 2. We enable the direct mapper and the block mapper by default. Our evaluation method is straightforward: we first apply our approach to every merge scenario and for every hole, we generate a program space of candidate resolutions, then enumerate candidates from the top-ranked to the bottom-ranked and compare them with the expected version. If any candidate hits, say it has no difference from the expected version syntactically, then we terminate and record the number of steps we have used so far as k. In case the program space is very large, we enumerate at most 50 programs for each hole. If the top 50 candidates all miss, then it is a failure.

Table 3 depicts the results. At a glance, 244 conflicts are detected spread over 138 files. These conflicts cannot be resolved by `JDime` and thus our conflict resolution technique can be put to good use. It correctly resolves 232 holes, i.e., a resolution rate as high as 95.1%. The enumeration converges quickly within an average step $k = 1.79$, and the maximal $k = 18$. Our implementation

Table 3. Evaluation results on conflict resolution. Time (last column) presents the execution time of conflict resolution (excluding merge) per hole. Conf. files: number of conflicting files, k: search steps for the expected resolution, P.S.: size of program space per hole.

Project	Conf. files	Holes	Resolved holes	Max. k	Avg. k	P.S.	Time (ms)
auto	4	11	10 (90.9%)	2	1.18	191.1	94.72
drjava	2	2	2 (100%)	2	1.50	515	297.50
error-prone	8	13	8 (61.5%)	13	4.62	6.31	146.46
fastjson	8	19	19 (100%)	18	2.37	8.37	119.16
freecol	22	57	57 (100%)	2	1.81	23.9	87.91
itextpdf	47	47	47 (100%)	1	1.00	6	231.94
jsoup	2	2	2 (100%)	1	1.00	6	116
junit4	33	51	45 (88.2%)	13	1.78	133	126.73
RxJava	1	1	1 (100%)	2	2.00	6	1
vert.x	11	41	41 (100%)	4	1.78	7.24	63.22
Overall	138	244	232 (95.1%)	18	1.79	48.88	127.10

of conflict resolution module is also efficient, with an average execution time of 127.10 ms. We later interpret the table carefully column by column.

Column *Project* shows project names. Column *Conf. files* exhibits the number of conflicting source files reported by the structured merger. There are totally 138 conflicting files found in 95 merge commits, in average 1.45 conflicting files per merge commit. The *Holes* column reveals the number of holes, one generated for each conflict. There are overall 244 holes, in average 1.77 per conflicting file, and 2.57 per merge commit. Conflicts are not uniformly distributed in the projects: *freecol* has 14.25 conflicts per merge commit, whereas *drjava* has only 1 per merge commit.

In particular, the performance of our approach is measured by columns *Resolved holes*, *Max. k* and *Avg. k*. Column *Resolved holes* presents the number of holes that have been resolved by `AutoMerge`. On 7 of the 10 projects, our approach succeeds in resolving all the conflicts. On projects *junit4* and *auto*, it resolves over 85% of the conflicts. It performs worst on the project *error-prone*: 61.5% of the conflicts are resolved. In total, 232 holes of 244 are successfully resolved: the resolution rate is as high as 95.1%. Moreover, columns *Max. k* and *Avg. k* describe the average and the maximal search steps k per hole, respectively. On 7 of the 10 projects, our approach figures out the expected resolution with $k \leq 4$. In all projects, $k = 18$ at most, and $k = 1.79$ in average. The project *error-prone* has the most average steps $k = 4.62$. On projects *itextpdf* and *jsoup*, $k = 1$ in average, which implies that the expected resolution is ranked to the top.

Ranking determines the usability of our approach. We demonstrate it by presenting the average size of the program space constructed by our approach for each hole in column *P.S.* Assume that the program space size is N and the expected resolution is included, then in order to figure it out without any ranking technique, we have to enumerate all N programs in the worst case. Results show that the average program space size is over 48, compared with an average steps of merely 1.79. Moreover, on 3 of the 10 projects, the average program space is larger than 100.

Column *Time* records the average execution time of conflict resolution (excluding merge) per hole. Results indicate that our implementation is efficient, with an average execution time of 127.10 ms. Moreover, all cases take no more than 1 second.

Failure Cases. For practicability, we limit our approach to enumerate up to 50 candidate resolutions for each conflict. In 244 holes found, we fail to find the expected resolution in the top 50 candidates

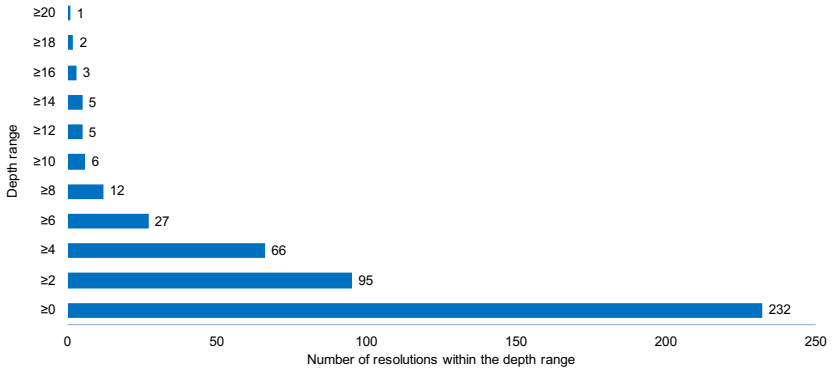


Fig. 8. Complexity of resolutions measured by the depths of the merged AST. The stripe annotated with $\geq d$ shows the number of resolutions with a depth greater or equal to d.

for 12 holes. Note that we are using the real merge commits of these projects in Github repositories as the expected resolutions. After manually inspecting these failure cases, we summarize 3 reasons that cause failure:

- assumption violation (8 cases). We have mentioned an assumption that the expected resolution is always a combination of the left and right versions (see Section 4.4). If the expected resolution includes components that are missing in both versions, then our approach fails because the VSA we constructed cannot express such a program. Unfortunately, it is impossible for us to tackle it since our approach has admitted the assumption. In our opinion, if the merged version introduces fairly new things, the only reasonable way is to manually resolve the conflict by developers.
- insufficient expressiveness (2 cases). The constructed VSA requires the root of the AST must have the identical kind with either the left or the right version. However, in these failure cases, we find the expected AST is actually a subtree of some candidate. It is possible for us to work it out by allowing the target AST to start from any VSA we have introduced. Nevertheless, we believe this will significantly enlarge the program space. Moreover, the results indicate that such situations are rarely seen in reality.
- huge program space (2 cases). If the generated hole is complicated, then the constructed VSA is also complex and the program space is huge. For these 2 cases, our ranking function is unable to rank the expected program within top 60. Without the limitation, it must be expensive to enumerate a significant number of candidate programs until the expected one is found.

Complexity of Resolutions. Figure 8 describes the complexity of the 232 resolutions found. We measure the complexity of the resolutions by the depths of their AST representations. We find that 27 of 232 (11.6%) resolutions have a depth of no less than 6, indicating that these resolutions are nontrivial. The most complex resolution has a depth of 20, with totally 123 nodes. We conclude that our mechanism is capable of resolving complicated conflicts and generating nontrivial resolutions.

Renaming Resolution. Identifying renaming is another important issue in software merging. It is not of our concern, but we find that our approach has the ability to resolve renaming conflicts similar to the following example[3]:

```
/* base */
if (...) {
  deserizer = parser.getConfig().getDeserializer(userType);
} else {
  ...
}

/* left */
deserizer = parser.getConfig().getDeserializer(userType);

/* right */
if (...) {
  deserializer = parser.getConfig().getDeserializer(userType);
} else {
  ...
}

/* expected */
deserializer = parser.getConfig().getDeserializer(userType);
```

We only show the conflicting sections in the left, right, base and the expected version in order, labelled with comments.

Note that it is a conflict because in the left version, the if-statement has been changed into an assignment statement; in the right version, variable `deserizer` has been renamed to `deserializer`. Our approach can find the expected version, where variable `deserizer` is renamed to `deserializer`, perceiving that `deserializer` takes place (as the left value of the assignment statement) in the right version.

6.3 Evaluation on Configurations (RQ2)

To answer RQ2, we evaluate the performance of our approach under 4 different configurations on the following 2 metrics:

- enable or disable the block mapper (M1),
- enable or disable the expression mapper (M2).

Our experiment is conducted on the 244 holes found in Section 6.2. We let the default configuration (c_1) be the baseline. Figure 9 shows the comparison between each configuration and the baseline. For every hole, if more search steps are needed, then we say it performs *better*; if less search steps are needed, then we say it performs *worse*; otherwise, it remains the *same*. Results show that no cases perform better than baseline. The red stripe represents the number of holes that perform worse than the baseline, and the others perform the same with baseline.

When comparing c_1 (enable M1) with c_3 (disable M1), we see that the block mapper (M1) has slight improvement as no cases become better but only one becomes worse. We find that it succeeds when using c_1 but fails when using c_3. We argue that the usage of the block mapper increases the number of successful cases, and improves the usability of our approach in practice.

[3]It is taken from the project *fastjson*, commit c4b6ed7, file `main/java/com/alibaba/fastjson/parser/deserializer/JavaBeanDeserializer.java`

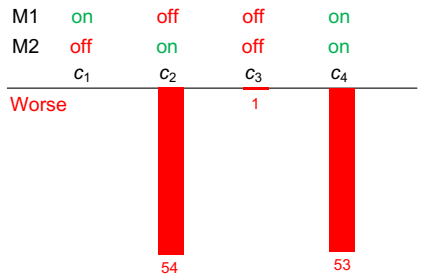


Fig. 9. Evaluation results on different configurations. A configuration is determined by enabling (on) / disabling (off) M1 and M2. Red stripe indicates the number of holes that perform worse than the baseline configuration c_1. No cases perform better, and the others keep unchanged.

The expression mapper (M2) significantly deteriorates the performance. Under configurations where the expression mapper is enabled (c_2 and c_4), more than 50 cases become worse. Originally, we intended to enhance the expressiveness of the VSA by this mapper so that more complicated resolutions can be discovered. However, in our benchmarks, such situations are not seen. Oppositely, it inflates the size of the program space.

6.4 Threats to Validity

Our benchmarks are extracted from a limited number of open-source projects. This is, however, not a threat to the validity of our approach. Firstly, we select the open-source projects over a wide variety of application domains, and hence the results give us a landscape of how often conflicts exhibit and how many of them can be resolved with our mechanism, in typical scenarios of real-world software evolvement. Secondly, our benchmark extraction method is unbiased and reproducible. For each project, all the merge commits are also evaluated in an unbiased way. Therefore, we believe that our approach is unbiased and applicable to other real-world projects.

`AutoMerge` currently supports the Java language only. However, it is applicable to other programming languages. Although different languages have different kinds of nodes in the abstract grammar tree, our approach can handle them uniformly with only three kinds of AST nodes (defined in Section 2.1): leaf node, constructed node and list nodes (ordered and unordered).

7 RELATED WORK

7.1 Software Merge

Syntactic Merge. Westfechtel [1991] and Buffenbarger [1995] have pioneered in proposing merge algorithms which incorporate context-free and context-sensitive structures. Their syntactic-oriented approaches are language-independent, since language features such as alternatives, lists, and structures are represented in an abstract level. Later, a variety of approaches on syntactic diff and merge have been proposed.

`JDime` [Leßenich et al. 2015] is a structured merge tool for Java programs. The main procedure is summarized as tree matching and amalgamation. During tree amalgamation, ordered and unordered nodes are distinguished and different algorithms are designed to improve precision. Based on structure merge, they also propose an auto-tuning approach, which switches between structured and unstructured merge depending on whether conflicts present. Based on `JDime`, we propose a VSA-based approach to resolve conflicts and enhance the overall merge precision.

Leßenich et al. [2017] propose a syntax-aware, heuristic optimization for `JDime` by introducing a looking ahead mechanism during tree matching. Their approach intends to handle shifted code and renaming uniformly. In `JDime`, matching is restricted to tree nodes at the same level, while the looking ahead mechanism allows matching to be performed at different levels. Inspecting the conflicts reported by structured merge in our benchmark, we find that conflicts caused by shifted code are successfully resolved using our VSA-based approach. Strictly speaking, the looking ahead mechanism is restricted to resolve special conditions of renaming where two nodes differs only in their names. For instance, two methods with distinct names have identical implementation. However, in this case, the underlying structured merge procedure fails to detect the conflict, and the VSA construction procedure will not be performed. Interestingly, we find that in some renaming scenarios, our approach succeeds in resolving the conflict, whereas the looking ahead mechanism fails. In such a way, our approach is capable of resolving a wider range of conflicts.

`JDiff` [Apiwattanapong et al. 2007] implements a differencing technique that handles object-oriented features, particularly for Java programs, such as modification of class and interface hierarchies and exception types. An augmented control-flow graph is employed to model behaviors caused by object-oriented features. Generally speaking, graph matching is infeasible compared with tree matching, thus it is not scalable for large projects.

Semistructured merge [Apel et al. 2011] is a trade-off of structured merge and unstructured merge, which inherits the generality of unstructured merge and precision of structured merge. In semistructured merge, only part of the programs are represented by abstract syntax trees and the rest are treated as plain text. In this way, semistructured merge is less precise than structured merge. That's why we prefer to perform structured merge before resolving conflicts. Compared to unstructured merge, semistructured merge reduces the number of conflicts in 60% of the sample merge scenarios by 34% on average.

Cavalcanti et al. [2017] compare merge results of several semistructured merge tools on more than 30,000 merges from 50 open-source projects, identifying conflicts incorrectly reported by only one of the tools (false positives), and conflicts correctly reported by only one of the tools (false negatives). Results show that the number of false positives is significantly reduced by semistructured merge, and they are easier to analyze and resolve than those reported by unstructured merge.

Operational-based Merge. Lippe and van Oosterom [1992] introduce the operation-based merging, whose core idea can be summarized by the following slogans invented by the authors: "merging = composition of operations from each development line" and "merge-conflict = commutation conflict". Operation-based merging is change-based, compared with the state-based syntactic approaches we have discussed above. Lippe and van Oosterom [1992] claim that operation-based approach offers more support for conflict resolution by given the example which a single global operation can be applied to a number of different points of the program. Operation-based approaches have been applied to conflict detection and resolution of UML models [Koegel et al. 2009].

Besides, tools are presented to merge software refactoring. Malpohl et al. [2003] attempt to automatically detect renamed identifiers across multiple files. When an identifier in one version is changed, then its references in other versions are actively updated. The technique above ensures that conflicts caused by renaming can be easily resolved during the merging phase. Dig et al. [2008] present `MolhadoRef` to merge software in the presence of object-oriented refactoring at the API level. It is also semantic-based and is concerned with operations which have well-defined semantics: inserting and deletion of packages, classes, method declarations and field declarations. Dotzler and Philippsen [2016] propose 5 general optimizations that can be integrated into tree differencing algorithms and decrease the number of resulting editing actions.

In our opinion, our VSA-based approach has the potential to solve conflicts caused by commutated operations. On the one hand, we can create a program space which represents possible permutations of operations and design a proper ranking function to select the expected results. On the other hand, it is possible that some operations have to be transformed before integrating with other operations. We can realize it by synthesizing a proper transformer, which is specified by a tree transformation DSL like `Refazer` [Rolim et al. 2017].

Buffenbarger [1995] argues that the practical scenarios of operation-based merging are limited to specialized development environment, where the required editing operations can be identified and recorded. Operational-based merge doesn't work in our merge scenarios, where the modification operations are absent.

Semantic Merge. Semantic-equivalent programs might look different syntactically, and thus syntactic merge may create fake conflicts, in which the left and right versions are actually equivalent in semantic. If we perform a semantic diff [Jackson and Ladd 1994] before merging, then conflicts caused by meaning-preserving transformations, such as renaming local variables, can be resolved.

Niu et al. [2013] propose a cost-benefit software conflict resolution recommender to guide the maintainer in resolving merge conflicts. It leverages the semantic information of the source code to better predict the cost of resolving conflicting software entities, and to estimate the benefit with the static structural dependencies. On conflict resolution, they concentrate on which conflicts are worth to resolve, rather than how to resolve them. We go a step further: we propose a method that resolves software conflicts automatically.

Semantic approaches have the ability to identify refactoring. Weissgerber and Diehl [2006] present a technique to detect changes generated by a syntactic and signature-based analysis that are likely to be refactoring, and rank them by likelihood. It is able to find structural refactoring and local refactoring. To identify more complex refactoring, Prete et al. [2010] propose a template-based approach to reconstruct complex refactoring using a logic programming engine to infer concrete refactoring instances.

7.2 VSA-based Program Synthesis

Program synthesis aims to find executable programs that accomplish a wide range of categories of user intents. *Programming by Example* (PBE) is a leading inductive synthesis technique which generates programs from input-output examples. It has the potential to revolutionize end-user programming [Gulwani et al. 2016] such as string processing [Gulwani 2011] and format normalization [Kini and Gulwani 2015] in spreadsheet, data extraction by highlighting texts on web pages [Le and Gulwani 2014], web automation by clicking a few buttons [Barman et al. 2016], and hierarchically structured data transformation [Yaghmazadeh et al. 2016].

VSA was first introduced by Mitchell [1982] and later expanded upon specific applications in program synthesis. Lau et al. [2000] present `SMARTedit`, a programming by demonstration application for repetitive text-editing. It shows the effectiveness in learning from only a small number of examples based on the notion of VSA. In PBE applications [Barman et al. 2016; Gulwani 2011; Kini and Gulwani 2015; Le and Gulwani 2014; Rolim et al. 2017; Yaghmazadeh et al. 2016], a *domain-specific language* (DSL) is specified to describes a rich space of candidate programs that commonly appear in practice and meet end users' intents. The program space defined by a practical DSL may have up to 10^{30} programs that are consistent with the given input-output example [Singh and Gulwani 2012], and therefore the application of VSA is critical thanks to its succinct memory usage.

Polozov and Gulwani [2015] propose a general PBE framework. First, a DSL is specified. It should be both expressive enough to meet the users' intent and efficient enough to figure out a result in a

reasonable time. Then, VSAs are created for different DSL operators, and constraints provided by the given input-output examples reduce the program space by efficiently operating on VSAs. Finally, functions and rules are defined to rank the synthesized programs based on the DSL structure.

Furthermore, Polozov and Gulwani [2015] remark that two kinds of representation for a set of programs – VSAs and formal languages (DSLs) – are isomorphic. In fact, the VSA constructed by our approach can be expressed with a strict subset of Java language, and the program space of the VSA is actually a set of programs that are defined by the language. We differ from studies on PBE mainly in two aspects: first, our language is automatically constructed by abstraction, while DSLs for PBE are manually specified; second, our design of the ranking function is motivated by the basic rule of three-way merge, while the ranking function is usually determined by the generality of the DSL constructors.

For general-purpose program synthesis or analysis, we usually take into account a strict subset of a general-purpose programming language, like Python [Singh et al. 2013], C [Long and Rinard 2015], or Scheme [Bornholt et al. 2016].

7.3 Program Transformation

`Refazer` [Rolim et al. 2017] is a framework that automatically learns program transformations from input-output examples. It leverages PBE methodology and specifies a program transformation DSL which contains filters and operators of AST nodes. Due to the generality of the technique, it is hard to evolve a reasonable conflicts resolution approach that employs the knowledge of three-way merge.

To the contrary, many studies focus on domain-specific program transformation. Feser et al. [2015] propose a method for example-guided synthesis of recursive data structure transformations in functional programming languages. Nguyen et al. [2010] present a graph-based technique that guides developers in adapting API usages. `HelpMeOut` [Hartmann et al. 2010] aids developers to debug compilation and run-time error messages by fixing the original broken code. However, none of them are suitable in the scenario of conflict resolution.

8 CONCLUSION

Recent studies and tools demonstrate the prospects of structured merge, which has a higher precision than the conventional unstructured merge. We address the problem that structured merge cannot resolve conflicts where concurrent changes contradict each other. In this case, we believe that it is valuable to provide the developers with a variety of useful candidate resolutions so that they can decide which one is suitable and applicable.

We present a VSA-based conflict resolution approach, motivated by collaborating modifications derived from the opposing versions. It is common that modifications can be collaborated in many ways and thus the resolution is ambiguous. We tackle the problem by first constructing a program space consists of possible resolutions and then discovering the expected result with a ranking mechanism. We succinctly represent the program space via version space algebra, which is capable of expressing cross products over a number of components with a memory sharing mechanism. We design a problem-specific ranking mechanism, integrating the knowledge of three-way merge.

We implement our approach as `AutoMerge`, built on the top of the structured merge tool `JDime`. We conduct experiments on 95 merge commits extracted from 10 real-world open-source projects from Github. `AutoMerge` detects 244 conflicts spread over 138 files, and successfully resolves as high as 95.1% of the conflicts. With our ranking technique, we only need to try 1.79 candidates in average until the expected resolution is found.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful comments. This material is based upon work supported in part by the National Natural Science Foundation of China under Grant No. 61672310 and Grant No. 61527812, the National Science and Technology Major Project under Grant No. 2016ZX01038101.

REFERENCES

Sven Apel, Jörg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kästner. 2011. Semistructured Merge: Rethinking Merge in Revision Control Systems. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, New York, NY, USA, 190–200. https://doi.org/10.1145/2025113.2025141

Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. 2007. JDiff: A differencing technique and tool for object-oriented programs. *Automated Software Engineering* 14, 1 (01 Mar 2007), 3–36. https://doi.org/10.1007/s10515-006-0002-0

Shaon Barman, Sarah Chasins, Rastislav Bodik, and Sumit Gulwani. 2016. Ringer: web automation by demonstration. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 748–764.

James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. 2016. Optimizing Synthesis with Metasketches. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, 775–788. https://doi.org/10.1145/2837614.2837666

Jim Buffenbarger. 1995. Syntactic software merging. In *Software Configuration Management*, Jacky Estublier (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 153–172.

Guilherme Cavalcanti, Paulo Borba, and Paola Accioly. 2017. Evaluating and Improving Semistructured Merge. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 59 (Oct. 2017), 27 pages. https://doi.org/10.1145/3133883

Danny Dig, Kashif Manzoor, Ralph E. Johnson, and Tien N. Nguyen. 2008. Effective Software Merging in the Presence of Object-Oriented Refactorings. *IEEE Trans. Softw. Eng.* 34, 3 (May 2008), 321–335. https://doi.org/10.1109/TSE.2008.29

Georg Dotzler and Michael Philippsen. 2016. Move-optimized Source Code Tree Differencing. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 660–671. https://doi.org/10.1145/2970276.2970315

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-output Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 229–239. https://doi.org/10.1145/2737924.2737977

Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, 317–330. https://doi.org/10.1145/1926385.1926423

Sumit Gulwani, Javier Esparza, Orna Grumberg, and Salomon Sickert. 2016. Programming by Examples (and its applications in Data Wrangling). *Verification and Synthesis of Correct and Secure Systems* (2016).

Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R. Klemmer. 2010. What Would Other Programmers Do: Suggesting Solutions to Error Messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. ACM, New York, NY, USA, 1019–1028. https://doi.org/10.1145/1753326.1753478

D. Jackson and D. A. Ladd. 1994. Semantic Diff: a tool for summarizing the effects of modifications. In *Proceedings 1994 International Conference on Software Maintenance*. 243–252. https://doi.org/10.1109/ICSM.1994.336770

Dileep Kini and Sumit Gulwani. 2015. FlashNormalize: Programming by examples for text normalization. In *Proceedings of the 24th International Conference on Artificial Intelligence*. AAAI Press, 776–783.

Maximilian Koegel, Jonas Helming, and Stephan Seyboth. 2009. Operation-based Conflict Detection and Resolution. In *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models (CVSM '09)*. IEEE Computer Society, Washington, DC, USA, 43–48. https://doi.org/10.1109/CVSM.2009.5071721

Tessa A. Lau, Pedro Domingos, and Daniel S. Weld. 2000. Version Space Algebra and Its Application to Programming by Demonstration. In *Proceedings of the Seventeenth International Conference on Machine Learning (ICML '00)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 527–534. http://dl.acm.org/citation.cfm?id=645529.657973

Vu Le and Sumit Gulwani. 2014. FlashExtract: A Framework for Data Extraction by Examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, 542–553. https://doi.org/10.1145/2594291.2594333

O. Leßenich, S. Apel, C. Kästner, G. Seibt, and J. Siegmund. 2017. Renaming and shifted code in structured merging: Looking ahead for precision and performance. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 543–553. https://doi.org/10.1109/ASE.2017.8115665

Olaf Leßenich, Sven Apel, and Christian Lengauer. 2015. Balancing precision and performance in structured merge. *Automated Software Engineering* 22, 3 (01 Sep 2015), 367–397. https://doi.org/10.1007/s10515-014-0151-5

Ernst Lippe and Norbert van Oosterom. 1992. Operation-based Merging. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments (SDE 5)*. ACM, New York, NY, USA, 78–87. https://doi.org/10.1145/142868.143753

Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 166–178.

Guido Malpohl, James J. Hunt, and Walter F. Tichy. 2003. Renaming Detection. *Automated Software Engineering* 10, 2 (01 Apr 2003), 183–202. https://doi.org/10.1023/A:1022968013020

T. Mens. 2002. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering* 28, 5 (May 2002), 449–462. https://doi.org/10.1109/TSE.2002.1000449

Tom M. Mitchell. 1982. Generalization as search. *Artificial Intelligence* 18, 2 (1982), 203 – 226. https://doi.org/10.1016/0004-3702(82)90040-6

Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson, Jr., Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. 2010. A Graph-based Approach to API Usage Adaptation. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. ACM, New York, NY, USA, 302–321. https://doi.org/10.1145/1869459.1869486

N. Niu, F. Yang, J. R. C. Cheng, and S. Reddivari. 2013. Conflict resolution support for parallel software development. *IET Software* 7, 1 (February 2013), 1–11. https://doi.org/10.1049/iet-sen.2012.0089

Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A framework for inductive program synthesis. *ACM SIGPLAN Notices* 50, 10 (2015), 107–126.

Kyle Prete, Napol Rachatasumrit, Nikita Sudan, and Miryung Kim. 2010. Template-based Reconstruction of Complex Refactorings. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM '10)*. IEEE Computer Society, Washington, DC, USA, 1–10. https://doi.org/10.1109/ICSM.2010.5609577

Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning Syntactic Program Transformations from Examples. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 404–415. https://doi.org/10.1109/ICSE.2017.44

Rishabh Singh and Sumit Gulwani. 2012. Learning Semantic String Transformations from Examples. *Proc. VLDB Endow.* 5, 8 (April 2012), 740–751. https://doi.org/10.14778/2212351.2212356

Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated Feedback Generation for Introductory Programming Assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, 15–26. https://doi.org/10.1145/2491956.2462195

P. Weissgerber and S. Diehl. 2006. Identifying Refactorings from Source-Code Changes. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. 231–240. https://doi.org/10.1109/ASE.2006.41

Bernhard Westfechtel. 1991. Structure-oriented Merging of Revisions of Software Documents. In *Proceedings of the 3rd International Workshop on Software Configuration Management (SCM '91)*. ACM, New York, NY, USA, 68–79. https://doi.org/10.1145/111062.111071

Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. 2016. Synthesizing Transformations on Hierarchically Structured Data. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, 508–521. https://doi.org/10.1145/2908080.2908088

Kaizhong Zhang and Tao Jiang. 1994. Some MAX SNP-hard Results Concerning Unordered Labeled Trees. *Inf. Process. Lett.* 49, 5 (March 1994), 249–254. https://doi.org/10.1016/0020-0190(94)90062-0

T. Zimmermann. 2007. Mining Workspace Updates in CVS. In *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*. 11–11. https://doi.org/10.1109/MSR.2007.22