# Semistructured Merge in JavaScript Systems

Alberto Trindade Tavares, Paulo Borba, Guilherme Cavalcanti, Sérgio Soares
Federal University of Pernambuco
{att, phmb, gjcc, scbs}@cin.ufpe.br

***Abstract*—Industry widely uses unstructured merge tools that rely on textual analysis to detect and resolve conflicts between code contributions. Semistructured merge tools go further by partially exploring the syntactic structure of code artifacts, and, as a consequence, obtaining significant merge accuracy gains for Java-like languages. To understand whether semistructured merge and the observed gains generalize to other kinds of languages, we implement two semistructured merge tools for JavaScript, and compare them to an unstructured tool. We find that current semistructured merge algorithms and frameworks are not directly applicable for scripting languages like JavaScript. By adapting the algorithms, and studying 10,345 merge scenarios from 50 JavaScript projects on GitHub, we find evidence that our JavaScript tools report fewer spurious conflicts than unstructured merge, without compromising the correctness of the merging process. The gains, however, are much smaller than the ones observed for Java-like languages, suggesting that semistructured merge advantages might be limited for languages that allow both commutative and non-commutative declarations at the same syntactic level.**

***Index Terms*—Collaborative development, Software merging, Semistructured merge, Version control systems, JavaScript.**

## I. Introduction

In a collaborative development setting, programmers often work on simultaneous tasks that involve common project artifacts (source code, build files, etc.). Consequently, when integrating changes from independent tasks, programmers might have to deal with conflicting changes. Such conflicts impair development productivity, because they may cause project delays by demanding substantial effort to understand and manually resolve conflicts [1]–[7].

To detect and automatically resolve conflicts, merge tools adopt approaches that differ in a number of dimensions (two-way versus three-way merging [8], state-based versus operation-based merging [9], etc.). They especially differ on how they represent the artifacts to be merged. Industry widely uses unstructured merge tools, which rely on a textual, line-based, representation [10], often leading to spurious conflicts (*false positives*) that waste developer effort to manually resolve them. Semistructured merge tools go further by partially leveraging syntactic information about the artifacts to be merged [11]. For languages such as Java, for instance, semistructured merge represents class, method, and field declarations as AST nodes, and merges them using tree matching and combination algorithms. Such trees, however, do not include full structural information. Method bodies, for example, are represented as plain text in leaf nodes. So, to merge bodies, a semistructured tool simply uses unstructured merge line-based algorithm.

Previous studies compare these merge approaches, showing that semistructured merge reports fewer conflicts than unstructured merge. Apel et al. [11] report an average reduction of 34% compared to unstructured merge. Cavalcanti et al. [12] replicate this study, finding an average reduction of 21% in the number of conflicts reported by semistructured merge. Considering only Java projects, the same authors [13] go further, and propose an improved semistructured merge tool, and show that it has significant advantages over unstructured merge: reduces by half the number of reported conflicts, reports no *false positives* in addition to the ones reported by unstructured merge, and has a slightly smaller number of *false negatives* (actual, non-reported, interference between the integrated changes).

While these results are expected to generalize to Java-like languages such as C#, there is little evidence that they also hold for other kinds of languages, including scripting languages such as JavaScript, PHP, and Ruby [14], [15]. It is also not clear whether FSTMerge [11], the underlying semistructured merge engine and framework used in the mentioned studies, could also be effectively instantiated to create merge tools for such languages. In this context, to clarify these issues and better understand the practical potential of semistructured merge and the FSTMerge engine, we experiment with a vanilla instantiation of this engine for JavaScript, and implement two JavaScript semistructured merge tools that adapt and extend FSTMerge. We then compare these tools to unstructured merge, which is language independent.

From the experience with the tool implementations, we find that current semistructured merge algorithms and frameworks are not directly applicable for scripting languages like JavaScript, which support statements at the same syntactic level of commutative and associative elements such as function declarations. In particular, a vanilla instantiation of FSTMerge for JavaScript yields an inaccurate merge tool that is inferior to unstructured merge and has no chance of practical adoption. FSTMerge was not designed to handle ordered (statements, for example) and unordered (function declarations, for example) program elements at the same syntactic level. However, by adapting the FSTMerge engine and the input grammar, we can build competitive semistructured merge tools for JavaScript. This result shows that the expected smaller effort for creating semistructured merge tools [11], one of the main advantages of semistructured merge over structured merge [16]–[19], might not be as small as expected, at least for scripting languages.

To evaluate these JavaScript merge tools, we analyze 10,345 merge scenarios from 50 JavaScript projects on GitHub. For each scenario, which consists of the three revisions involved in a three-way merge (*base*, *left*, and *right*), we replay the merge operation with the two semistructured merge tools we implemented, and compare the results with the ones generated by unstructured merge. We measure the frequency of *false positives* and *false negatives*. We find evidence that our JavaScript tools report fewer spurious conflicts than unstructured merge, without compromising the correctness of the merging process. The gains, however, are much smaller than the ones observed for Java-like languages, suggesting that semistructured merge advantages might be limited for languages that support both commutative and non-commutative declarations at the same syntactic level. We can then conclude that, to be effectively adopted in practice, semistructured JavaScript merge tools require substantial adaptation, improvements, and tuning.

## II. Semistructured Merge

Semistructured merge partially represents programs as trees (*program structure trees*), and uses algorithms that know how to merge nodes and their subtrees [11]. With FSTMerge, a semistructured merge tool for a given language can be created by annotating the language grammar. The annotations specify whether a language element should be represented as a node or as plain text in leaves. Elements represented as nodes in a program structure tree are three-way merged via a process called *superimposition* [20], which merges trees recursively, composing nodes that have the same name and type. Leaves are, by default, merged by traditional three-way unstructured merge.

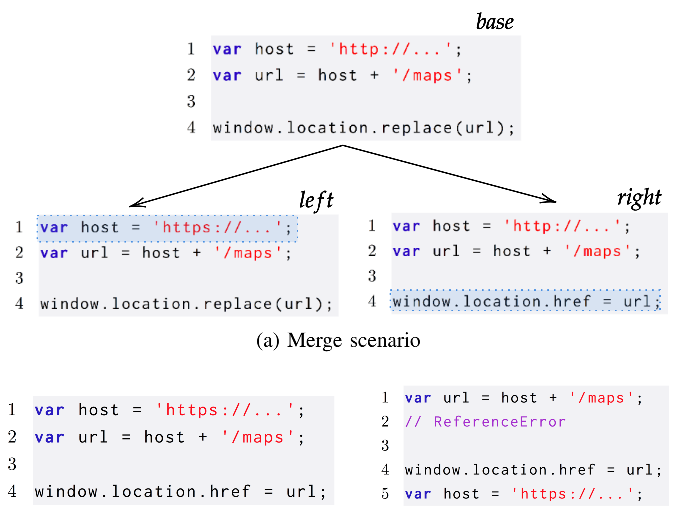
To simply instantiate FSTMerge for the 5th edition of ECMAScript (ES5),[1] we decided to annotate its off-the-shelf grammar [21], partially shown in the following.

| | | |
|---|---|---|
| $\langle \mathit{Program} \rangle$ | ::= | $\langle \mathit{SourceElement} \rangle$* |
| $\langle \mathit{SourceElement} \rangle$ | ::= | $\langle \mathit{Statement} \rangle$ |
| | \| | $\langle \mathit{FunctionDeclaration} \rangle$ |

It defines a program as a sequence of statements and/or function declarations. First we decided to represent function declarations as nodes because FSTMerge assumes that the order of nodes in a program structure tree is arbitrary [11], and, in JavaScript, the order of function declarations does not matter as a result of hoisting.[2] Semistructured merge can automatically solve the so-called *ordering conflicts* (like when two developers declare different functions in the same area of the program text) based on this observation that the order of some elements may change without affecting the semantics of the program. Furthermore, we can specify that function declarations are non-terminal nodes, as function declarations may contain other function declarations.

[1]Chosen for popularity reasons, and for better representing scripting languages.

[2] Hoisting is a JavaScript mechanism that loads variable and function declarations into the memory prior to a program execution, allowing a function to be used before its declaration [22].



```
var host = 'http://...';
var url = host + '/maps';

window.location.replace(url);
```

```
var host = 'https://...';
var url = host + '/maps';

window.location.replace(url);
```

```
var host = 'http://...';
var url = host + '/maps';

window.location.href = url;
```

(a) Merge scenario

```
var host = 'https://...';
var url = host + '/maps';

window.location.href = url;
```

(b) Unstructured merge

```
var url = host + '/maps';
// ReferenceError

window.location.href = url;
var host = 'https://...';
```

(c) Semistructured merge using TOSTRING name for statements

Fig. 1. Example of merge scenario and respective merges produced by unstructured and semistructured (based on an off-the-shelf grammar with TOSTRING names for statements) tools

Concerning statements, we initially annotate them as terminal nodes, implying that their content is represented as plain text in a leaf. In each element annotation, we can specify how the corresponding nodes should be named, so that they can be properly matched during the merging process. While function declarations have a natural name (the function identifier), statements do not [23], [24]. For instance, there is no general way to properly name an *IfStatement* in such a manner that the corresponding element in a different revision, with potential changes, can be accurately matched. By default, all nodes are named with the following constant "-". An alternative for statements is to name them with its textual representation, specified by TOSTRING in an annotation.

In either case— using default or TOSTRING names— annotating individual statements as terminal nodes compromises the correctness of merged revisions produced by superimposition. Consider the three-way merge scenario presented in Figure 1a. As shown in Figure 1b, unstructured merge produces the desired output, integrating independent contributions from each developer, and changing the first and third statements. Contrasting, semistructured merge, representing statements as nodes, does not produce a valid merge.

Using TOSTRING naming for statements makes semistructured merge too sensitive to minor changes, failing to match statements when one of them is edited by a revision. The consequence is that any statement edited by a revision, but not by the other, is interpreted as a statement deletion (with the old content) followed by a statement addition (with the new content). Issues arise as superimposition does not guarantee the order of new nodes. Differently from function declarations, which are associative and commutative, the order of statements is relevant, once they may cause side effects. An example of this problem can be seen in the generated merge in Figure 1c. The declaration of the variable *host* was moved to after its usage in Line 1, causing a *ReferenceError* during runtime. Additionally, if both derived revisions add changes to the same statement, instead of reporting a conflict— the desired result— semistructured merge interprets this as deletion of the statement by both revisions, and addition of new statements.

Using the default name for statements, in turn, would cause superimposition to mistakenly match nodes that do not refer to the same statement. The matching between terminal nodes that do not refer to the same statement from the base revision launches unstructured merge that reports spurious conflicts.

This shows that, for any three-way merge scenario that involves JavaScript programs with statements changed by both left and right revisions, a vanilla instantiation of FSTMERGE for JavaScript will either report spurious conflicts or produce an invalid merge. In Section V, we better illustrate how inefficient is a vanilla implementation of FSTMERGE for JavaScript. Considering those issues, we opted for not including such vanilla instantiations in our empirical study; they are too far from being reliable as a replacement for commercial unstructured tools. Instead, we implement and evaluate new semistructured tools. In these cases, we first change the grammar to better handle statements, already diverting from a vanilla instantiation that simply annotates an existing grammar in a plug-in fashion [11].

## III. PROPOSED SEMISTRUCTURED MERGE TOOLS

In order to overcome some of the problems described in the previous section, we first grouped consecutive statements into a *StatementList* element, as shown in the following excerpt of our modified grammar:

$$
\begin{array}{lcl}
\langle \textit{Program} \rangle & ::= & \langle \textit{SourceElement} \rangle^* \\
\langle \textit{SourceElement} \rangle & ::= & \langle \textit{StatementList} \rangle \\
 & | & \langle \textit{FunctionDeclaration} \rangle \\
\langle \textit{StatementList} \rangle & ::= & \langle \textit{Statement} \rangle + 
\end{array}
$$

This modified grammar is given as an input for FSTMERGE to support the two semistructured merge tools proposed in this paper: JSFSTMERGE V1 and JSFSTMERGE V2. In this grammar, *StatementList* is annotated as a terminal node, so statements are no longer individually represented as nodes in the program structure tree. Instead, there is only a single node for each contiguous sequence of statements separated by function declarations; that is, we have an opaque leaf with statements as textual content. The difference between JSFSTMERGE V1 and JSFSTMERGE V2 come from further adaptations to the FSTMERGE merge algorithm.

### A. *jsFSTMerge v1*

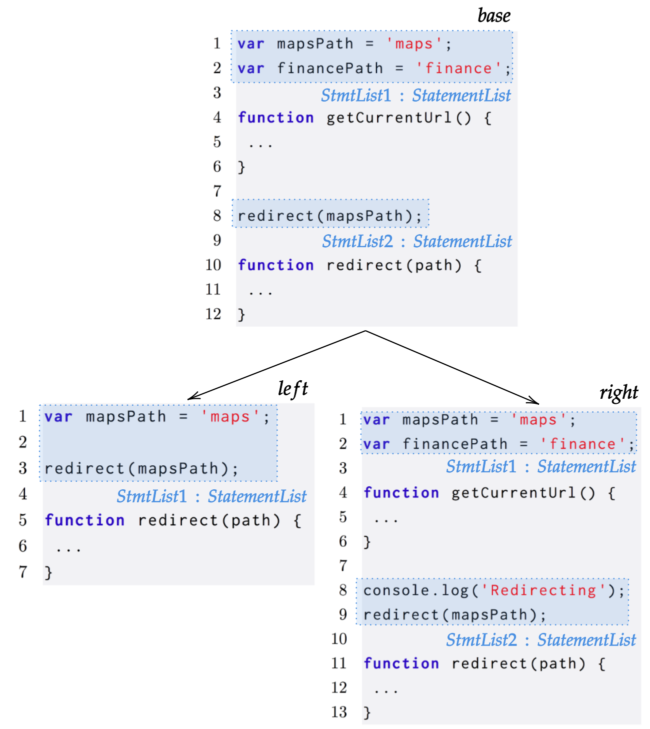
Our adapted grammar provides a more reasonable level of granularity by not representing most program elements as nodes in the tree. However, having default or TOSTRING names for statement lists still would cause matching issues as seen in the previous section. To ensure statement lists are properly handled in the superimposition we first changed the original FSTMERGE merge implementation to assign new names to statement lists, replacing the default one ("-"), according to their position as child nodes. These new names must be unique for each statement list among children of the same parent, but they do not need to be unique across the full program structure tree. Our strategy to generate such names is to use incremental counters for each parent node, assigning names recursively. For a parent node, its *N* children that are statement lists are respectively named *StmtList1*, *StmtList2*, . . . , *StmtListN*.

With that change in the merge algorithm to name statement lists, we implement JSFSTMERGE V1, which works in a consistent manner, producing merged revisions that are often semantically correct. A drawback of this version, however, is that it relies on the maintenance of one-to-one relationships between statement lists from two revisions. When one of the new revisions introduces changes that add or remove a *StatementList* node from the respective program structure tree, the tool might report spurious conflicts.

To illustrate that, Figure 2a shows an example of a three-way merge scenario in which there is no one-to-one mapping between statement lists from a base revision to a new revision. In the *base* revision, there are two statement lists at the top level. Then, *left* removes the unused variable *financePath* and function *getCurrentUrl*. The deletion of this function makes statements, which were in different statement lists in *base*, to be in a sequence, composing a single *StatementList* node. On the other hand, *right* keeps the number of statement lists at the top level, just changing the content of STMTLIST2. As a result, when merging *left* and *right*, superimposition does not understand that the statement from STMTLIST2 in base revision (Line 8) is now in *left*, within STMTLIST1. The algorithm simply interprets that *left* removed STMTLIST2, while *right* edited it, which causes JSFSTMERGE V1 to report a conflict (Figure 2c). An unstructured merge tool is able to merge *left* and *right* without reporting conflicts (Figure 2b).

### B. *jsFSTMerge v2*

To avoid that type of false positive in semistructured merge, we implemented another tool, JSFSTMERGE V2, which handles matching and statement lists ordering by using a different approach. Instead of position based naming, the strategy used by JSFSTMERGE V2 to deal with statement lists is based on a recursive operation performed on program structure trees prior to superimposition. This operation takes, for a given parent node, sibling statement list nodes and concatenate them into a single *StatementList* node, keeping the order of their statements. The idea behind that strategy comes from the observation that, as long as we keep the statements in order, it does not matter their order with respect to function declarations, because of the hoisting semantics. This allows us to freely rearrange group of statements in relation to function declarations without changing program semantics.


base
StmtList1 : StatementList
StmtList2 : StatementList
left
StmtList1 : StatementList
right
StmtList1 : StatementList
StmtList2 : StatementList

```
var mapsPath = 'maps';
var financePath = 'finance';

function getCurrentUrl() {
 ...
}

redirect(mapsPath);

function redirect(path) {
 ...
}
```

```
var mapsPath = 'maps';

redirect(mapsPath);

function redirect(path) {
 ...
}
```

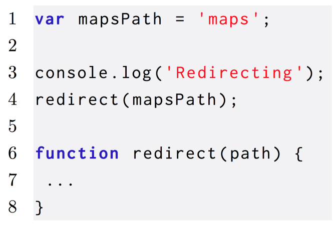
```
var mapsPath = 'maps';
var financePath = 'finance';

function getCurrentUrl() {
 ...
}

console.log('Redirecting');
redirect(mapsPath);

function redirect(path) {
 ...
}
```

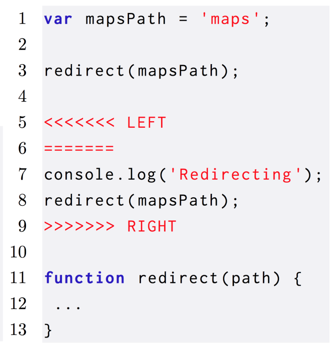
(a) Merge scenario



```
var mapsPath = 'maps';

console.log('Redirecting');
redirect(mapsPath);

function redirect(path) {
 ...
}
```



```
var mapsPath = 'maps';

redirect(mapsPath);

<<<<<<< LEFT
=======
console.log('Redirecting');
redirect(mapsPath);
>>>>>>> RIGHT

function redirect(path) {
 ...
}
```

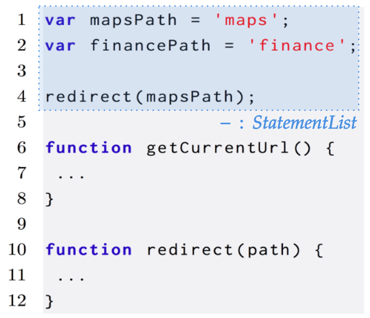
(b) Unstructured merge

(c) Semistructured merge using jsFSTMerge v1

Fig. 2. Additional false positive of JSFSTMERGE V1

Figure 3 illustrates such program transformation that is applied to revisions, by JSFSTMERGE V2, during merging. The transformed program is obtained from the *base* revision presented in Figure 2a by joining its statement lists into a single one; both programs are semantically equivalent.

A drawback of JSFSTMERGE V2 is that it potentially changes code format by rearranging groups of statements. Differently, JSFSTMERGE V1 keeps the original order of elements. As we can see in Figure 3, only the first statement list, given a parent node, maintains its position. All the other statement lists are moved to after the end of the first one. The trade-off between additional false positives introduced by JSFSTMERGE V1 and code reformatting added by JSFSTMERGE V2 is discussed in Section V.


– : StatementList

```
var mapsPath = 'maps';
var financePath = 'finance';

redirect(mapsPath);

function getCurrentUrl() {
 ...
}

function redirect(path) {
 ...
}
```

Fig. 3. Transformed program used by jsFSTMerge v2

## IV. COMPARING MERGE APPROACHES

To compare unstructured merge with semistructured merge, we would ideally measure how often each merge tool is able to detect interference between development contributions, so that it reports interfering changes as conflicts and automatically integrates non-interfering ones [3], [5]. Changes introduced by two developers are considered to be non-interfering when they do not affect each other [25]. As interference is not computable in our context, we adopt the study design proposed by Cavalcanti et al. [13]. They relatively compare unstructured and semistructured merge approaches in terms of the occurrence of *additional* false positives (conflicts that do not represent interference and are reported by just one of the tools) and false negatives (actual interference missed by one of the tools but reported by the other).

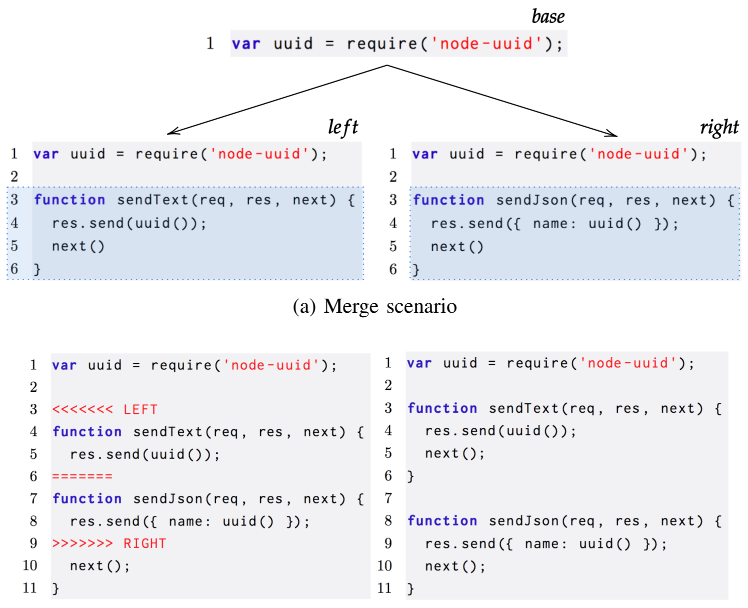
To guide this kind of relative comparison, we first manually analyzed a small sample of merge scenarios to identify, considering JavaScript programs and our implementations, categories of false positives and false negatives. These categories were enriched along the execution of our empirical study, whenever we observed new kinds of false positives and false negatives, and are detailed in the following subsections. As semistructured tools, we use both JSFSTMERGE V1 and JSFSTMERGE V2.[3] As unstructured tool, we use *KDiff3*,[4] one of several unstructured tools available, and a representative implementation of the *diff3* algorithm [10].

### A. Additional False Positives of Unstructured Merge

A shortcoming of unstructured merge is its inability to identify commutative and associative elements. For the version of JavaScript we consider here, only function declarations, due to the hosting semantics, are commutative and associative. However, in the context of our JSFSTMERGE V2 semistructured merge tool, statement lists can also be rearranged with respect to function declarations; in particular, a joined statement list node can be freely permuted with its siblings. All this is ignored by unstructured tools, which simply rely on textual analysis, leading to the so-called ordering conflicts.

[3]Throughout this section, for the purpose of analyzing false positives and false negatives, we refer to both versions of JSFSTMERGE as semistructured merge tool, unless a specific version is mentioned.

[4]http://kdiff3.sourceforge.net/



base:
```
var uuid = require('node-uuid');
```

left:
```
var uuid = require('node-uuid');

function sendText(req, res, next) {
  res.send(uuid());
  next()
}
```

right:
```
var uuid = require('node-uuid');

function sendJson(req, res, next) {
  res.send({ name: uuid() });
  next()
}
```

(a) Merge scenario

```
var uuid = require('node-uuid');

<<<<<<< LEFT
function sendText(req, res, next) {
  res.send(uuid());
=======
function sendJson(req, res, next) {
  res.send({ name: uuid() });
>>>>>>> RIGHT
  next();
}
```

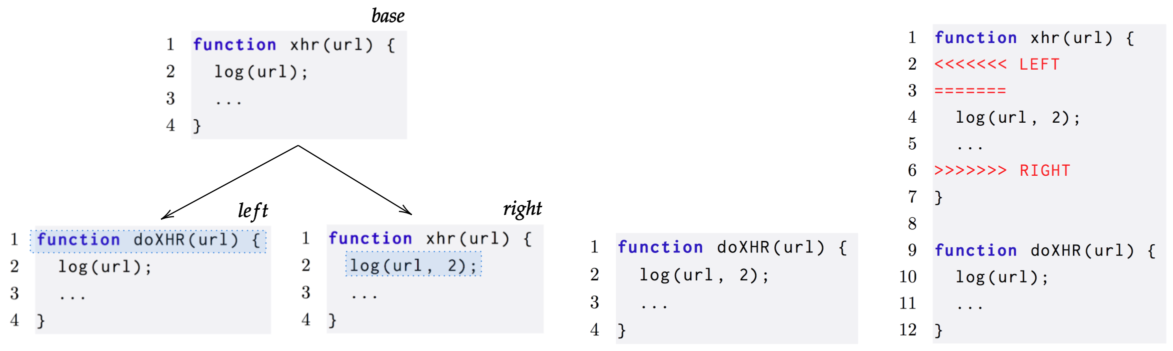
```
var uuid = require('node-uuid');

function sendText(req, res, next) {
  res.send(uuid());
  next();
}

function sendJson(req, res, next) {
  res.send({ name: uuid() });
  next();
}
```

(b) Unstructured merge (c) Semistructured merge

Fig. 4. Additional false positive of unstructured merge (ordering conflict)



base:
```
function xhr(url) {
  log(url);
  ...
}
```

left:
```
function doXHR(url) {
  log(url);
  ...
}
```

right:
```
function xhr(url) {
  log(url, 2);
  ...
}
```

(a) Merge scenario

```
function doXHR(url) {
  log(url, 2);
  ...
}
```

(b) Unstructured merge

```
function xhr(url) {
<<<<<<< LEFT
=======
  log(url, 2);
  ...
>>>>>>> RIGHT
}

function doXHR(url) {
  log(url);
  ...
}
```

(c) Semistructured merge

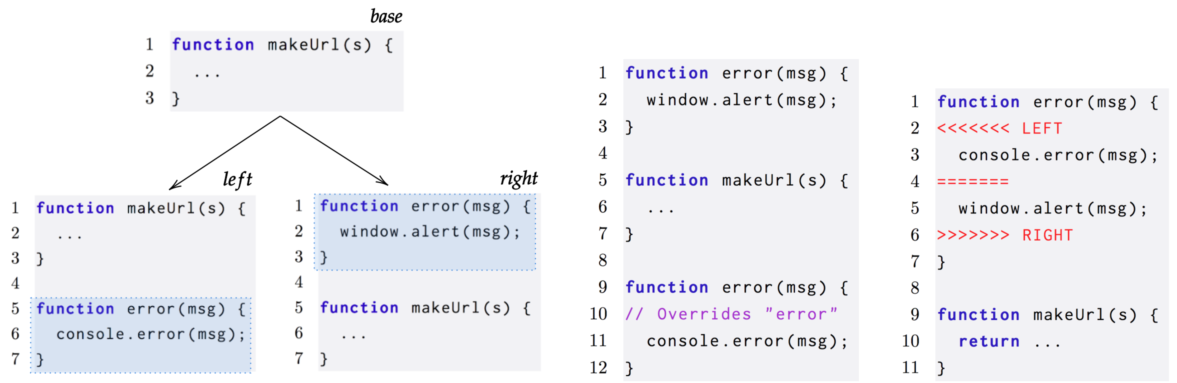
Fig. 5. Additional false positive of semistructured merge (function renaming conflict)

To illustrate an ordering conflict, Figure 4a shows a merge scenario in which two developers try to add different functions (`sendText` and `sendJson`) to the same area of the program text. An unstructured merge tool reports a conflict (see Figure 4b). In contrast, semistructured merge identifies that each function declaration corresponds to a distinct element, and yields the desired output without conflicts, by superimposing the program structure trees (see Figure 4c).

## *B. Additional False Positives of Semistructured Merge*

As previously noted [11], [13], renaming is a challenge for semistructured merge. When an element is renamed in a revision, the semistructured merge algorithm is not able to detect that. In fact, it can detect that the revision does not have an element with the original name, but the element could have been deleted, or even renamed with body changes, making it hard to accurately map the renamed element to its original version in the base revision. To illustrate that, we use the merge scenario in Figure 5a. Unstructured merge does not report conflicts, as the changes introduced by revisions *left* and *right* occur in distinct text areas. It generates a program that combines contributions from both developers, using the new function name and the new function body (Figure 5b) in the resulting program. Semistructured merge, on the other hand, is not able to understand that *left* renamed `xhr`. Instead, the algorithm interprets the function renaming as a deletion, and assumes that *left* deleted a function modified by *right*, thus reporting a conflict (Figure 5c).

Two additional kinds of semistructured merge false positives are special cases of renaming conflicts, because they are all caused by the same root problem: transformation of function declarations in one revision, triggering deletion of nodes that are edited in the other revision. The first one comes from the conversion, in one revision, of a function declaration into the assignment of a function expression to a variable. The false positive is reported if the other revision changes the function involved in the conversion (*function conversion conflict*). The



```
function makeUrl(s) {
  ...
}
```

```
function makeUrl(s) {
  ...
}

function error(msg) {
  console.error(msg);
}
```

```
function error(msg) {
  window.alert(msg);
}

function makeUrl(s) {
  ...
}
```

(a) Merge scenario

```
function error(msg) {
  window.alert(msg);
}

function makeUrl(s) {
  ...
}

function error(msg) {
// Overrides "error"
  console.error(msg);
}
```

(b) Unstructured merge

```
function error(msg) {
<<<<<<< LEFT
  console.error(msg);
=======
  window.alert(msg);
>>>>>>> RIGHT
}

function makeUrl(s) {
  return ...
}
```

(c) Semistructured merge

Fig. 6. Additional false negative of unstructured merge (duplicate function declaration)

same kind of false positive is reported when the assignment is converted into a declaration. The second kind of false positive might arise when a developer moves a function declaration into a sibling node (e.g., a *StatementList*). This causes the semistructured merge algorithm to interpret such change as a deletion of the moved function. A *function declaration displacement conflict* is then reported when one revision moves such a function declaration and the other revision changes the same declaration. Usually, such conflicts occur when a developer moves a function declaration into an *Immediately Invoked Function Expression* (IIFE).[5]

All the aforementioned cases of additional semistructured merge false positives are reported by both JSFSTMERGE V1 and JSFSTMERGE V2. However, as explained in the previous section, there is an extra type of false positive that is exclusive to JSFSTMERGE V1, which occurs when one-to-one mappings between *StatementList* nodes are not kept.

### C. Additional False Negatives of Unstructured Merge

Similarly to the findings obtained by Cavalcanti et al. [13] for Java systems, duplicate function declarations are additional false negatives of unstructured merge for JavaScript. When function declarations with the same name are added to different areas of the program, an unstructured merge tool reports no conflict. In JavaScript, there is no function overloading or checking of duplicate function declarations; when two functions— no matter their formal parameters— are declared with the same name within the same scope, the last function declaration simply overrides the previous one [22]. Not reporting a conflict is often a problem in such cases. For instance, consider the merge scenario in Figure 6a. Unstructured merge yields a program with two *error* functions (see Figure 6b), as they are declared in disjoint areas. This program is perfectly valid for JavaScript interpreters, but its behaviour might be disappointing for at least one of the developers responsible for the revisions. The integrated changes interfere, once the developer who introduced the first function relies on the behaviour of its implementation, and not on the behaviour of the overriding function. In contrast, semistructured merge is able to match the functions by name and type (signature) via superimposition, correctly identifying a conflict (see Figure 6c).

Function renaming can also be an additional false negative of unstructured merge. This occurs when a developer renames a function, and another developer, in addition to changing its body, adds a call to it by referring to the original name. Unstructured merge only detects such conflict by accident: when changes occur in the same text area. Otherwise, it unsoundly performs the merge, and generates a program that throws a runtime error for not being able to find a function declaration. Likewise, converting a function declaration into an assignment of a function expression to a variable can be an additional false negative of unstructured merge as well. When the developer who is still relying on the hoisting property of a function declaration adds new references before the declaration, semistructured merge soundly reports a conflict, and this is not detected by unstructured merge.

### D. Additional False Negatives of Semistructured Merge

We did not find additional false negatives of semistructured merge that conform to a set of recurring syntactic change patterns. In all such cases, unstructured merge *accidentally* detects semantic conflicts that would otherwise not be reported if changes were made in slightly different text areas. This often happens when a developer adds a new element that references an existing one, and, in another revision, a developer edits the referenced element in a manner that both changes occur in the same area. The first developer might not be expecting the changes introduced by the second one, potentially leading to unexpected behaviour.

### E. Comparing JavaScript and Java

The different types of false positives and false negatives identified by Cavalcanti et al. [13], for Java, are mostly equivalent to the ones we identified for JavaScript. Ordering conflicts are false positives for unstructured merge in both languages. Renaming conflicts can be either false positive for

[5]IIFE is simply a function expression, either named or anonymous, which is executed immediately right after its creation. This pattern is useful to avoid polluting the global scope [22].

semistructured merge or false negative for unstructured merge in the same manner in JavaScript and Java, with the difference that, in Java, this renaming conflict can happen to different elements (e.g., methods and classes), while for JavaScript (ES5), it is restricted to functions. Furthermore, additional JavaScript cases of false positive for semistructured merge can be seen as special cases of renaming conflicts. Regarding false negatives for semistructured merge, accidental conflicts can happen in both Java and JavaScript, but in Java, there are additional cases specific to the language (e.g., static blocks).

## V. EVALUATION

In this section, we analytically evaluate the generality of the FSTMERGE approach [11] to implement semistructured merge tools by simply instantiating a framework with an annotated language grammar, and implementing language specific plugins (handlers) to resolve certain types of conflicts. Furthermore, we empirically evaluate the merge tools we implemented by adapting instantiations of FSTMERGE for JavaScript. We compare them to unstructured tools, which are still widely used by industry. In particular, we investigate the research questions that name the following two sections.

### A. RQ1: Is the FSTMERGE semistructured approach generalizable for JavaScript?

After successfully instantiating FSTMERGE with a few programming languages, including Java and C#, Apel et al. [11] conjecture that the semistructured framework they implemented is generic enough to be applied for other languages. To fully support such conjecture, FSTMERGE's engine should be sufficiently generic, not requiring adaptations to provide semistructured merge support for languages with properly annotated grammars establishing how each language element should be represented, merged, and handled when conflicts are detected. However, as illustrated in Section II, FSTMERGE generality is limited. It is certainly possible to instantiate the framework for a JavaScript-like language, but the resulting merge tool would have serious weaknesses that preclude its use in practice.

These weaknesses derive from two main, and dependent, language characteristics:

1) allowing, at the same syntactic level, the mix of elements whose order is relevant (statements, in JavaScript's case) with elements whose order is arbitrary (function declarations, in JavaScript's case); and
2) not providing, in such situations, natural unique names for elements whose order is relevant.

Much of the power of semistructured merge comes from its capacity to properly match and change the order of commutative and associative language elements. But these capacities are seriously compromised by the mentioned characteristics, which are often observed in scripting languages (including PHP and Python, besides JavaScript). That is what led us to adapt not only JavaScript's off-the-shelf grammar, but also FSTMERGE's merge engine. Because such adaptations are necessary for deriving a proper FSTMERGE based merge tool for JavaScript, we claim that the FSTMERGE semistructured framework is not effectively generalizable for JavaScript.[6]

It is noteworthy to mention that FSTMERGE allows conflict handlers to deal with particular cases of terminal node merging (e.g., merge of *implements* list in Java) [11], which were not used in any version of JSFSTMERGE. Nevertheless, such plugins are not able to handle issues concerning matching and ordering of statements, which were only solved by modifying JavaScript grammar and merge engine.

FSTMERGE semistructured framework, in its pure form, is not effectively generalizable for JavaScript because of its limitation in dealing, at the same syntactic level, with both commutative elements and non-commutative elements that are not uniquely named. This lack of effective generalizability is also expected for other languages with similar characteristics.

### B. RQ2: How effective in practice is semistructured merge for JavaScript?

Many factors play a role in making a JavaScript semistructured merge tool a competitive alternative to traditional unstructured tools. Here we focus on two factors: 1) integration effort reduction, which leads to developer productivity improvements; and 2) correctness of the merging process, which is associated with software quality. To evaluate these factors, we conduct an empirical study based on a previous study [13] that focuses on Java systems. We investigate the same research questions, but targeting JavaScript systems:

- **RQ2.1:** *When compared to unstructured merge, does semistructured merge reduce unnecessary integration effort by reporting fewer false positives?*
- **RQ2.2:** *When compared to unstructured merge, does semistructured merge compromise integration correctness by missing more conflicts (having more false negatives)?*

As in the previous study, we answer these two research questions by first reproducing merges from the development history of a number of GitHub projects. For each merge scenario, we separately invoke three tools: unstructured merge, and the two semistructured tools JSFSTMERGE V1 and JSFSTMERGE V2. By comparing the tools' results, we compute, for each tool, the *additional false positives* and *additional false negatives* metrics discussed in the previous section. In particular, we use additional false positives of the analyzed merge tools as a metric to answer **RQ2.1**, and we use additional false negatives to answer **RQ2.2**.

*1) Evaluation Design:* We essentially use the same setup adopted by Cavalcanti et al. [13], with three main steps.

Initially, in the **mining step**, we searched for the top 100 projects that primarily use JavaScript and have the highest

[6]This claim applies to ES5 and to newer, backward compatible, versions of JavaScript. However, newer versions of JavaScript that support class declarations could benefit from a vanilla instantiation of FSTMERGE for files that simply declare classes, or do not mix statements and declarations at the same syntactic level.

number of stars on GitHub, which is a metric of project activity and relevance [26]. In addition to this list of 100 projects, we considered other JavaScript projects that got traction in the last few years in terms of developers popularity [27]. We then discarded projects that primarily use ES6 and newer versions of JavaScript. Subsequently, we selected 50 projects— the same number of projects considered in the study led by Cavalcanti et al. [13]— aiming to obtain diversity in the number of lines of code, number of developers, and number of commits. The list of selected projects is in our online appendix [28]. In total, we extracted 10,345 three-way merge scenarios from the 50 selected JavaScript projects.

We used GITMERGESMINER[7] to extract three-way merge scenarios from GitHub repositories for the selected projects. GITMERGESMINER first clones each project locally and, then, converts their development history into a graph database. This graph database represents commits as nodes, and each merge commit has true value for its `isMerge` attribute [13]. For each merge commit, we copy revisions involved in the three-way merge scenario: the common ancestor revision (*base*) and the two parent revisions of the merge commit (*left* and *right*).

After collecting sample projects and merge scenarios as described in the previous step, in the **execution step** we run unstructured (KDIFF3) and semistructured (JSFSTMERGE V1 and JSFSTMERGE V2) merge tools to reproduce merges of the collected scenarios, capturing the merge results and conflict information.

Finally, in the **analysis step** we generate lists of false positives and false negatives candidates. We collect all conflicts detected in the previous step, and use scripts that categorize the conflicts into three groups: 1) conflicts reported by both unstructured and semistructured merge (JSFSTMERGE V1 or JSFSTMERGE V2), 2) conflicts reported only by unstructured merge, and 3) conflicts reported only by semistructured merge. To determine if two conflicts reported by different merge tools are the same, we check if both resulting merged files and textual content surrounded by conflict markers match. To complement this step, we manually verify whether pairs of conflicts that are, respectively, in the second and third groups (conflicts reported by only one of the tools) refer to the same conflict, but with slight textual differences. In this case, they are moved into the first group. The remaining conflicts in the second and third groups are marked as candidates of additional false positives and false negatives. We then proceed to manually classify them into specific types of false positive and false negative according to the characteristics described in Section IV.

*2) Evaluation Results:* Table I presents overall results for the three merge tools considered in this work, considering the number of reported conflicts, merge scenarios with conflicts, additional false positives, and additional false negatives. In particular, JSFSTMERGE V1 reported 884 conflicts, compared to 918 reported by KDIFF3 (unstructured tool), representing a reduction of 3.85% in the total number of conflicts.

[7]https://git.io/fhHha

TABLE I
COMPARING UNSTRUCTURED AND SEMISTRUCTURED MERGE TOOLS

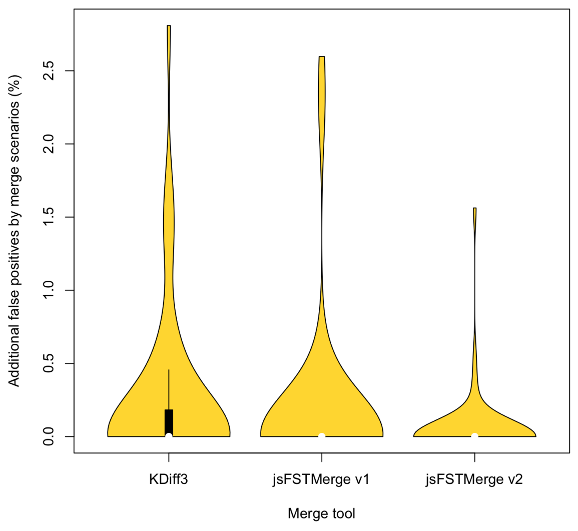
| | **KDIFF3** | **JSFSTMERGE V1** | **JSFSTMERGE V2** |
|---|---|---|---|
| **Reported Conflicts** | 918 | 884 (-3.85%) | 866 (-6%) |
| **Conflicting Merge Scenarios** | 582 | 566 (-2.75%) | 557 (-4.3%) |
| **Additional False Positives** | 58 | 25 | 7 |
| **Additional False Negatives** | 0 | 1 | 1 |

Looking at JSFSTMERGE V2 figures, we observe a greater reduction in the number of reported conflicts when compared to unstructured merge: it reports 866 conflicts, a reduction of 6%. Detailed results of additional false positives and false negatives, for each conflict type and sample project, are available in our online appendix [28]. Such details include number of conflicts, conflicting merge scenarios, conflicting files, and conflicting lines of code.

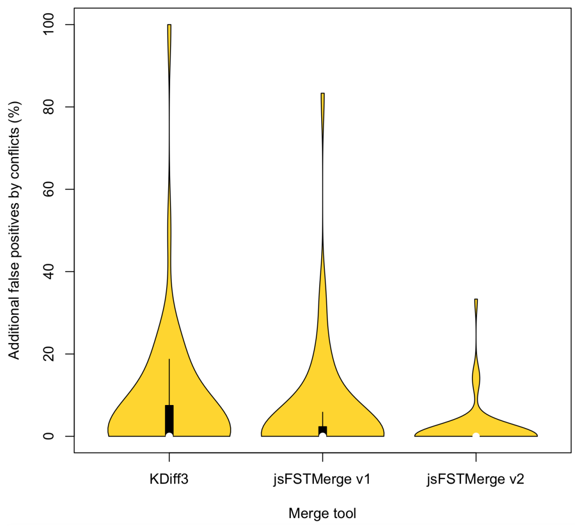
As discussed at the end of Section II, we decided to not include a vanilla instantiation of FSTMERGE in the evaluation because it has serious weaknesses. We found that such tool either reports too many spurious conflicts or produces invalid merges. To illustrate this, we consider one of the analyzed projects: ANGULARJS. As shown in our online appendix [28], out of 34 extracted merge scenarios from this project, unstructured merge presents only one additional false positive, whereas JSFSTMERGE V1 and JSFSTMERGE V2 present none. In contrast, a vanilla implementation of FSTMERGE, based on default naming, reports 54 additional false positives; clearly, an unacceptable performance.

For answering **RQ2.1**, we compare the number of additional false positives of the unstructured merge tool (KDIFF3) and the semistructured merge tools (JSFSTMERGE V1 and JSFSTMERGE V2). Figure 7a violin plots indicate that semistructured merge tends to have fewer merge scenarios with additional false positives than unstructured merge. Note that the 3rd quartile (top of the black box) of the unstructured tool is higher than zero, indicating that at least 25% of the projects in our sample had one or more merge scenarios with additional false positives when using KDIFF3. In contrast, the 3rd quartile for both semistructured merge tools is zero (no black box visible). When comparing JSFSTMERGE V1 and JSFSTMERGE V2, we find that the latter adds far fewer false positives. Figure 7b shows similar violin plots, this time illustrating the distribution of the percentages of conflicts that are additional false positives. A similar distribution can be observed, with the semistructured merge tools showing lower rates of reported conflicts accounted as additional false positive.

The plots in both figures suggest that there is no significant difference between the KDIFF3 and JSFSTMERGE V1 rates, whereas we can see a more promising difference between KDIFF3 and JSFSTMERGE V2 rates. To better investigate the statistical significance of differences in additional false positives percentages among the evaluated merge tools, we use Wilcoxon Signed-Rank tests given that our data are paired, deviate from normality, and come from the same sample [29].



(a) Merge scenarios



(b) Conflicts

Fig. 7. Per project distribution of additional false positives in terms of merge scenarios and conflicts

First, comparing JSFSTMERGE V1 and JSFSTMERGE V2, we find statistically significant differences, both in terms of merge scenarios and in terms of conflicts (*p-value* equals to, respectively, 0.0295 and $0.00296 < 0.05$, with medium effect size, according to Mann-Whitney's U test [30]). Conversely, when comparing KDIFF3 and JSFSTMERGE V1, a Wilcoxon Signed-Rank test finds no significant difference in percentages of merge scenarios with additional false positives (*p-value* $= 0.204 > 0.05$) and reported conflicts accounted as additional false positives (*p-value* $= 0.275 > 0.05$). Finally, comparing KDIFF3 and JSFSTMERGE V2, we find a statistically significant difference, in favour of the semistructured merge tool, both in terms of merge scenarios and conflicts (*p-value* equals to, respectively, 0.019 and $0.003 < 0.05$, with large effect size, according to Mann-Whitney's U test).

Both semistructured merge tools evaluated in this work reduced the overall number of reported conflicts, when compared to unstructured merge (see Table I), but only JSFSTMERGE V2 effectively presents fewer false positives than unstructured merge. However, to properly measure integration effort reduction caused by the use of a merge tool, we have to go beyond the numbers and analyze the nature of the additional false positives of each tool.

Considering an unstructured merge tool, the only kind of additional false positive in relation to semistructured merge is ordering conflict. When an ordering conflict involves two function declarations added to the same text area, it is usually easy to analyze and resolve it, since the developer just needs to choose one of the functions, or decide to keep both of them (by simply removing conflict markers). However, ordering conflicts are not always simple to resolve. We also observed what Cavalcanti et al. [13] call *crosscutting conflicts*, which do not respect boundaries of syntactic structures, mixing parts of different elements. In the context of JavaScript programs, such conflicts are typically reported by unstructured merge when statements are added to the same text area as a function declaration; in our sample, they occur more often than the simpler conflicts involving only function declarations.

Regarding semistructured merge additional false positives, most of them are caused by transformations of function declarations, triggering deletions of edited nodes. When this kind of conflict is reported, it may be initially hard to understand because it involves duplicate elements. But it is often quite easy to resolve, once the combination of contributions from both revisions is straightforward, apart from possible differences in indentation. In the case of function renaming conflicts, for instance, a resolution basically consists of selecting the new name from a revision and the new body from the other.

In our sample, both semistructured merge tools reduce the overall number of reported conflicts, and have fewer additional false positives than unstructured merge. Moreover, semistructured merge additional false positives tend to be easier to resolve.

Due to the small number of additional false negatives detected in our sample, we cannot properly answer **RQ2.2**. We find a minor difference between the number of false negatives exclusively missed by one of the merge tools. So, in our sample, unstructured and semistructured merge accuracies with respect to false negatives are essentially the same. It is, then, safe to say that, in our sample, semistructured merge does not compromise integration correctness. However, due to the lack of statistical significance, we cannot guarantee that this will hold for other samples.

We also observed that both unstructured and semistructured merge false negatives do not cause compilation or interpreter errors in JavaScript. The corresponding bugs might escape for end users unless they are detected by tests. So we consider that such errors are hard to detect during code integration. In Java, conversely, such false negatives are detected as compilation errors, which guide developers towards the location of the problem [13]. The dynamic nature of JavaScript (for example allowing multiple functions with the same name, but the last one overriding the previous ones) imposes additional challenges when handling such kinds of false negatives.

In our sample, the number of merges leading to unstructured or semistructured additional false negatives is insignificant. As a result, semistructured merge does not compromise integration correctness, but we cannot guarantee the same applies for other projects.

*3) Choosing a* JSFSTMERGE *Version:* The only difference between the results from JSFSTMERGE V1 and JSFSTMERGE V2 comes from the incidence of a kind of false positive that occurred only in the former tool. This false positive originates from the lack of a one-to-one mapping between *StatementList* nodes, as explained in Section III. All the other metrics and observations, in our sample, are exactly the same for both tools.

In summary, JSFSTMERGE V2 significantly reduces the number of reported conflicts and additional false positives, when compared to JSFSTMERGE V1. Moreover, JSFSTMERGE V2 has the same false negatives result as JSFSTMERGE V1, so they have the same impact on correctness. Consequently, we argue that JSFSTMERGE V2 is superior, in terms of merge accuracy, to JSFSTMERGE V1.

However, JSFSTMERGE V2 has a significant practical weakness because it rearranges statement lists, grouping them together, for a certain syntactic level, thus modifying the original code format. If keeping the order of statements with respect to function declarations is a requirement from a development point of view, JSFSTMERGE V2 becomes a less interesting and competitive alternative. To completely preserve the original format of the revisions, an improved version of JSFSTMERGE V2 could introduce special markers around statement lists that are joined before superimposition. These markers could then record the original position of each statement list, by using information such as line numbers in the revision file. An extra step could be added to the merge engine to leverage these positional metadata after finishing the merging process. The merged program structure tree could be processed to split joined statement lists back into individual statement lists, and, then, to move them to their original position. This, however, is left as future work.

*4) Choosing between Unstructured and Semistructured Merge:* Considering the quantitative metrics we used to relatively compare these merge tools, our findings indicate that semistructured merge, particularly the JSFSTMERGE V2 implementation, is a better merge tool than unstructured merge. When also analyzing the necessary effort for handling additional false positive and false negatives, semistructured merge also beats unstructured merge because it has false positives that are easier to resolve. The benefits, however, are not as significant as the ones observed for Java tools [13], and would hardly justify the use of JavaScript semistructured merge tools without further significant improvements.

Besides that, we have to consider other factors that influence the adoption of a merge tool in an industrial context. For example, the code reformatting introduced by JSFSTMERGE V2 might be, for most teams, a serious negative aspect of semistructured merge. Additionally, other factors such as team culture and project maturity might also influence merge tools selection; if many function renamings or conversions are expected, developers that use semistructured merge would have to face extra false positives, diminishing the already timid observed benefits.

*5) Threats to Validity:* With respect to **construct validity**, we measure integration effort mainly by the number of false positives reported by a merge approach. As such metric might not accurately approximate the actual effort necessary to analyze and resolve conflicts, we manually analyze every additional false positive (and false negative) in our sample. This way we could better assess conflict resolution effort.

Our merge scenario selection approach, based on mining public Git repositories, threatens the **internal validity** of our study. By using Git commands such as *rebase* and *cherry-pick*, developers might have locally and silently integrated code, leaving no trace on public repositories [31]. Such integrations are not considered in our analysis. Likewise, we may have lost merge scenarios if developers rewrote development history by, for example, squashing commits. The impact on the results presented here is hard to predict, but we are not aware of factors that could make the missed conflicts different from the ones we analyzed. Our manual analysis of false positives and false negatives is laborious and error-prone, but we double-checked each analyzed conflict.

Given that our proposed semistructured tools only support the ES5 version of JavaScript, and that we evaluated a restricted set of open-source projects, we cannot claim **external validity** to JavaScript systems in general. However, most, if not all, false positives and false negatives identified and analyzed here are likely to occur in projects written in newer versions of JavaScript, and even in similar languages such as Python and PHP.

## VI. RELATED WORK

A number of studies propose different merge tools and strategies to automatically detect and resolve code integration conflicts. Structured merge and diffing tools [16]–[19], which incorporate full structural information about the programs to be merged, sounded like a significant improvement over state of the practice unstructured merge tools. But they have not yet been adopted in industry, likely due to a number of adoption barriers, including the lack of solid evidence about the potential benefits, the associated performance penalties associated with building and merging trees, and the costs of developing language specific tools.

To tackle such barriers, Apel et al. [11] proposed semistructured merge and implemented a framework that reduces the costs of deriving language specific merge tools. They also show evidence in favour of semistructured merge, but considering only the number of reported conflicts. The results are partially confirmed in a replication considering different projects and another version control system [12]. Cavalcanti et al. [13] bring stronger evidence in favour of an improved version of a semistructured merge tool for Java. They consider not only conflict reduction, but also reduction in the numbers of additional false positives and false negatives as we do here. In fact, part of this work can be considered a replication of that study for JavaScript projects. Our results, however, are not as promising as the ones obtained for Java. Moreover, we bring evidence that, for languages with certain characteristics, the costs of deriving language specific merge tools are not as low as expected by Apel et al. [11]. Vanilla instantiations of their framework might not be enough for obtaining practical tools, as we illustrated here for the JavaScript case.

Accioly et al. [32] further investigated characteristics of semistructured merge conflicts in Java projects, and Apel et al. [33] propose a hybrid approach, developing a structured merge tool that automatically tunes the merging process by switching between unstructured and structured merge, according to the presence of certain conflicts. Other tools leverage additional semantic information of the underlying language, such as Binkley et al. [34], which propose a merge algorithm based on program dependence graphs. Additionally, Yang et al. [35] propose a merging algorithm that incorporates a notion of semantics-preserving transformations.

Contrasting with our mix of quantitative and qualitative evaluation of integration effort, Prudencio et al. [36] measure integration effort as the number of extra steps (creation, deletion or modification of software artifacts) that a developer needs to carry out to soundly integrate changes from independent code contributions. In this context, Santos and Murta [37] find a correlation between the number of steps and the amount of conflicts, suggesting that conflict reduction might lead to integration effort reduction. In a different direction, Kasi and Sarma [2] measure integration effort based on how many days a conflict was kept in the repository of a project, assuming that, during this period, the programmers worked solely to resolve this conflict, which may not always be the case.

Lastly, empirical studies provide evidence of the frequency and impact of conflicts. For instance, Brun et al. [1], and Kasi and Sarma [2], similarly to what was performed in our work, reproduce merge scenarios from different GitHub repositories to measure the frequency in which merge scenarios result in conflicts. Likewise, Zimmermann [7] reproduces merge scenarios from CVS projects. These works indicate that conflicts, in fact, occur frequently. Cavalcanti et al. [13] and the study presented here add to the body of knowledge by collecting evidence about how often conflicts that demand unnecessary integration occur, as well as how often actual interference are undetected by different merge tools; in our context, considering JavaScript programs. Regarding additional false negatives, Brun et al. [1], and Kasi and Sarma [2] investigate the frequency of merge scenarios that present build or test failures, which can be seen as a consequence of undetected interference introduced in the merging process. We identify and categorize unstructured merge additional false negatives that are specific for JavaScript, and that might not cause compilation errors (e.g., by allowing two functions with the same name), but that might trigger test errors, since behavioural errors introduced by false negatives might cause test assertions to fail.

## VII. Conclusions

Previous studies [11]–[13] suggest that semistructured merge might reduce integration effort when compared to unstructured merge. As such studies focus mostly on Java, it was uncertain whether semistructured merge would be similarly effective for languages such as JavaScript, due to significant differences in syntax and semantics. We found that implementations of semistructured merge tools for JavaScript, based on vanilla instantiations of the FSTMerge framework [11], often incorrectly merge programs. This happens because JavaScript syntax allows, at the same syntactic level (say at the program top level), interleaving statements and function declarations, and FSTMerge has limitations to handle, at the same level, commutative language elements (function declarations) and non-commutative elements (statements) that are not uniquely named. We then show that, unless adaptations are made to the input grammar and merge engine, the FSTMerge framework cannot be used to derive effective merge tools for languages with similar characteristics to JavaScript.

We propose and implement two semistructured merge tools for JavaScript, by adapting the FSTMerge framework and its merge engine. The first tool basically uses the order to uniquely name statement sequences, and the second tool basically concatenates statement sequences before merging. By reproducing 10,345 merge scenarios from 50 JavaScript projects, we found that the second tool significantly reduced, in our sample, the number of additional false positives when compared to unstructured merge and the first semistructured merge tool. Furthermore, both semistructured tools presented exactly the same false negatives. When comparing the second semistructured merge tool to an unstructured merge tool, we observed that the semistructured merge tool reduced the total number of reported conflicts by 6%, reducing the number of additional false positives, while not significantly compromising integration correctness. The benefits, however, are much inferior to the ones obtained for Java semistructured merge tools, and hardly justify the use of JavaScript semistructured merge tools without further significant improvements.

## Acknowledgments

We thank Delano Oliveira and Flavio Filho for testing our semistructured tools, and evaluating them in a few projects. Thanks also to members of the Software Productivity Group for suggestions that contributed to improve this work. We are grateful to Sven Apel and colleagues for making the FSTMerge framework available. This research was partially funded by INES 2.0, FACEPE grants APQ 0388-1.03/14 and APQ-0399-1.03/17, CAPES grant 88887.136410/2017-00, and CNPq grant 465614/2014-0.

## REFERENCES

[1] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Proactive Detection of Collaboration Conflicts," Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, 2011.

[2] B. K. Kasi and A. Sarma, "Cassandra: Proactive Conflict Minimization Through Optimized Task Scheduling," Proceedings of the 35th International Conference on Software Engineering, 2013.

[3] S. Horwitz, J. Prins, and T. Reps, "Integrating Noninterfering Versions of Programs," ACM Transactions on Programming Languages and Systems, vol. 11, n.3, 1989.

[4] R. E. Grinter, "Using a Configuration Management Tool to Coordinate Software Development," Proceedings of Conference on Organizational Computing Systems, 1995.

[5] D. E. Perry, H. P. Siy, and L. G. Votta, "Parallel Changes in Large-scale Software Development: an observational case study," ACM Transactions on Software Engineering and Methodology, vol. 10, n. 3, 2001.

[6] J. Estublier and S. Garcia, "Process Model and Awareness in SCM," Proceedings of the 12th International Workshop on Software Configuration Management, 2005.

[7] T. Zimmermann, "Mining Workspace Updates in CVS," Proceedings of the 4th International Workshop on Mining Software Repositories, 2007.

[8] T. Mens, "A State-of-the-Art Survey on Software Merging," IEEE Transactions on Software Engineering, vol. 28, n. 5, 2002.

[9] E. Lippe and N. Oosterom, "Operation-based merging," Proceedings of the 5th ACM SIGSOFT symposium on Software development environments, 1992.

[10] S. Khanna, K. Kunal, B. C. Pierce, "A Formal Investigation of Diff3," Proceedings of the 27th International Conference on Foundations of Software Technology and Theoretical Computer Science, 2007.

[11] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner, "Semistructured merge: Rethinking merge in revision control systems," Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, 2011.

[12] G. Cavalcanti, P. Accioly, and P. Borba, "Assessing Semistructured Merge in Version Control Systems: A Replicated Experiment," Proceedings of the 9th International Symposium on Empirical Software Engineering and Measurement, 2015.

[13] G. Cavalcanti, P. Accioly, and P. Borba, "Evaluating and improving semistructured merge," Proceedings of the ACM on programming languages, vol. 28, n. 5, 2017.

[14] A. Nederlof, A. Mesbah, and A. V. Deursen, "Software engineering for the web: the state of the practice," Proceedings of 36th International Conference on Software Engineering, 2014.

[15] L. H. Silva, M. T. Valente, A. Bergel, N. Anquetil, and A. Etien, "Identifying classes in legacy JavaScript code," Journal of Software: Evolution and Process, vol. 1, n. 1, 2017.

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

[17] J. Buffenbarger, "Syntactic Software Merging," Selected Papers from the ICSE SCM-4 and SCM-5 Workshops, on Software Configuration Management, 1995.

[25] J. A. Goguen and J. Meseguer, "FSecurity policies and security models," IEEE Symposium on Security and Privacy, 1982.

[18] J. E. Grass, "Cdiff: A Syntax Directed Differencer for C++ Programs," Proceedings of the USENIX C++ Conference, 1992.

[19] T. Apiwattanapong, A. Orso, and M. J. Harrold, "JDiff: A Differencing Technique and Tool for Object-oriented Programs," Automated Software Engineering, vol. 14, n. 1, 2007.

[20] S. Apel and C. Lengauer, "Superimposition: A language independent approach to software composition," Proceedings of the 7th International Conference on Software Composition, 2008.

[21] ECMA, "ECMA-262: ECMAScript Language Specification. Edition 5.1," [Online]. Available: https://www.ecma-international.org/ecma-262/5.1/.

[22] S. Stefanov, "JavaScript Patterns," O'Reilly, 2010.

[23] S. Apel and D. Hutchins, "A Calculus for Uniform Feature Composition," ACM Transactions on Programming Languages and Systems, vol. 3, n. 5, 2010.

[24] S. Apel, C. Lengauer, B. Möller, and C. Kästne, "An Algebraic Foundation for Automatic Feature-Based Program Synthesis," Science of Computer Programming, vol. 75, n. 11, 2010.

[26] H. Borges and M. T. Valente, "What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform," Journal of Systems and Software, vol. 146, n.1, 2018.

[27] S. Grief and M. Rambeau, "2018 JavaScript Rising Stars," [Online]. Available: https://risingstars.js.org/2018/en/, 2018.

[28] "Online Appendix," [Online]. Available: https://ase2019author.github.io, 2019.

[29] F. Wilcoxon and R. A. Wilcox, "Some rapid approximate statistical procedures," Lederle Laboratories, 1964.

[30] H. Mann and D. Whitney, "On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other," Annals of Mathematical Statistics, 1947.

[31] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, "The Promises and Perils of Mining Git," Proceedings of the 6th Working Conference on Mining Software Repositories, 2009.

[32] P. Accioly, P. Borba, and G. Cavalcanti, "Understanding semi-structured merge conflict characteristics in open-source Java projects," Empirical Software Engineering, vol. 23, n. 4, 2017.

[33] S. Apel, O. Lessenich, and C. Lengauer, "Structured Merge with Auto-tuning: Balancing Precision and Performance," Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, 2012.

[34] D. Binkley, S. Horwitz, and T. Reps, "Program Integration for Languages with Procedure Calls," ACM Transactions on Software Engineering and Methodology, 1995.

[35] W. Yang, S. Horwitz, and T. Reps, "A program integration algorithm that accommodates semantics-preserving transformations," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 1, n. 3, 1992.

[36] J. G. Prudencio, L. Murta, C. Werner, and R. Cepeda, "To lock, or not to lock: That is the question," Journal of Systems and Software, vol. 85, n. 2, 2012.

[37] R. Santos and L. Murta, "Evaluating the Branch Merging Effort in Version Control Systems," Proceedings of the 26th Brazilian Symposium on Software Engineering, 2012.