Pós-Graduação em Ciência da Computação

Alberto Trindade Tavares

Semistructured Merge in JavaScript Systems

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
http://cin.ufpe.br/~posgraduacao

Recife
2019

Alberto Trindade Tavares

Semistructured Merge in JavaScript Systems

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Engenharia de Software
Orientador: Sérgio Castelo Branco Soares
Coorientador: Paulo Henrique Monteiro Borba

Recife
2019

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

T231s Tavares, Alberto Trindade
Semistructured merge in JavaScript systems / Alberto Trindade Tavares. – 2019.
105 f.: il., fig., tab.

Orientador: Sérgio Castelo Branco Soares.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2018.
Inclui referências e apêndice.

1. Engenharia de software. 2. Integração de software. I. Soares, Sérgio Castelo Branco (orientador). II. Título.

005.1 CDD (23. ed.) UFPE- MEI 2019-023

Dissertação de Mestrado apresentada por **Alberto Trindade Tavares** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título "**Semistructured Merge in JavaScript Systems**" **Orientador: Sérgio Castelo Branco Soares** e aprovada pela Banca Examinadora formada pelos professores:

Prof. Leopoldo Motta Teixeira
Centro de Informática/ UFPE

Prof. Marco Tulio de Oliveira Valente
Departamento de Ciência da Computação / UFMG

Prof. Paulo Henrique Monteiro Borba
Centro de Informática / UFPE

Prof. **Orientador**: Sérgio Castelo Branco Soares
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 25 de Fevereiro de 2019.

Prof. Ricardo Bastos Cavalcante Prudêncio
Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.

I dedicate this work to everyone, especially my family and girlfriend, who gave me the necessary support and encouragement to get here

ACKNOWLEDGEMENTS

This thesis would not have been possible without the help of many people who directly or indirectly contributed to its completion. First of all, I would like to thank my family and my girlfriend, who always gave me encouragement, since the beginning of my academic life and, especially, in these more recent tough moments when there was an ocean between us. All the support I received from them was beyond invaluable and essential to keep me going towards concluding this thesis.

A very special thank you goes out to my old friends, living in Brazil, and my new friends, living in Denmark, for their companionship and for sharing many memorable experiences, which helped make my academic journey smoother. Here, I also include my former co-workers, from SENAI Innovation Institute (SII), and the current ones, from Trustpilot, with whom I spent most of my days over the last years.

I convey my sincere thanks to Guilherme Cavalcanti, who greatly helped me in setting up my first experiments with semistructured merge and in better understanding its technicalities, besides all the assistance to my research. Regarding the research, I would like to also acknowledge Delano Oliveira and Flavio Filho, who helped me in the evaluation of our semistructured merge tools.

I wish to thank the members of my defense committee, including Dr. Leopoldo Teixeira and Dr. Marco Tulio Valente, for investing time and providing their valuable feedback. I feel honoured that they have accepted to be on my committee.

I am deeply grateful to Dr. Paulo Borba, who co-advised this work with an exceptional competence and dedication, guiding me throughout my research and providing an endless supply of suggestions and constructive criticisms which were essential for making this study achieve more relevant results.

Last but not least, I would like to express my immense gratitude to my advisor, Dr. Sérgio Soares, who— years before becoming my boss at SII and, then, my advisor— was my professor during my undergraduate studies. While working with me at SII, he pushed me to pursue my Master's degree and was very understanding of my situation while working and studying at the same time, always being prompt to help.

ABSTRACT

In a collaborative development environment, programmers often work on simultaneous tasks which involve common software artifacts. As a consequence, when developers merge independent code contributions from different tasks, one might have to deal with conflicting changes, hampering the productivity of such collaborative development. The industry widely uses unstructured merge tools, that rely on textual analysis, to detect and resolve conflicts between developers' code contributions. On the other hand, semistructured merge tools go further by partially analyzing the syntactic structure and semantics of the code artifacts involved in a conflict. Previous studies compared these merge approaches, showing that semistructured merge is superior to unstructured one with respect to the number of reported conflicts, reducing the integration effort spent by developers, but, also, possibly negatively impacting the correctness of the merging process. However, these studies are based on semistructured merge tools, built on top of the FSTMerge architecture, that support different languages such as Java and C#, but not JavaScript, the most popular programming language for the Web. JavaScript has distinctive features when compared to those languages, which potentially lead to different results of effectiveness in solving conflicts by using the semistructured merge approach. In this work, we implement different versions of semistructured tools— based on FSTMerge— that work with JavaScript, and we conduct a study to compare them to an unstructured tool in order to better understand how semistructured merge works across different languages. During the implementation of tools for JavaScript, we found that the FSTMerge approach is not fully generalizable for programming languages that share similar characteristics with JavaScript; in particular, languages that allow statements at the same syntactic level as commutative and associative declarations. For those languages, further adaptations to the FSTMerge architecture are necessary. Nevertheless, we found evidences that semistructured merge approach for JavaScript reports fewer spurious conflicts than unstructured merge, without significantly impacting the integration correctness. Even though the reduction of reported conflicts is lower than that reported in previous studies for Java and C#, semistructured merge still seems to be a promising alternative to traditional unstructured merge when working with JavaScript.

Keywords: Collaborative development. Software merging. Semistructured merge. Version control systems. JavaScript.

RESUMO

Em um ambiente de desenvolvimento colaborativo, programadores frequentemente trabalham de forma paralela em tarefas de desenvolvimento que envolvem artefatos de software em comum. Como consequência, durante a integração de contribuições de código resultantes de diferentes tarefas, programadores podem ter que lidar com alterações conflitantes, o que afeta a sua produtividade. A indústria usa, em sua maior parte, ferramentas de integração não-estruturadas, que se baseiam somente em uma análise textual, para resolver conflitos entre as contribuições dos programadores. Por sua vez, ferramentas de integração semi-estruturadas tentam ir além, explorando a estrutura sintática do código envolvido em um conflito. Estudos anteriores compararam essas duas abordagens de integração de código e eles obtiveram resultados que mostraram que a integração semi-estruturada é superior à não-estruturada no que diz respeito à quantidade de conflitos reportados, o que reduz o esforço de desenvolvedores na integração de código, mas, ao mesmo tempo, também mostraram que a integração semi-estruturada pode ter um impacto negativo na corretude do código produzido. No entanto, esses estudos são baseados em ferramentas de merge semi-estruturada, construídas a partir de uma arquitetura conhecida como FSTMerge, que suportam diferentes linguagens de programação, tais como Java e C#, mas não JavaScript, que é a linguagem de programação mais popular para a Web. JavaScript possui características distintas quando comparada com essas outras linguagens, o que leva a integração semi-estruturada a apresentar resultados diferentes de efetividade na resolução de conflitos. Neste trabalho, nós implementamos diferentes versões de ferramentas semi-estruturadas, baseadas no FSTMerge, para JavaScript e conduzimos um estudo para compará-las com uma ferramenta não estruturada, com o objetivo de entender melhor como a abordagem semi-estruturada se comporta em diferentes linguagens. Durante a implementação dessas ferramentas para JavaScript, nós observamos que a abordagem proposta pelo FSTMerge não é totalmente generalizável para JavaScript e outras linguagens que compartilham características similares; em especial, linguagens que permitem comandos no mesmo nível sintático que declarações comutativas e associativas. Para tais linguagens, é necessário realizar adaptações na arquitetura do FSTMerge. Não obstante, nós obtivemos resultados que indicam que a abordagem semi-estruturada para JavaScript reporta menos conflitos espúrios que a abordagem não-estruturada, sem afetar negativamente a corretude da integração de código. Embora essa redução no número de conflitos reportados seja menor que a obtida em estudos baseados em Java e C#, a abordagem semi-estruturada, ao considerar programas escritos em JavaScript, ainda se mostra uma alternativa promissora às tradicionais ferramentas não-estruturadas.

Palavras-chave: Desenvolvimento colaborativo. Integração de software. Integração semi-estruturada. Sistemas de controle de versão. JavaScript.

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

As software projects become ever larger, both in terms of number of development team members and source code size, coordination of changes to a system without causing harm or unnecessarily hindering productivity turns out to be a challenge (BIRD; ZIMMERMANN, 2012). In a collaborative development setting, programmers often work on simultaneous tasks that involve common project artifacts (e.g., source code, build files, etc). Consequently, when integrating code changes from independent tasks, programmers might have to deal with conflicting changes, impairing development productivity. These conflicts may arise when different programmers introduce changes to the same project artifacts, or even when parallel changes are made to different artifacts, but leading to build or test errors (BRUN et al., 2011; KASI; SARMA, 2013). Such conflicts are costly, once they delay the software project while developers dedicate substantial effort to understand and resolve each conflict (HORWITZ; PRINS; REPS, 1989; GRINTER, 1995; PERRY; SIY; VOTTA, 2001; ESTUBLIER; GARCIA, 2005; ZIMMERMANN, 2007).

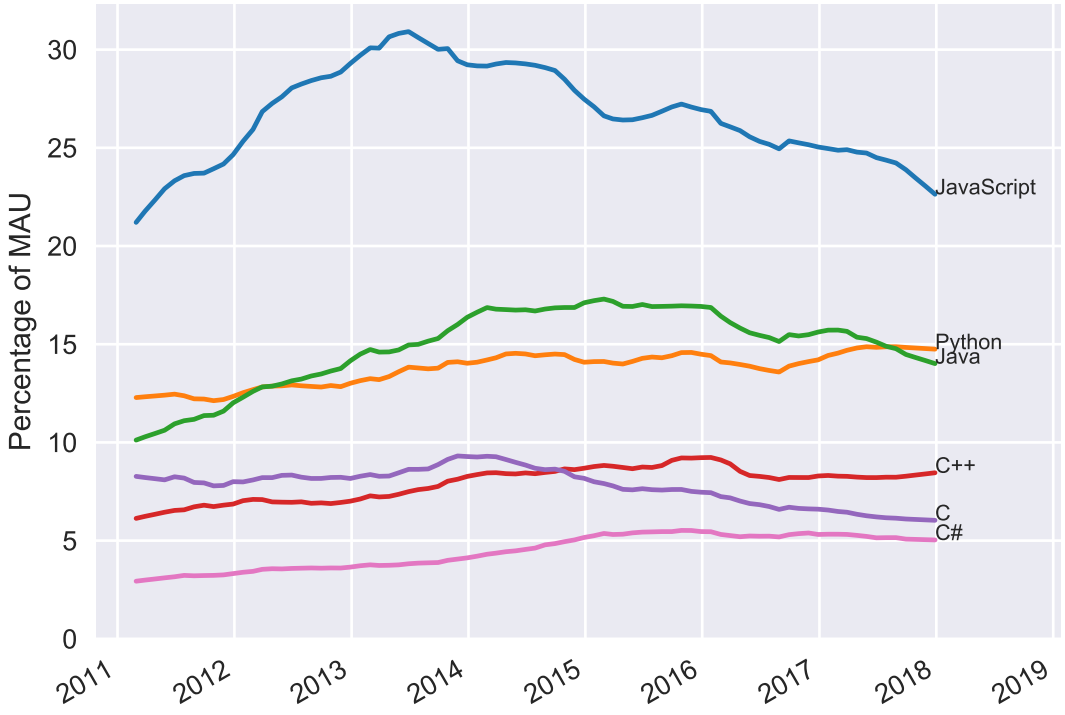
Version control systems (VCSs), such as Git (GIT, 2018), support the coordination of multiple programmers, allowing them to work on their own workplaces and provide revision of their code in parallel (O'SULLIVAN, 2009). One of the capabilities of VCSs is merging software artifact revisions, which might involve conflicting changes. VCSs can employ distinct merge tools to mitigate and resolve conflicts. These tools implement merge approaches which mainly differ on how software artifacts are represented. The industry widely uses unstructured merge approach to detect and resolve merge conflicts. However, unstructured tools rely purely on textual analysis, identifying conflicts via textual similarity (KHANNA; KUNAL; PIERCE, 2007), which leads to the report of spurious conflicts that require effort from developers to manually resolve them. On the other hand, semistructured merge approach goes further by leveraging structural information about the underlying code artifacts involved in a conflict (APEL et al., 2011).

Previous studies compare these merge approaches, showing that semistructured merge is superior to unstructured one with respect to the number of detected conflicts. Apel et al. (2011) report an average reduction of 34% compared to unstructured merge. Cavalcanti, Accioly and Borba (2015) conducted a replication of this study, finding an average reduction of 21% of reported conflicts by semistructured merge. In contrast, Cavalcanti, Accioly and Borba (2017) found that, while still reducing the number of reported conflicts (a reduction of 24%), semistructured merge introduces false negatives, i.e., actual interferences between developers not reported as conflicts, which compromises integration correctness. To evaluate the semistructured approach, these studies used tools built on top of a semistructured merge engine implemented by Apel et al. (2011), called `FSTMerge`, having support for different programming languages such as Java and C#, but not JavaScript.

1.1 MOTIVATION AND PROBLEM

The industrial adoption of semistructured merge depends on many factors, including, for example, usability of a tool (FAVRE; ESTUBLIER; SANLAVILLE, 2003). Other relevant factors that could justify the usage of semistructured merge in practice are evidences that it reduces the integration effort without risking the correctness of the merging process, when compared to the unstructured approach. Cavalcanti, Accioly and Borba (2017), in their research, investigate the impact of semistructured merge usage on integration effort and correctness, but they focus on Java systems. While their results may be generalized to other programming languages that share similar characteristics (e.g., C# and C++), little could be said with respect to scripting languages that are more dynamic and loosely typed, such as JavaScript, PHP, and Python. The latter languages have a more flexible syntax that allows top-level statements in a program, which might affect the effectiveness of semistructured merge approach.

Figure 1 – Percentage of monthly active users per programming language on GitHub



Source: Frederickson (2018)

The support for programming languages that are often used in industry is another important factor regarding adoption of semistructured merge. And there was still no implementation of a semistructured tool for the most popular language for the Web: JavaScript. JavaScript was initially created to extend Web pages with small portions of code, but since then, its popularity and relevance have only grown (NEDERLOF; MESBAH; DEURSEN, 2014; SILVA et al., 2017). JavaScript is now the most popular programming

language on GitHub, being used by 34.2% of the top-2,500 most popular systems on the platform (BORGES; HORA; VALENTE, 2016). Figure 1 presents, for different languages, the percentage of monthly active users (MAU) on GitHub, showing JavaScript as the most used language. The number of MAU was calculated by Frederickson (2018), using `GHTorrent` (GOUSIOS, 2013) as data source, based on how often users have pushed code, forked or starred a repository, or opened an issue. Even though the most common runtime environment for JavaScript is the Web browser, the language has started to be employed on the server-side of applications by means of a framework called Node.js. Node.js became one of the main components of the "JavaScript everywhere" paradigm, enabling Web development around a single programming language (PEREIRA, 2012).

Considering the importance of JavaScript for software development, especially Web applications, and the evidence, reported by previous studies, that semistructured merge can provide better results for other programming languages, it becomes quite relevant to investigate how semistructured merge would behave for JavaScript. It is important not only to evaluate whether the semistructured approach can be a better alternative to the unstructured one when targeting JavaScript systems, but also to better understand how semistructured merge can be effectively implemented for other programming languages; in particular, scripting languages that have similar features to JavaScript (e.g., PHP and Python).

1.2 OBJECTIVES

In order to address the lack of implementation and evaluation of a semistructured merge tool for JavaScript, the major objectives of this thesis are to propose and implement such tool, by leveraging the `FSTMerge` architecture, and to evaluate its effectiveness when used in practice. In particular, we want to investigate the following main research questions:

- **RQ1:** *Is the FSTMerge semistructured approach generalizable for JavaScript?*
- **RQ2:** *Could semistructured merge for JavaScript be effective in practice?*

For answering **RQ1**, we investigate whether `FSTMerge` semistructured approach, as proposed by Apel et al. (2011), is generic enough to support the implementation of an effective merge tool for JavaScript. In an attempt to answer **RQ2**, we conduct an empirical study based on a research led by Cavalcanti, Accioly and Borba (2017), which focuses on Java, to compare unstructured and semistructured merge approaches for JavaScript in terms of 1) frequency in which spurious conflicts are reported (*false positives*), which affects integration effort, and 2) frequency in which actual interferences between development tasks are not reported (*false negatives*), which affects integration correctness. We reproduce 10,526 merge scenarios (sets consisting of a common ancestor revision and its

derived revisions) from 50 JavaScript projects for collecting data to enable this comparison. The research questions investigated by Cavalcanti, Accioly and Borba (2017) become sub-questions of **RQ2** in our study:

- **RQ2.1:** *When compared to unstructured merge, does semistructured merge reduce unnecessary integration effort by reporting fewer spurious conflicts?*
- **RQ2.2:** *When compared to unstructured merge, does semistructured merge compromise integration correctness by missing more non spurious conflicts?*

1.3 THESIS OUTLINE

The remainder of this thesis is structured as follows:

- Chapter 2 presents background information on version control systems, merge approaches and JavaScript; the main concepts used to understand this thesis.
- Chapter 3 describes our proposed approach to implement semistructured merge tools for JavaScript. We also document different types of conflicts unstructured and semistructured tools are able to detect or not when merging JavaScript artifacts.
- Chapter 4 presents an analytical evaluation of the generalizability of `FSTMerge` as a framework to create semistructured tools for JavaScript and other programming languages. We also present an empirical evaluation of semistructured merge tools developed in this work, comparing them to an unstructured merge tool, with respect to integration effort and correctness.
- Chapter 5 sums up our conclusions and contributions, and discusses related and future work.

2 BACKGROUND

In this chapter, we present background information related to our work. First, we discuss central concepts of version control systems (VCSs), and how they support collaborative software development (Section 2.1). In this context, we explain how VCSs perform software merging, and how they can be empowered to detect and resolve merge conflicts automatically, highlighting unstructured and semistructured merge tools (Section 2.2). Finally, in Section 2.3, we present the fundamentals of the JavaScript language, discussing characteristics, including lexical grammar and semantics, that are relevant for the implementation of a semistructured merge tool.

2.1 VERSION CONTROL SYSTEMS

Version control systems arise in the context of software configuration management (SCM). SCM is a software engineering discipline used by organizations to manage the evolution of large and complex systems (TICHY, 1988), including techniques to assist developers in carrying out parallel changes to software artifacts. These techniques include version control mechanisms to track the composition of programs evolved into parallel versions from which new versions can be derived by means of software merging (CONRADI; WESTFECHTEL, 1998; MENS, 2002). The fundamental idea of VCSs, from the first systems developed in the early 70s to modern ones, is tracking files in such a manner that for each time a file is significantly changed, a revision is created. As a result, a file evolves as a sequence of revisions. From any revision, a branch can be created, which is a new line of development that leads to a revision tree. One of the key ideas of VCSs is storing only the differences between successive revisions, vastly reducing the amount of required memory (ESTUBLIER, 2000).

Figure 2 – Revision branching and merging

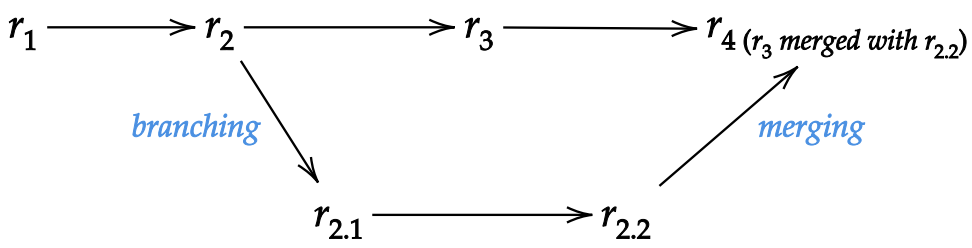


Figure 2 illustrates a succession of revisions created from an initial one referred by r_1: r_2 and r_3. From r_2, a branch is issued, generating the revision $r_{2.1}$, which starts an independent line of development. Such mechanism allows developers to work completely separate from time to time. On the other hand, there is a need to integrate parallel code

contributions, and this can be done via merging. In that example, the revisions r_3 and $r_{2.2}$ are merged into a new shared revision: r_4. The section that follows explains in more detail different approaches to deal with merge of revisions, and with conflicts between concurrent changes that need to be resolved (Section 2.2).

In a practical context, VCSs provide, in order to enable developers to work on revisions, repositories to store software artifacts, as well access and modification support to these artifacts (PRUDENCIO et al., 2012). There are two different approaches to determine how such repositories are disposed, and how their access is made available: 1) the *centralized* model, implemented by Centralized Version Control Systems (CVCSs), and 2) the *distributed* model, implemented by Distributed Version Control Systems (DVCSs).

The centralized model has long dominated the design of version control systems. In CVCSs, there is a single canonical source repository, from which all the developers work against, and synchronize their local working area. These centralized systems rely on a client-server architecture, having developers communicate their changes through a central server (ALWIS; SILLITO, 2009; RIGBY et al., 2009). Examples of CVCSs include Subversion (SUBVERSION, 2018), CVS (CVS, 2018), and RCS (RCS, 2018). However, that centralized model started to fail in enabling a development process based on a large number of possibly geographically-distributed programmers. One of the main limitations of CVCSs is the dependence on a main server, which is a single point of failure and may not be always accessible, e.g., requiring internet connection to save code revisions.

DVCSs have emerged to address some of the limitations of the centralized model, introducing a peer-to-peer architecture that allows the propagation of changes between repositories, and no longer requiring, even though it may still be employed, a central server. Each developer, now, has their own local repository, a copy that includes the full commit history, and that can be synchronized with other distributed repositories (RIGBY et al., 2009). In general, DVCSs provide more flexible merge capabilities due to the information they maintain in local repositories across different branches, better supporting decentralized workflows. Furthermore, a local repository enables developers to work on revisions completely offline, and, also, most of the operations (e.g., *diff* between revisions) are much faster because the files are available locally. Among open-source projects, Git (GIT, 2018) and Mercurial (MERCURIAL, 2018) are examples of distributed version control systems.

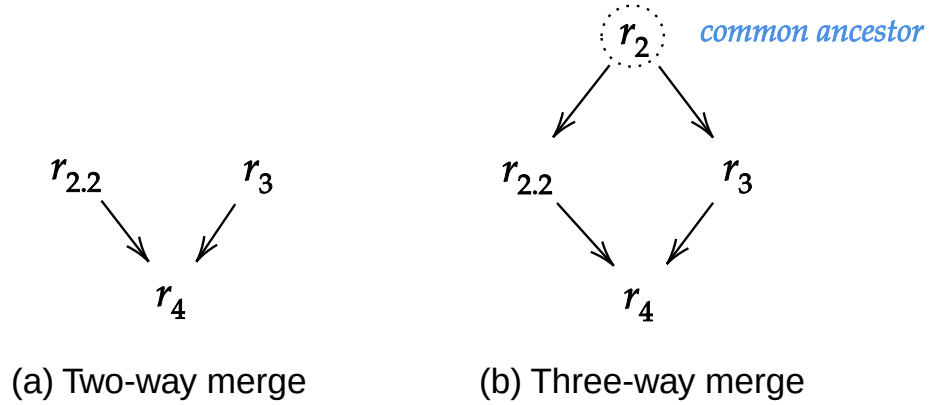
Particularly in the open-source community, but also in the industry, the number of software projects using DVCSs, rather than traditional CVCSs, has rapidly increased (BIRD et al., 2009; BRINDESCU et al., 2014). At the end of 2018, GitHub (GITHUB, 2018), the most popular collaborative hosting site built on top of Git (KALLIAMVAKOU et al., 2014), hosted over 100M repositories, whereas SourceForge (SOURCEFORGE, 2018), the main hosting site for Subversion, had only 430K repositories.

2.2 MERGE APPROACHES

A large-scale— and potentially distributed— software development, where independent lines of development are implemented by different programmers, requires support for merging of artifact revisions (PERRY; SIY; VOTTA, 2001). Different merge techniques have been proposed over the last decades, and they can be categorized according to a number of orthogonal dimensions. One of the dimensions concerns how the differences between two revisions are extracted and compared when they are merged. *Two-way merge* tries to merge two revisions of a software artifact without using any additional information from the branching history. Conversely, *three-way merge* leverages the information in the common ancestor from which the revisions originated (MENS, 2002). A three-way merge scenario is a set that consists of a common ancestor revision and its derived revisions.

To illustrate the difference between two-way and three-way merge, let us revisit the scenario introduced in Figure 2, where r_4 is created from $r_{2.2}$ and r_3. In the two-way merge, only the content of the version of artifacts provided by $r_{2.2}$ and r_3 are used during the merge process (see Figure 3(a)). This information is insufficient to determine if differences between $r_{2.2}$ and r_3 are caused by a line addition, modification or removal in either one of the evolved revisions or by a simultaneous change in both of them. With three-way merge, the information in the common ancestor, the revision r_2, is used to decide where a change came from, and whether a conflict should be reported or not (see Figure 3(b)). This makes three-way merge more powerful, and, as a consequence, the vast majority of modern merge tools avail themselves of three-way merge instead of its two-way variant (PERRY; SIY; VOTTA, 2001; MENS, 2002; O'SULLIVAN, 2009).

Figure 3 – Two-way merge and three-way merge



Another dimension to which merge techniques can be classified is on how software artifacts are represented. Over the last years, two classes of version control systems have emerged: 1) VCSs that perform merge based on plain text, and 2) VCSs that operate on more abstract and structured representations (ESTUBLIER et al., 2005; APEL et al., 2011).

2.2.1 Unstructured Merge

VCSs that are in the aforementioned first class use *unstructured merge* tools, which represent program as text files. The most popular approach is to make use of text-line merging, where lines of a text are treated as indivisible units (HUNT; MCILROY, 1975). Textual, line-based merge tools have been widely used in commercially available VCSs. The main reason is that such tools are typically language-independent, since any software artifact can be considered as a piece of text, making them very flexible. Besides generality, performance is another strength of unstructured merge, once it applies the same algorithm to any non-binary file, regardless of the programming language used. Examples of VCSs that, by default, use unstructured merge tools include CVS, Subversion, and Git (MENS, 2002; APEL et al., 2011).

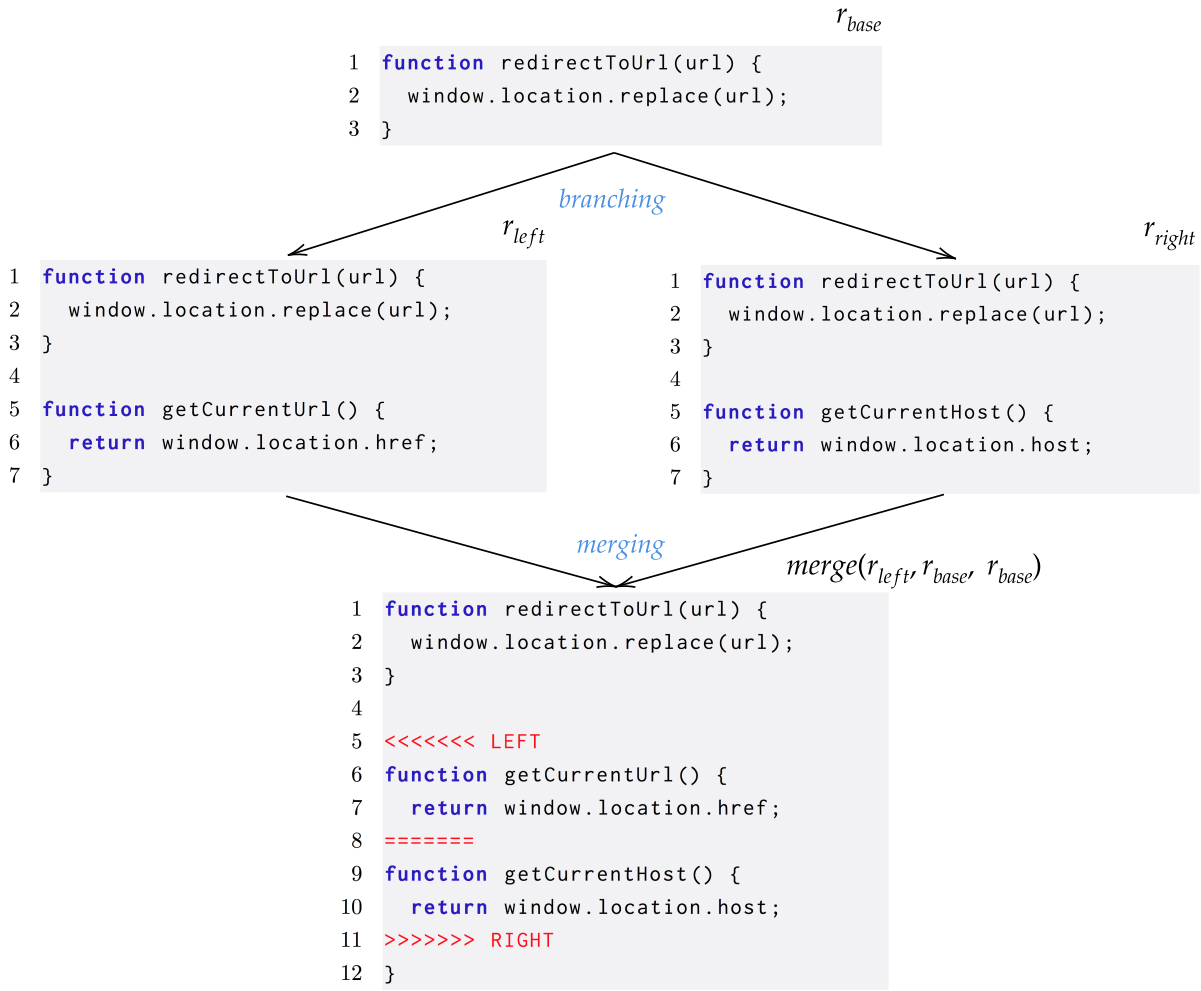
In general, when merging two revisions, textual, line-based, three-way merge tools compare their files in relation to their common ancestor, and express their differences as a minimum list of line changes (HUNT; MCILROY, 1975). For each set of differing lines, called chunks, the merge algorithm checks if there are subsets that are common to the three revisions, splitting the content of the chunk into two different areas. If two revisions change or extend text in the same area, the tool reports a conflict. (KHANNA; KUNAL; PIERCE, 2007). Conflicts are usually marked by a `<<<<<<<` line, followed by the first developer's version, then `=======` followed by second developer's version, then a `>>>>>>>` line.

Figure 4 presents an example of three-way merge performed by an unstructured tool. Initially, we have a simple program, written in JavaScript, with a single function called `redirectToUrl`. The revision of that initial version of the program is referred by r_{base}. Then, a programmer creates a branch (r_{left}) from r_{base} to add a new function, called `getCurrentUrl`. In parallel, another developer also creates a new development line (r_{right}) from the same base program, adding another function, `getCurrentHost`, but also changing the body of the function `redirectToUrl`. At some point, the two branches are merged in order to combine both contributions. Regarding the change on the `redirectToUrl`'s body introduced by the second developer, the thee-way merge is able to detect that it was a change made only by that developer, while the other one kept the implementation from the base revision, so no conflict is reported for that chunk, which would happen if a two-merge were used. However, since the revisions r_{left} and r_{right} add functions to the same text area, a conflict is reported, as shown in Figure 4. The output of the merge is an indication for the developers that they need to manually resolve the conflict, and decide which fragments (zero, one or both functions) should be in the merged version.

That example illustrates a limitation of the conflict resolution employed by unstructured merge. The tool identifies that two different text fragments are added to the same area of the program, but it does not know that those fragments are actually JavaScript functions, and that a merge of them should be straightforward because their order in

the program can be permuted without changing its behaviour, as explained in the next section (Section 2.3). As a result of unstructured merge not using knowledge about the structure of the program and the syntax of languages in which they are written, it might report spurious conflict, miss relevant conflicts, and generate syntactically or semantically incorrect output (HORWITZ; PRINS; REPS, 1989; BUFFENBARGER, 1995).

Figure 4 – Example of unstructured merge



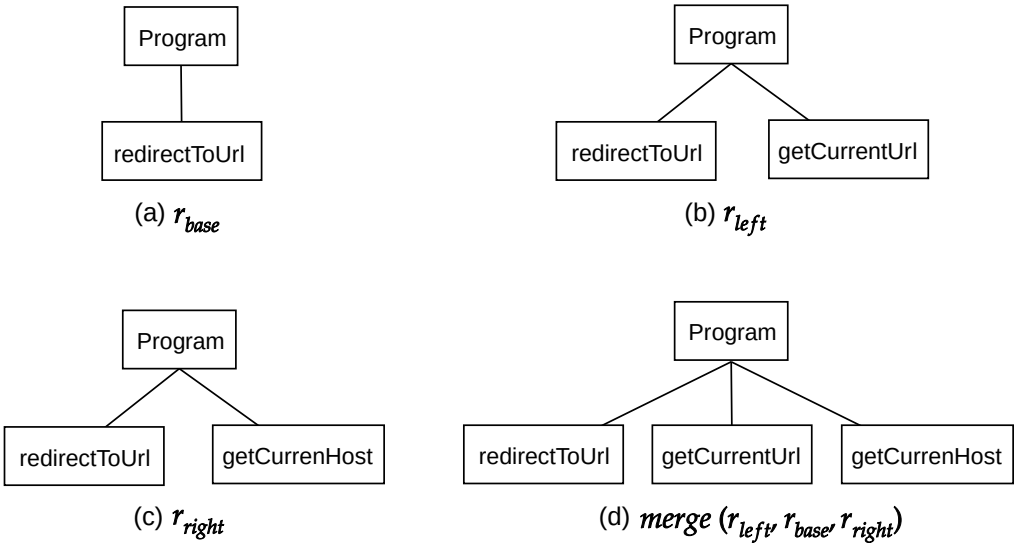
2.2.2 Structured and Semistructured Merge

In contrast to unstructured merge, many tools have been proposed that leverage information on the language a program was written in to automatically resolve as many conflicts as possible (MENS, 2002). Among the first ones, Westfechtel (1991) and Buffenbarger (1995) created tools that use structured information of a program, such as the context-free and context-sensitive syntax when performing merge and resolving conflicts. Then, several other tools have been developed that are tailored to a particular programming language, implementing a *structured merge*. For example, Grass (1992) proposed a merge tool specific to C++, whereas Apiwattanapong, Orso and Harrold (2007) created a tool

specific to Java. The lack of generality of structured version control systems, since they are specially built to deal with a particular language, is pointed out as one of the limitations of structured merge. An additional limitation of structured merge concerns performance, which is affected by the complexity of handling full syntax trees.

To find a suitable balance between structured and unstructured merge, Apel et al. (2011) proposed an approach called *semistructured merge*. The idea is to partially represent programs as trees, and to provide information on how nodes of specific types (e.g., functions or classes) and their subtrees are merged. These trees, in which software artifacts are represented, are called *program structure trees*. Figure 5 illustrates simplified program structure trees that represent the programs shown in Figure 4, including the base, left, and right revisions. Such trees include some, but not all structural information of the programs. For instance, there are no nodes that represent expressions or statements from the body of the functions; they are contained as plain text in the leaves. For Java, one of the programming languages that were initially supported by semistructured merge tool implementations, only classes, methods, and fields are represented as nodes in the program structure trees. Concerning JavaScript, functions can be represented as nodes, as shown in Figure 5, but the inclusion of additional nodes depends on implementation decisions, which are discussed in the next chapter (Chapter 3).

Figure 5 – Example of simplified program structure trees from semistructured merge



The decision of which types of structural element are represented by a node depends on the expressiveness we want to to have with semistructured merge. With `FSTMerge`, the prototype implemented by Apel et al. (2011), the specification of elements that should be either represented as nodes or as plain text in leaves is given by an annotated grammar of the language we want to support. Production rules can be annotated with `@FSTNonTerminal` and `@FSTTerminal`. Both annotations define, in common, that the annotated element is 1) represented as a node in the corresponding program structure tree,

and 2) that the order of such node is arbitrary. The difference between `@FSTNonTerminal` and `@FSTTerminal` comes in the definition on how to represent subelements of the annotated element. `@FSTNonTerminal` specifies that subelements are represented as subnodes. For example, in Java, class declarations can be annotated with `@FSTNonTerminal` because the order of class declarations in a file does not matter, and it may contain further classes, methods, etc. In turn, an element annotated with `@FSTTerminal` has its subelements represented as plain text. The idea of the proposed architecture is making it relatively easy to include new languages by providing specific annotated grammars, but using a single generic engine (APEL et al., 2011).

Elements annotated as non-terminal (`@FSTNonTerminal`) are merged via a process called *superimposition* (APEL; LENGAUER, 2008), which merges two trees by composing their corresponding nodes, starting from the root, and proceeding with subtrees in a recursive fashion. In that recursion, two subnodes are composed to form a new node only when they have the same name and type. If a node does not have a matching element to be composed with, it is added as a new child to the composed parent node. When reaching elements annotated as terminal ones (`@FSTTerminal`), they are, by default, merged by a traditional textual, line-based merge algorithm, as used in unstructured merge. Alternatively, dedicated conflict handlers can be added to define further merge and conflict resolution strategies (APEL et al., 2011). In Figure 5(d), we have the tree obtained via superimposition of the base, left, and right revisions, having the function `redirectToUrl` from the base revision, with the change the second developer made (not seen in the tree), along with the functions added by both developers. In that case, no conflicts are reported, differently from unstructured merge, because of the ability of the semistructured merge to resolve the so-called *ordering conflicts*. Semistructured merge can automatically solve such conflicts based on the observation that the order of some elements, in a given programming language, does not matter. Because of that, superimposition simply adds the newly introduced functions next to each other in the merged tree in any possible order (see Figure 5(d)).

In summary, the proposal of semistructured merge is to leverage information from program structure trees to merge revisions with fewer conflicts than unstructured merge, while not having to handle the merge of expressions and statements from function (as in JavaScript) or method (as in Java) bodies, decreasing the complexity of the merging algorithm, as well as positively affecting its performance, when compared to full structured merge tools (APEL; LESSENICH; LENGAUER, 2012). This balance, between unstructured and structured merge, allows semistructured merge to support a greater number of programming languages than structured merge systems, and to provide a more effective conflict resolution, in most cases, when compared to unstructured merge tools (APEL et al., 2011; CAVALCANTI; ACCIOLY; BORBA, 2015).

2.3 JAVASCRIPT OVERVIEW

JavaScript is one of the core technologies of the World Wide Web, along with HTML and CSS. HTML is used to specify the content of Web pages, CSS is used to specify their presentation, and JavaScript is used to specify their behaviour (FLANAGAN, 2011). JavaScript is a programming language that was originally created in 1995, by Brendan Eich, to be released with the Web browser Netscape Navigator 2.0. In 1996, Netscape submitted JavaScript to ECMA Internationals, an European standards organization, in order to create a standard specification of the language, which led to the release of the first edition of the ECMAScript standard. The next major version of the language was ECMAScript edition 5 (ES5), first published in 2009, and, then, updated in 2011 (ECMA, 2011). In 2015, the ECMAScript edition 6 (ES6) was published (ECMA, 2015), which is the latest version to bring major changes to the language, and that is fully supported by modern browsers.

One of the main characteristics of JavaScript is that it is a dynamically typed and object-based scripting language (MARTINSEN et al., 2011). An object, in JavaScript, is a set of properties that behaves similarly to an associate array, mapping strings to values, which may be data or functions (RICHARDS et al., 2011). However, unlike Java and C#, JavaScript does not have native support for classes, but it provides inheritance by means of prototypes, that allow objects to inherit properties directly from other objects. Another characteristic of JavaScript is that functions are first-class objects, making JavaScript a multi-paradigm language, that supports object-oriented, imperative, and functional programming styles (STEFANOV, 2010).

A JavaScript program needs a runtime environment to be executed. The most common environment, which was the only one for a long time, is the Web browser, where JavaScript can be run by means of scripts to provide dynamic behaviour to HTML pages. Besides dynamic interaction on Web pages with users, the language is capable of interacting asynchronously with servers (POWELL, 2008). JavaScript code can be embedded in HTML pages in four different ways (SILVA et al., 2017): 1) directly in an HTML file, within a `<script>` tag, 2) from an external file specified in the `src` attribute of a `<script>` tag, 3) in an HTML event handler attribute (e.g., `onclick`), and 4) in a URL that specifies the `javascript` protocol. Recently, JavaScript started to be used in a new environment: servers. This became possible thanks to Node.js, a framework, based on the Google Chrome engine, that provides a complete server-side environment for creating high-performance and scalable programs written in JavaScript (TILKOV; VINOSKI, 2010).

2.3.1 Grammar Summary and Basic Syntax

The core of the current specification of JavaScript is defined in the 5th edition of ECMAScript, ES5, which is a fully compatible subset of ES6 (ECMA, 2015). Understanding

the basics of ES5 syntax is a gateway to understand how JavaScript programs are structured. Figure 6 presents a summary of the grammar of ES5. Essentially, considering that edition of ECMAScript, a program in JavaScript is simply a sequence of `SourceElements`, which can be either a `Statement` or a `FunctionDeclaration` (ECMA, 2011). This is an important distinction of JavaScript from many other object-oriented languages, because we can have statements at the top level of a program, without being wrapped in a declaration, such as a method. A programmer can declare a function, and invoke it right after in a statement; actually, it is possible to call a function that was defined from a `FunctionDeclaration` even before its declaration, as we explain later in this section.

Figure 6 – Grammar summary for a program in ES5

Program :
SourceElements$_{\text{opt}}$

SourceElements :
SourceElement
SourceElements *SourceElement*

SourceElement :
Statement
FunctionDeclaration

Statement :
Block
VariableStatement
EmptyStatement
ExpressionStatement
IfStatement
IterationStatement
ContinueStatement
BreakStatement
ReturnStatement
WithStatement
LabelledStatement
SwitchStatement
ThrowStatement
TryStatement

Source: ECMA (2011)

Listing 2.1 shows an example of a JavaScript program with a function declaration, defining the function `redirectToGooglePage`, and two statements, one declaring a variable (`VariableStatement`), and another one invoking that function (`ExpressionStatement`). JavaScript, up to ES5, uses functions to manage scope. A variable declared in a function is local to that function, being not available from outside. For example, the variables `googleHost` and `url` cannot be used outside of `redirectToGooglePage`. In turn, variables declared outside of any function (e.g., `mapsPath`), or that are simply used without being declared (when not using the keyword `var`), are considered to be global, and, thus, can be used anywhere in the program (even before their declaration, with the caveat that their value will be `undefined` at that point). In every JavaScript environment, there is a global object that is accessible from any point of a program. When a global variable is created, it becomes a property of that global object. In case of Web browsers, just for convenience,

there is an extra property of the global object called `window`, that points to the global object itself (STEFANOV, 2010). In our example, `window` is referred to access the Location API (line 6), which allows page redirections, provided by browsers as a global variable.

Listing 2.1 – Example of JavaScript program with function declaration and statements

```
1  // Function declaration
2  function redirectToGooglePage(path) {
3    var googleHost = 'https://www.google.com';
4    var url = googleHost + '/' + path;
5
6    window.location.replace(url);
7  }
8
9  // Statement
10 var mapsPath = 'maps';
11
12 // Statement
13 redirectToGooglePage(mapsPath);
```

Figure 7 – Grammar for function definition in ES5

FunctionDeclaration:
function *Identifier* **(** $FormalParameterList_{opt}$ **)** **{** *FunctionBody* **}**

FunctionExpression:
function $Identifier_{opt}$ **(** $FormalParameterList_{opt}$ **)** **{** *FunctionBody* **}**

FormalParameterList:
Identifier
FormalParameterList **,** *Identifier*

FunctionBody:
$SourceElements_{opt}$

Source: ECMA (2011)

Functions, in JavaScript, can be defined in two different ways: 1) by means of a `FunctionDeclaration`, as seen previously, and 2) in a `FunctionExpression`. Part of the ES5 grammar specification of function definition is shown in Figure 7. A function expression is quite similar to, and has almost the same syntax, a function declaration, but it is defined inside an expression. In syntactic terms, the main difference between a function declaration and a function expression is where they can appear in a program. Since the latter is an expression, it can appear, for instance, as a value assigned to a variable or as an argument of a function, whereas a function declaration cannot. Moreover, while a func-

tion declaration always has a name, function expression may be named, when an identifier is provided, or not. Unnamed function expressions are commonly known as *anonymous functions* (STEFANOV, 2010). As indicated in the grammar, both function declaration and expression have a body that can include one or more `SourceElements`. It means that functions can have further nested functions, either by having nested function declarations or function expressions defined in inner statements. Similarly to what happens to variables, a nested declared function also has a scope local to the parent function.

In Listing 2.2, we have a function (`generateGoogleUrl`) that is declared inside of another function (`redirectToGooglePage`). Due to function scoping, `generateGoogleUrl`, as well as `url`, are visible only from the body of `redirectToGooglePage`. It is important to note that if `url` were used in line 8 without the keyword `var`, it would be a global variable and, thus, visible outside of `redirectToGooglePage`. An example of unnamed function expression is provided by Listing 2.3, where a function is created at runtime, and assigned to a variable (`redirectToUrl`). The variable, from that point, can be used to invoke a function as if it had been created by a function declaration, passing arguments within parenthesis (line 6).

Listing 2.2 – Example of nested function declaration

```
1  // Outer function declaration
2  function redirectToGooglePage(path) {
3    // Inner function declaration
4    function generateGoogleUrl(path) {
5      return 'https://www.google.com/' + path;
6    }
7
8    var url = generateGoogleUrl(path);
9    window.location.replace(url);
10 }
11
12 redirectToGooglePage('maps');
```

Listing 2.3 – Example of unnamed function expression assigned to a variable

```
1  // Function expression assigned to a variable
2  var redirectToUrl = function(url) {
3    window.location.replace(url);
4  };
5
6  redirectToUrl('https://www.google.com');
```

Regardless of the mechanism from which functions are created, either from a declaration or an expression, they are objects, and, then, can be assigned to variables, can be passed as argument to other functions, and can also be returned by other functions.

In particular, a function that is passed as an argument to another function, which is expected to invoke it either immediately or after some time, is a pattern in JavaScript called *callback* (GALLABA; MESBAH; BESCHASTNIKH, 2015).

The syntax of ES5 presented in this subsection has been kept in ES6, which just extended the grammar with new elements, but the specification introduced changes on how variables can be scoped. In ES5, a variable can only be declared using the `var` keyword, as seen in previous examples. ES6 introduced the `let` and `const` keywords that allow the programmer to declare a variable that exists only within a block (similarly to what happens in Java and C#). Function scope, however, is still employed when using `var`, so the semantics of such declaration was preserved. Among other syntactic additions to the language, ES6 provides a support to classes, but that are, actually, just syntactic sugar over an existing prototype-based pattern, which uses functions, from ES5 and previous versions (ECMA, 2015; ZAKAS, 2016; SILVA et al., 2017).

2.3.2 Function Declarations vs. Function Expressions

JavaScript developers often have to decide whether they should use a function declaration or a function expression. There are scenarios in which syntactically we cannot use declarations. For example, when specifying directly a function object as an argument (a callback) or when defining directly a function as a value of an object property. In such scenarios, only a function expression is allowed. But, when creating a function to be referred by a name, programmers tend to use interchangeably between a function declaration and a function expression assigned to a variable (as in Listing 2.3), being, in many cases, just a matter of style to use one over the other. Nevertheless, there is a fundamental difference between a function declaration and a variable that holds a function expression, and it lies in the *hoisting* behaviour (STEFANOV, 2010).

Hoisting is a JavaScript mechanism that loads variable and function declarations into the memory prior to a program execution. In terms of behaviour, variables and function declarations are moved to the top of the function to which they are scoped to, or to the top of the program in case they are not in a function (MOZILLA, 2018). For variables, it means that they can be referred before its declaration, but their value is, until an assignment is reached, `undefined`. For functions, on the other hand, there is no separation between declaration and a corresponding function definition, so the definition of the function also gets hoisted. That means that a function can be used before we declare it. Listing 2.4 illustrates how hoisting works for both function declaration and expression. We try to use two functions, each one created by one of the methods, before their definitions. The function created by a declaration (`getProtocol1`) returns the expected value when invoked in line 3. In line 5, we can see that the variable created to hold a function expression (`getProtocol2`) can be referred without any `ReferenceError` (which is thrown when trying to access a variable that does not exist), but it has `undefined` as value, which

causes a `TypeError` when trying to invoke it in line 6.

In general, function declarations behave similarly to methods in object-oriented languages, such as Java and C#, where the order of them does not matter to the execution of the program. Function expressions stored in variables, in turn, are similar to functions in C and Python, which can typically be used only after their definition. When considering the implementation of a semistructured merge tool, where we need to specify elements whose order may be arbitrary, without changing the behaviour of the program, it is important to take that difference, between functions created either by declarations or expressions, into consideration.

Listing 2.4 – Invocation of function before its definition

```
1  var url = 'https://www.google.com.br';
2
3  console.log(getProtocol1(url)); // "https"
4
5  console.log(getProtocol2); // undefined
6  console.log(getProtocol2(url));// TypeError: getProtocol2 is not a function
7
8  // Function declaration
9  function getProtocol1(url) {
10   return url.split('://')[0];
11 }
12
13 // Function expression
14 var getProtocol2 = function(url) {
15   return url.split('://')[0];
16 }
```

An additional scenario in which a function expression is commonly used is in a pattern called *Immediately Invoked Function Expression* (IIFE). 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, once that any variable created in the function expression will not be available outside of it, as long as the keyword `var` is employed (STEFANOV, 2010; MOZILLA, 2018). Listing 2.5 shows an example of an IIFE that is used to clear a text field in an HTML page, and the variable created in it (`usernameField`) is no longer available after the function execution (line 6).

Listing 2.5 – Example of Immediately Invoked Function Expression

```
1 (function () {
2     var usernameField = document.getElementById('username');
3     userNameField.value = '';
4 })();
5
6 console.log(usernameField); // undefined
```

When taking the implementation of a semistructured merge tool based on `FSTMerge` into account, the interchangeable usage of function declaration and a function expression assigned to a variable might affect the effectiveness of this tool. In particular, the decision of which grammar elements (`Statement`, `FunctionDeclaration`, etc.) are represented as nodes in program structure trees has an impact on how a semistructured merge behaves. For example, if a `Statement` is not represented as node— being represented as plain text in a leaf instead—, while `FunctionDeclaration` is, an implication is that a function expression assigned to a variable will not exploit superimposition in the same manner as a function declaration will do. Additionally, if a `Statement` is represented as plain text in a terminal node, having a function declaration wrapped into an IIFE make this function not visible as a node in a program structure tree, despite of a definition of `FunctionDeclaration` element as a non-terminal node. Further discussion about the implications of using function declarations or function expressions, according to a given semistructured merge tool implementation, is presented in Chapter 4.

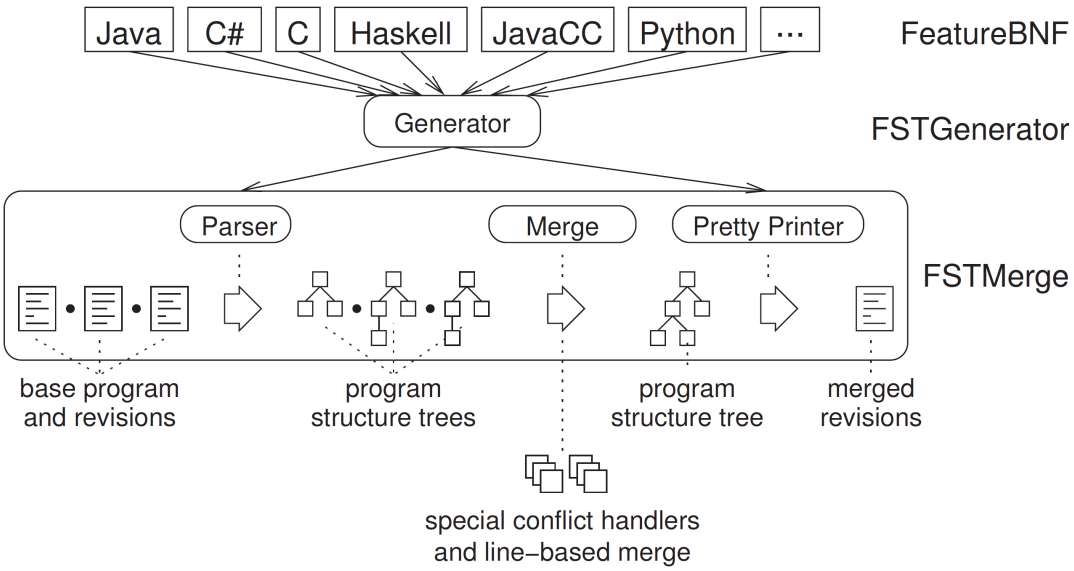
3 SEMISTRUCTURED MERGE TOOL FOR JAVASCRIPT

In this chapter, we explain our approach to implement a semistructured merge tool for JavaScript, which enables us to evaluate how the semistructured approach works with a language that is quite relevant for the industry and has not been investigated in previous studies. Section 3.1 describes different versions of semistructured tools which were implemented in this work. Initially, we present a version that simply instantiates the `FSTMerge` architecture by annotating an off-the-shelf grammar for a specific version of JavaScript, and, then, we discuss limitations in that architecture when dealing with individual statements (Subsection 3.1.1). To overcome such limitations, we implemented two new versions of the tool— which are further detailed, respectively, in the Subsections 3.1.2 and 3.1.3—, based on adaptations made on the original grammar and `FSTMerge` merge algorithm. We conclude this chapter by describing different types of conflicts that semistructured, considering implementations proposed in this work, and unstructured merge tools are able to detect or not when merging JavaScript software artifacts (Section 3.2).

3.1 DESIGN AND IMPLEMENTATION

The architecture of `FSTMerge` takes advantage of a tool infrastructure called `FeatureHouse` (APEL; KäSTNER; LENGAUER, 2009), as illustrated in Figure 8.

Figure 8 – FSTMerge architecture



Source: Apel et al. (2011)

One of the `FeatureHouse` tools, `FSTGenerator`, takes a Backus–Naur form (BNF) grammar written in a specific format called `FeatureBNF` to generate an LL(k) parser, which

is responsible to produce program structure trees, and a corresponding pretty printer (APEL et al., 2011). A new semistructured tool, for a given language, can be created by using the generated parser to obtain trees for revision programs, the `FSTMerge` merge engine to perform semistructured merge, and, finally, the generated pretty printer to write files for merged revisions to disk.

By integrating the parser and pretty printer generated by `FSTGenerator`, and the (possibly modified) `FSTMerge` merge engine, we implemented different versions of a semistructured tool for JavaScript, called `jsFSTMerge`. The differences among such versions result from modifications we made to the `FeatureBNF` grammar used as input for the `FSTGenerator`, and to the semistructured merge algorithm itself. Each version of the tool is referred by a number (e.g., `jsFSTMerge v2`).

In this work, we have decided to focus on the 5th edition of ECMAScript (ES5) as an initial effort to have a semistructured merge tool for the language. The decision of implementing a tool that supports only ES5 was made mainly for the following reasons: 1) ES5 is still the latest specification of JavaScript that is supported by essentially all Web browsers that are currently in use (ECMA, 2015; ZAKAS, 2016), 2) ES5 provides a grammar that is less complex than the ones from newer specifications, simplifying the scope of this first implementation of semistructured merge for JavaScript, and 3) since newer versions of the language, ES6 in particular, are fully backwards compatible with ES5, it would already be necessary to properly support ES5 syntax when targeting those more recent specifications. Furthermore, the implementation of a semistructured merge tool that supports ES6, or newer versions of ECMAScript, is purely an extension of one that supports ES5, which can be accomplished as a continuation of this work.

3.1.1 jsFSTMerge v0: Annotating an Off-the-shelf Grammar

Our first approach to implement a semistructured merge tool for JavaScript was working on the original grammar specification of ES5 as published in ECMA (2011). The first step was writing the grammar, including lexical definition, in the `FeatureBNF` format. The next step was deciding on how to annotate production rules, which basically involves selecting elements to be represented as nodes in program structure trees, considering restrictions and assumptions set by the `FSTMerge` engine.

Listing 3.1 shows an excerpt of the annotated grammar[1], in the `FeatureBNF` format, for the first version of a semistructured tool denominated `jsFSTMerge v0`. We annotate the rule for function declarations with `@FSTNonTerminal` (lines 9-10) because, as discussed in the previous chapter, the order of function declarations, within their parent node (either `Program` or another `FunctionDeclaration`), does not matter as a result of hoisting, and, more importantly, because function declarations may contain further functions, and so

[1] https://github.com/AlbertoTrindade/jsFSTMerge/blob/version-0/grammars/javascript_merge_fst.gcide

on. The annotation parameter `name` is used to assign a name to the corresponding nodes in the program structure tree, so that they can be exploited by matching during the merge process; an identifier is used as a name for function declarations. For statements, to keep the semistructured approach of not having a full representation of the program in the tree, we initially annotate their rules with `@FSTTerminal` (lines 6-7). The annotation parameter `merge` defines that the content of statements is merged by a textual, line-based unstructured merge. The `@FSTInline` annotation is used (line 12) to simplify the grammar writing, replacing an element with its composition (APEL; KäSTNER; LENGAUER, 2009).

Listing 3.1 – Excerpt of annotated grammar for `jsFSTMerge v0`

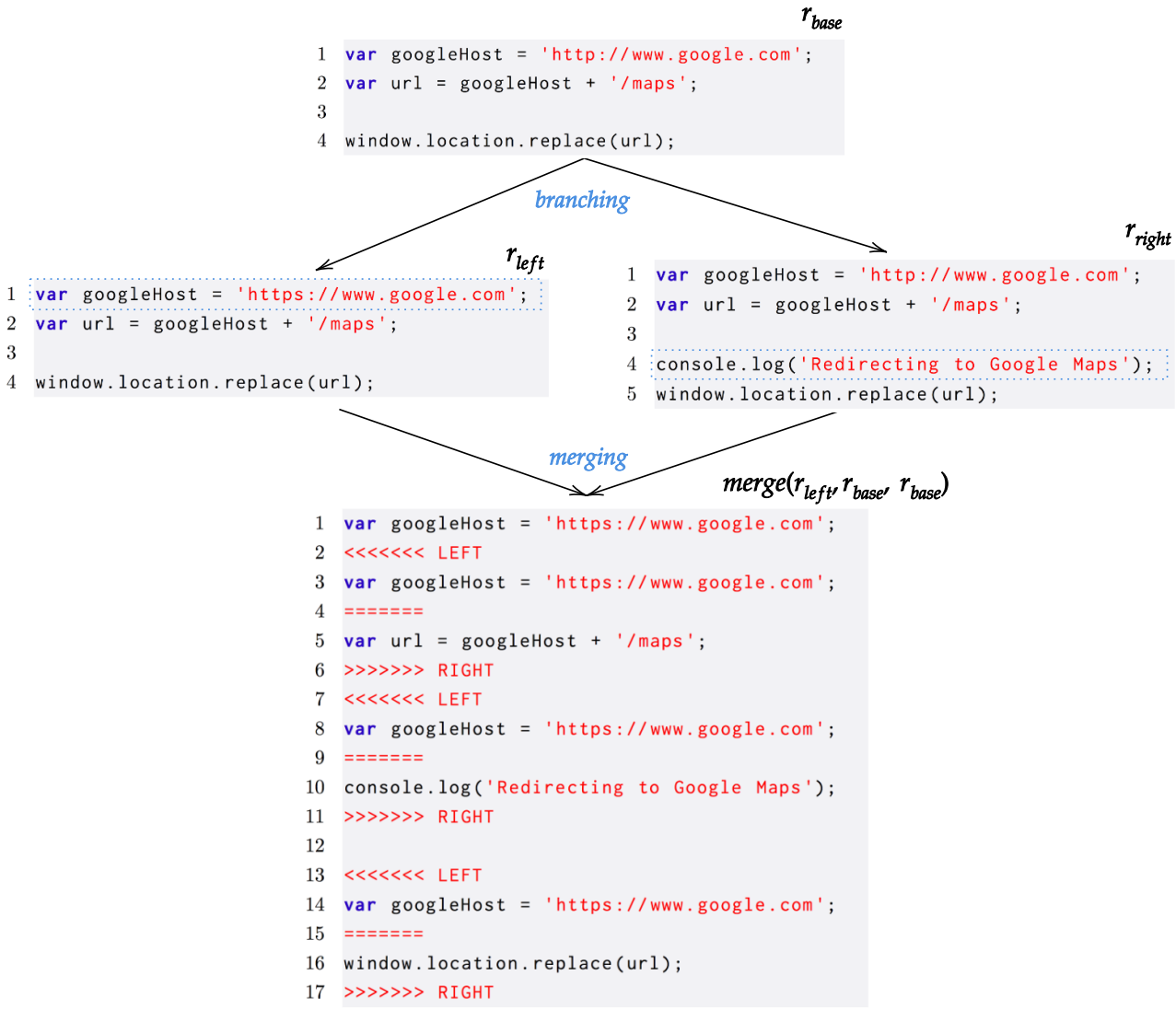
```
1  ...
2  @FSTNonTerminal()
3  Program: ( SourceElement )* <EOF>;
4
5  SourceElement:
6      @FSTTerminal(merge="LineBased")
7      Statement
8      |
9      @FSTNonTerminal(name="{Identifier}")
10     FunctionDeclaration;
11
12 @FSTInline
13 FunctionDeclaration:
14     Function
15     Identifier
16     "(" [ FormalParameterList ] ")"
17     "{" FunctionBody @! "}";
18
19 @FSTTerminal(merge="LineBased")
20 Function: "function"<NONE>;
21
22 @FSTTerminal(name="{<IDENTIFIER_NAME>}")
23 Identifier: <IDENTIFIER_NAME>;
24
25 @FSTTerminal(merge="LineBased")
26 FormalParameterList: Identifier ( "," Identifier )*;
27
28 @FSTNonTerminal()
29 FunctionBody: ( SourceElement )*;
30 ...
```

However, annotating individual statements as terminal nodes has implications that jeopardize the correctness of merged revisions produced by superimposition (see Subsection 2.2.2). The following issues come up when having statements annotated with `@FSTTerminal`:

- Superimposition of trees requires that every element must have a name (APEL; LENGAUER, 2008). While function declarations naturally have a name (an identifier defined after `function` keyword), statements do not (APEL; HUTCHINS, 2010; APEL et al., 2010). In general, it is difficult to uniquely identify statements. For instance, there is no way, by means of grammar annotation, to properly name an `IfStatement` in such a way that the same element from a different revision, with potential few differences, can be matched via superimposition. In fact, by default, all nodes are named with a constant "-" when not defined otherwise; so, from the grammar defined in Listing 3.1, all statements have the same name, which causes superimposition to mistakenly match nodes that do not refer to the same statement. The consequences of using `jsFSTMerge v0`, the tool built on top of the parser generated from that grammar, may include 1) reporting spurious conflicts, and 2) unsoundly discarding statements from revisions. An example of the former is shown in Figure 9, that presents a merge scenario in which the revision r_{left} edits the first statement from r_{base} (line 1), whereas the revision r_{right} adds a new statement (line 4). Several spurious conflicts are reported by `jsFSTMerge v0` because, when performing a three-way merge, each statement from the revision r_{right} is matched against the same statement of the resulting merge of r_{base} and r_{left}. This happens because there are multiple `Statement` nodes with the same default name, so, instead of returning the desired statement when looking up nodes by name and type, superimposition returns the first `Statement` node that has the default name. Moreover, as a result of the same problem of having multiple statements with the same default name, the discard of statements introduced by the left revision would happen if it did not change the first `Statement` node (line 1). The outcome, in this case, is that superimposition would match all statement nodes from r_{right} against this unaltered node, causing the final merge to have the same statements as r_{right}, thus unsoundly discarding changes introduced by r_{left}, while not reporting conflicts.

- For a parent node marked as non-terminal, `FSTMerge` engine considers that not only child elements whose production rules are annotated with `@FSTNonTerminal`, but also the ones annotated with `@FSTTerminal` have an order that does not matter. The order of child nodes of a common parent is assumed to be arbitrary, that is, it may change without affecting the semantics of the program. But, according to the semantics of JavaScript, the order of statements does matter, once they may cause side effects. For example, if, during superimposition, an `IfStatement` that refers a certain variable is moved after an `ExpressionStatament` that contains an assignment to that same variable, the behaviour of the program could have been changed. The order of statements needs to be preserved in order to ensure that the program semantics is preserved; and it is not guaranteed by the generic merge engine from `FSTMerge` when relying purely on `@FSTTerminal` annotation.

Figure 9 – Example of spurious conflicts reported by `jsFSTMerge v0`



To try to solve the first aforementioned issue, which comes from the lack of unique and unambiguous names for statements, one could propose using the annotation parameter `name`, supported by `FeatureBNF`, to specify the name of statements as `TOSTRING`, instead of using the default "-". When using this value as name, the annotated elements are named after their textual content, i.e., two elements that have the same type are matched if they have exactly the same textual content (APEL; KäSTNER; LENGAUER, 2009). A similar, but possibly more efficient, approach is obtaining a hash from the textual content of such elements, and using this hash as a name, which would be shorter than the actual full content. However, using textual content to match nodes makes the merge too sensitive to minor changes, potentially generating duplicated statements. For example, if a developer edits a statement, changing the value used in the initialization of a variable, this statement will be seen, by semistructured merge, as a new node in the program structure tree, once its textual content becomes different, causing the program to have duplicated variables.

To try to solve both naming and ordering issues, a possible solution is modifying the semistructured merge algorithm to— before starting the superimposition— assign names to statement elements based on their position as children of a parent node (and, actually, a similar solution is implemented for the versions of `jsFSTMerge` we discuss in the next subsections), but there are, at least, two problems with that approach to deal with statements. The first one concerns granularity. Keeping the representation of individual statements as terminal nodes in program structure trees implies a granularity that is too fine, which leads to more complex trees and a worse performance, behaving similarly to structured merge (APEL et al., 2011). Additionally, matching individual statements in the superimposition makes the merge less resilient to addition or removal of statements. For instance, if a developer adds a variable assignment at the same position as another statement, it might cause that assignment to be matched against a different type of statement from a different revision, either introducing duplicated statements or generating spurious conflicts that would not be reported by unstructured merge.

Considering those issues and implications, we have decided to not include `jsFSTMerge v0` in the empirical study that is presented in the next chapter (Chapter 4), because that tool is too far from being reliable enough to be used as a replacement for commercial unstructured tools. Instead, we decided to create a new version of the tool by first introducing changes to the grammar to better handle statements, already diverting from a "pure" implementation of support for a new programming language on top of `FSTMerge`, in which we are supposed to simply annotate existing grammars in a plug-in fashion (APEL et al., 2011). In fact, directly annotating an off-the-shelf grammar and instantiating `FSTMerge` does not lead to an effective semistructured merge tool for JavaScript.

3.1.2 jsFSTMerge v1: Annotating an Adapted Grammar and Renaming Nodes

To provide a reasonable level of granularity as well as to overcome some of the problems that have arisen in `jsFSTMerge v0`, we have decided to group consecutive statements into a `StatementList` element, which is already defined in the original specification of ES5, but only used in a few production rules (e.g., rule for `Block` element) (ECMA, 2011). `StatementList` is annotated as a terminal node, so the statements are no longer individually represented as nodes in the program structure tree. Instead, there is only a single node, for each sequence of statements separated by function declarations, that is a opaque leaf with statements as textual content. The implementation of that change can be seen in lines 7-13 from Listing 3.2. This new grammar[2] is used to generate a parser for two new versions of our semistructured merge tool: `jsFSTMerge v1` and `jsFSTMerge v2`. The difference between these versions come from further adaptions made to the `FSTMerge` merge algorithm, which are detailed later on this text.

[2] https://github.com/AlbertoTrindade/jsFSTMerge/blob/master/grammars/javascript_merge_fst.gcide

Listing 3.2 – Excerpt of annotated grammar for `jsFSTMerge v1` and `jsFSTMerge v2`

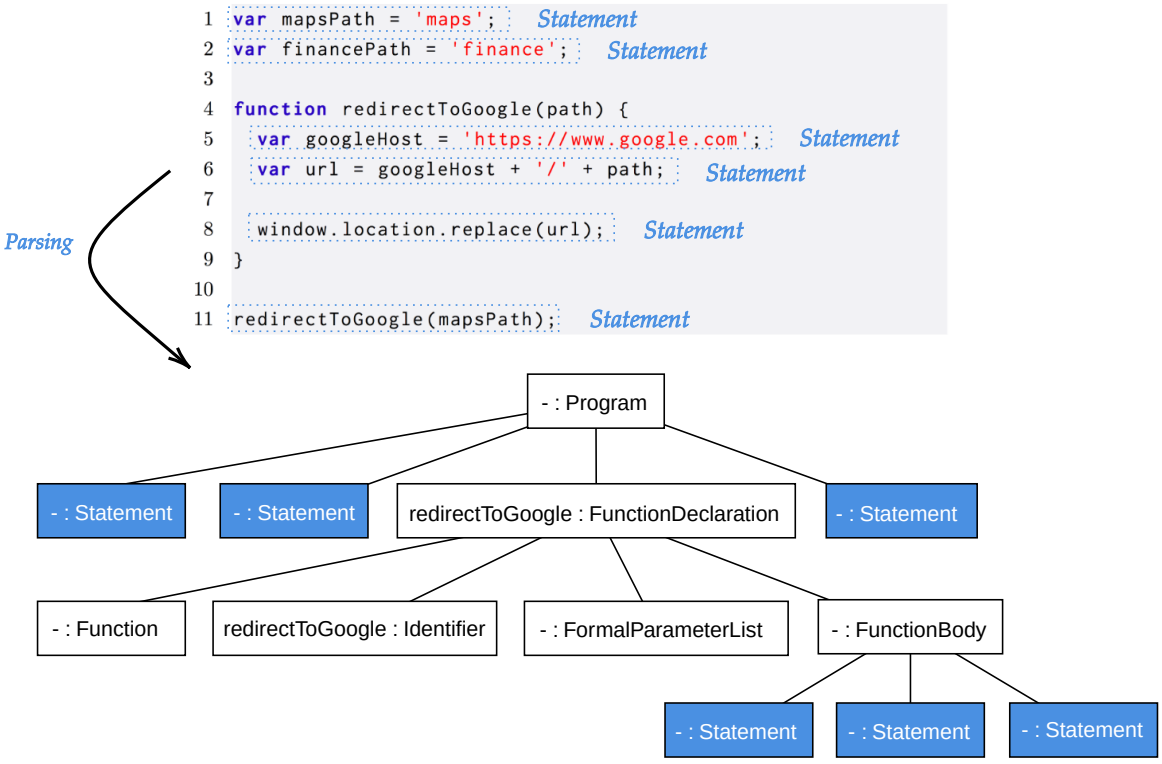
```
...
@FSTNonTerminal()
Program: ( SourceElement )* <EOF>;

@FSTNonTerminal()
SourceElement:
    StatementList
    |
    @FSTNonTerminal(name="{Identifier}")
    FunctionDeclaration;
...
@FSTTerminal(merge="LineBased")
StatementList: ( LOOK_AHEAD(2) Statement )+;
...
```

Figure 10 presents two program structure trees that are produced from the same program, but using parsers generated by different grammars. In Figure 10(a), we have the tree generated by the grammar that was written for `jsFSTMerge v0`, with statements—highlighted in blue— represented as nodes. In Figure 10(b), we can see that the statements that appear in sequence, both at the top level of the program and within the function body, are wrapped into a `StatementList` node. It is important to note that for each node, we have *name : type* as a label. For nodes whose type occur only once across children of a given parent, there is no problem in making use of the default name ("-"), since it will be the only option for matching. Issues, as mentioned before, come up when having multiple children with the same default name, as it happens to `StatementList`. Besides naming, another recurrent problem is that those statement lists are also sensitive to order. Permuting statement lists, among function declarations, may change program semantics due to side effects. This means that we need to preserve the order of statement lists when performing semistructured merge to not change the behaviour of revisions.

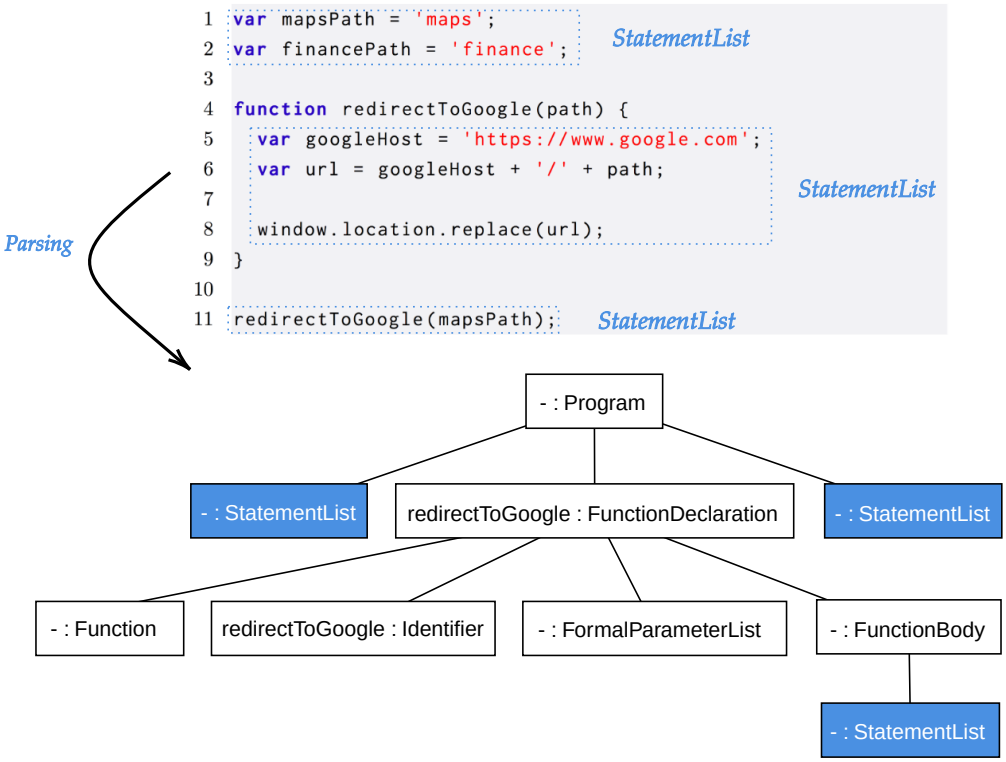
The modification of the `FeatureBNF` grammar to group together consecutive statements in a terminal node solves some of the issues we discussed before, e.g., the highly fine granularity for having one node for each statement. However, there is still a need of having, during superimposition, proper matching of statement lists among revisions, respecting the order that they appear in a program.

A first effort to ensure statement lists are properly handled in the superimposition was changing the merge implementation from `FSTMerge` 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 counts for each parent node. Before performing

Figure 10 – program structure trees generated by different versions of grammar



(a) Program structure tree from grammar for jsFSTMerge v0



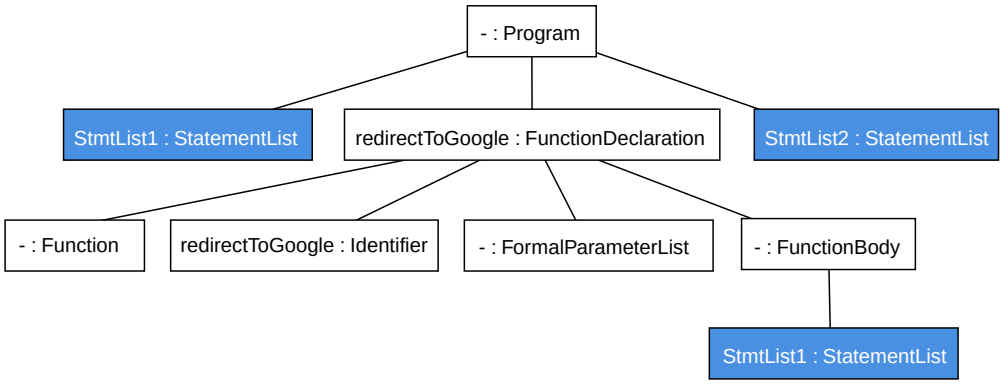
(b) Program structure tree from grammar for jsFSTMerge v1/v2

the superimposition on three revisions (base, left, and right), we take each revision and execute a recursive operation that traverses their trees to assign names to `StatementList`

nodes based on a numeric count for a given common parent. For a parent node, its N children that are statement lists are named, respectively, *StmtList1*, *StmtList2*, ..., *StmtListN*. And the process proceeds recursively, performing the same operation on the other children that are non-terminal nodes (in this case, `FunctionDeclaration` ones).

Figure 11 illustrates how the name assignment for statement list works. We have the same program structure tree shown in Figure 10(b), in which all `StatementList` nodes have default names (*- : StatementList* as label), but now the statement lists have names according to their position, across nodes of the same type, within their parent. The result of applying that naming strategy is that now the superimposition, given two revisions and two matched nodes, say A and B, will compose the first statement list from A with the first one from B, the second statement list from A with the second one from B, and so on and so forth. In case there is no conflict, the composition of two statement lists generates a new node, in the merged revision, whose content is produced by the unstructured merge of the statement lists, once they are terminal nodes.

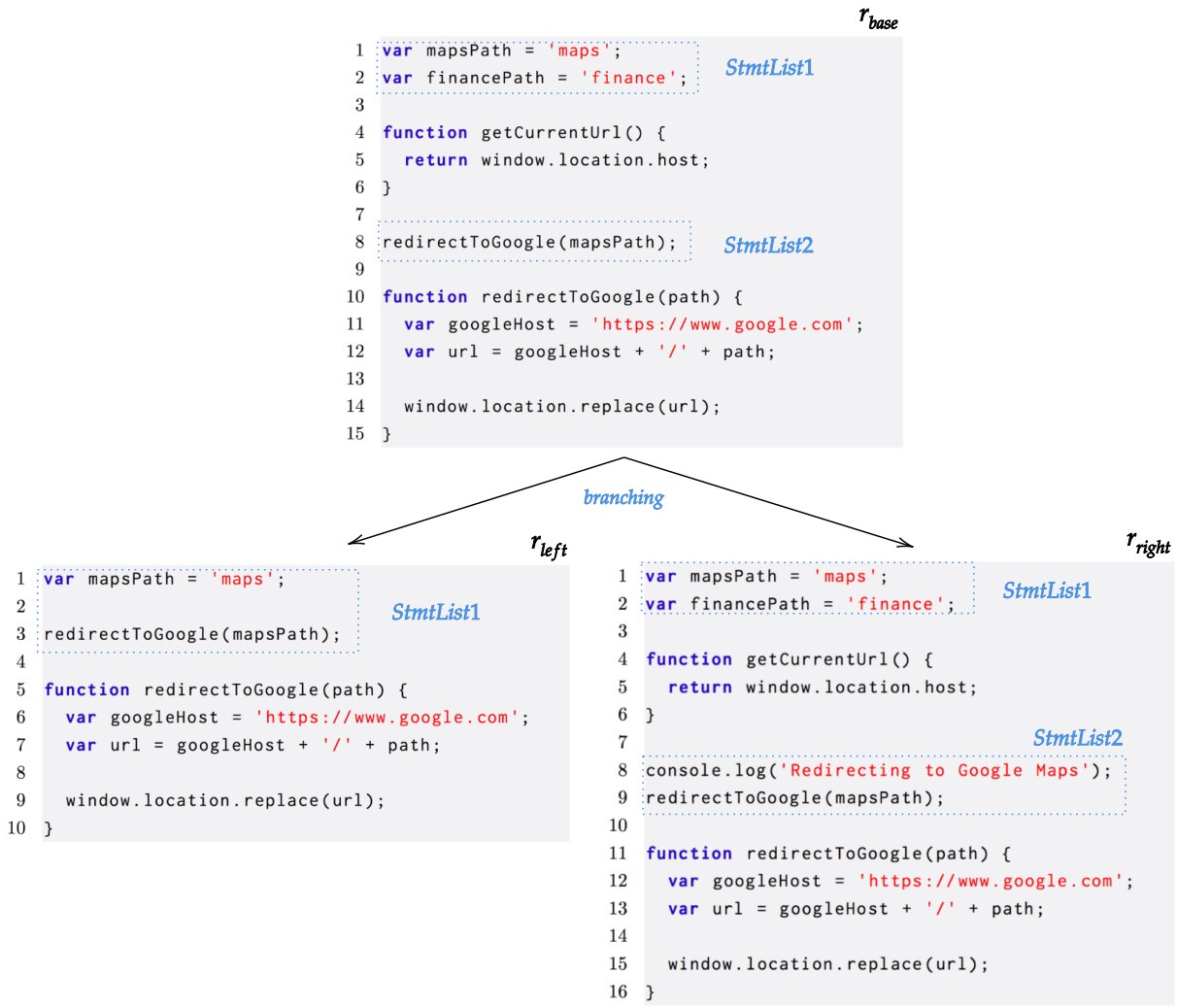
Figure 11 – program structure trees after assigning names to statement lists



With that change in the merge algorithm to name statement lists, we specified a new version of our tool: `jsFSTMerge v1`, which is the first one that works in a consistent manner, producing merged revisions which are often semantically correct. A drawback of that 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. Such conflicts are considered to be a case of false positive of semistructured merge when analyzing that version of the tool.

Figure 12 presents an example of merge scenario in which an one-to-one mapping between statement lists from a base revision no longer exists in one of the new revisions. In the base revision, r_{base}, there are two statement lists at the top level of the program. Then, r_{left} removes the unused variable `financePath` and function `redirectToGoogle`. The deletion of that function makes statements, which were in different statement lists in r_{base}, to be in a sequence, composing a single `StatementList` node. On the other hand, r_{right} keeps the number of statement lists at the top level of the program, just changing

Figure 12 – Merge scenario with false positive added by `jsFSTMerge v1`



the content of `StmtList2`. As a result, when merging r_{left} and r_{right}, superimposition does not understand that the statement from `StmtList2` in base revision (line 8) is now, in r_{left}, within `StmtList1`. The algorithm simply interprets that r_{left} removed `StmtList2`, while r_{right} edited it, which causes `jsFSTMerge v1` to report a conflict:

```
1  var mapsPath = 'maps';
2
3  redirectToGoogle(mapsPath);
4
5  <<<<<<< LEFT
6  =======
7  console.log('Redirecting to Google Maps');
8  redirectToGoogle(mapsPath);
9  >>>>>>> RIGHT
10
11 function redirectToGoogle(path) {
12   ...
13 }
```

In turn, an unstructured merge tool is able to merge r_{left} and r_{right} without reporting any conflicts, and producing a valid program:

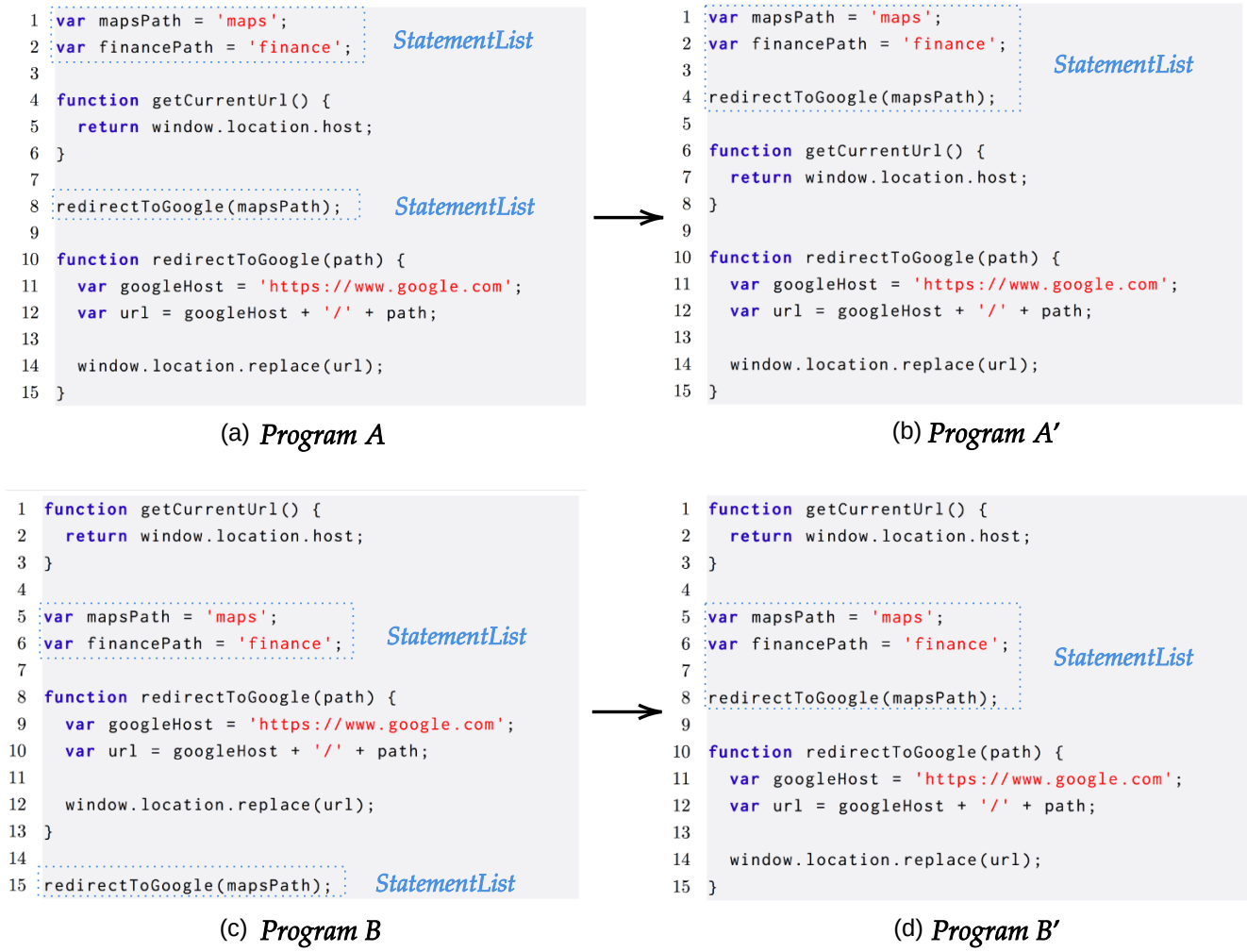
```
1  var mapsPath = 'maps';
2
3  console.log('Redirecting to Google Maps');
4  redirectToGoogle(mapsPath);
5
6  function redirectToGoogle(path) {
7    var googleHost = 'https://www.google.com';
8    var url = googleHost + '/' + path;
9
10   window.location.replace(url);
11 }
```

3.1.3 jsFSTMerge v2: Annotating an Adapted Grammar and Joining Nodes

In order to avoid that type of false positive in the semistructured merge, we created a new version of the tool, denominated `jsFSTMerge v2`, that handles matching and order maintenance of statement lists by using a different approach, in place of naming based on position. This version, as in `jsFSTMerge v1`, uses the parser generated from the adapted grammar that groups consecutive statements into statement lists (Listing 3.2). The relevance in working on a new version of a semistructured merge tool solely to avoid those false positives depends on how often they happen. This is discussed in the next chapter (Chapter 4), that presents a study in which both `jsFSTMerge v1` and `jsFSTMerge v2` are evaluated, and analyzes the frequency of this type of false positive in real-world software projects.

The strategy used by `jsFSTMerge v2` to deal with statement lists is also based on a recursive operation performed on revision trees prior to superimposition. This operation takes, for a given parent node, sibling statement list nodes and joins them into a single `StatementList` node, keeping their order. The idea behind that strategy comes from the observation that, as long as we keep the statements in order, it does not matter the order of them with respect to function declarations, because of their hoisting behaviour. This allows us to freely rearrange group of statements in relation to function declarations without changing the semantics of the program. Figure 13 illustrates the transformation of programs that is applied on revisions as part of the merge process when using `jsFSTMerge v2`. The program A' is obtained from A by joining its statement lists into a single one. Likewise, The program B is transformed into B'. The joining operation is able to rearrange statements to make them appear in a sequence while preserving the behaviour of the program. In fact, all four programs (A, A', B, and B') are semantically equivalent.

Figure 13 – Transformation of programs by joining statement lists



(a) *Program A*

(b) *Program A′*

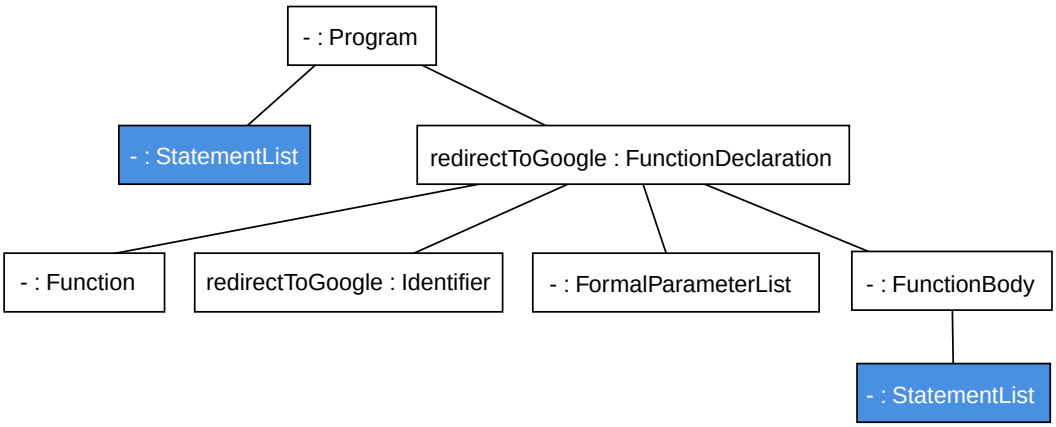
(c) *Program B*

(d) *Program B′*

The joining of statement lists is an operation that is executed on program structure trees which are produced by the generated parser. So, basically, what happens is a transformation of program structure trees as a pre-processing step for the superimposition. For example, Figure 14 shows the tree that is obtained from the one presented in Figure 10(b). Now there is, at most, one `StatementList` node among child ones, which eliminates issues with matching of statements that continue to be direct children of the same parent across revisions. Nonetheless, there are cases in which statements are moved into a function declaration, as discussed later in this section, that might cause false positives due to the inability of semistructured merge to match statements when they are not kept at the same structural level in the tree.

It is worth noting that, by joining statement lists, the property of having an arbitrary order that any node (either generated by `@FSTNonTerminal` or `@FSTTerminal`) in a program structure tree should respect becomes valid, since function declarations and a single statement list can be safely permuted, preserving the behaviour of the program. Moreover, it is no longer necessary to assign names to statement lists. The matching by

Figure 14 – program structure trees after joining statement lists



name— using the default "-"— and type works properly, because there is never more than one `StatementList` node as children.

A drawback of `jsFSTMerge v2` is that it potentially changes the format of the code by rearranging groups of statements, differently from `jsFSTMerge v1`, which keeps original order of elements. As we can see in Figure 13, 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 introduced by `jsFSTMerge v2` is discussed in Chapter 4.

3.2 COMPARING MERGE APPROACHES

To compare unstructured merge implemented by traditional tools with semistructured merge implemented by `jsFSTMerge v1` and `jsFSTMerge v2`, a relevant metric concerns how often each merge tool is able to detect interference between development tasks, so that it reports interfering changes as conflicts, and automatically integrates non-interfering ones (HORWITZ; PRINS; REPS, 1989; PERRY; SIY; VOTTA, 2001). A change introduced by a developer is considered to be non-interfering with respect to a second developer when that change has no effect on what the latter expects (GOGUEN; MESEGUER, 1982). In that context, merge tools might report conflicts that do not represent interference between development tasks, i.e., spurious conflicts (*false positives*), and they might also miss actual interference between revisions (*false negatives*) (CAVALCANTI; ACCIOLY; BORBA, 2017).

For a developer, it is important for a merge tool to not only minimize false positives, reducing unnecessary effort in resolving spurious conflicts, but also to minimize false negatives, avoiding the possibility of missing conflicts that could appear in other integration phases, such as building and testing, or even as errors during runtime. The challenge is establishing ground truth for interference between developers' changes, in particular, false positives and false negatives. Determining interference, on the contrary, is not computable (BERZINS, 1986), even though it could be possible with the assistance of experts who have knowledge about the involved artifacts, while still being error-prone. Instead, Cavalcanti,

Accioly and Borba (2017) recommend to relatively compare the merge approaches in terms of *added* occurrence of false positives and false negatives. To compare the approaches, we analyze when they report different results for the same three-way merge scenario.

With the purpose of guiding our relative comparison, we manually identified, considering programs written in JavaScript, cases of spurious conflicts reported by one approach, but not by the other (false positives), and cases of actual interference reported as conflicts by one approach, but missed by the other (false negatives). As semistructured merge tool, we take both `jsFSTMerge v1` and `jsFSTMerge v2` into account[3], but not `jsFSTMerge v0`, which is disregarded in this analysis of false positives and negatives. As unstructured merge tool, in the same manner as Cavalcanti, Accioly and Borba (2017) proceeded in their study, we use `KDiff3`, one of several unstructured merge tools available, which is a representative implementation of the *diff3* algorithm (KDIFF3, 2018). For both approaches, we empirically assess a sample of JavaScript scenarios to analyze where the tools behave differently, and how such differences can lead to false positives and false negatives. For each conflict reported by one of the tools, we checked if it was reported by the other tool as well. In case the conflict is reported by only one of the tools, we assess whether it is a false positive or a false negative, ignoring cases in which conflicts reported by both tools are, basically, the same one, but with slight textual differences.

In the following subsections, we describe the observed types of added false positives and false negatives for each merge tool. It is important to mention that, as our objective is to relatively compare the unstructured approach with the semistructured one, we disregard common cases of false positives and negatives across both approaches. The approaches may behave identically, for instance, when superimposition composes two `StatementList` nodes, launching unstructured merge. Also, when a semistructured merge tool is used with revisions which contain software artifacts not written in JavaScript— it is fairly common to have HTML and CSS files in projects that primarily use JavaScript—, unstructured merge is called, making both approaches provide common false positives and negatives.

3.2.1 False Positives Added by Unstructured Merge

As discussed in Subsection 2.2.1, a shortcoming of unstructured merge is its inability to identify commutative and associative elements. Generally for JavaScript, the latter include only function declarations, due to their hosting behaviour, but, in the context of our semistructured merge tool, statement lists can also be rearranged with respect to function declarations; particularly for `jsFSTMerge v2`, a joined statement list node can be freely permuted among its node siblings. All this is ignored by unstructured tools that rely on textual analysis, leading to ordering conflicts, which are the only kind of false positives added by unstructured merge.

[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.

Figure 15 – Merge scenario with false positive added by unstructured merge (ordering

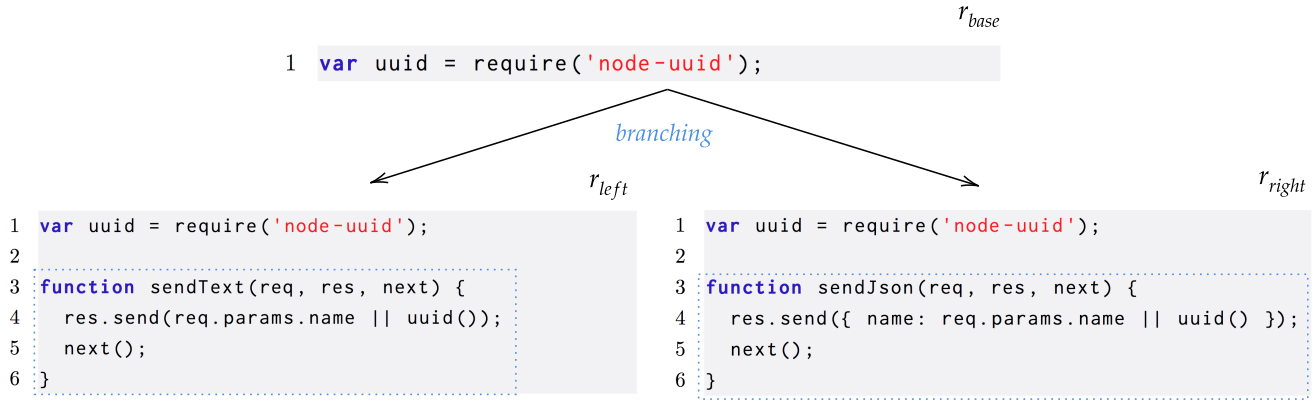


Figure 15 shows a merge scenario, adapted from a real case in the *node-restify* project[4], in which an ordering conflict can be observed. An unstructured merge tool reports a conflict, because two developers added two different functions (`sendText` and `sendJson`) to the same text area:

```
1  var uuid = require('node-uuid');
2
3  <<<<<<< LEFT
4  function sendText(req, res, next) {
5    res.send(req.params.name || uuid());
6  =======
7  function sendJson(req, res, next) {
8    res.send({ name: req.params.name || uuid() });
9  >>>>>>> RIGHT
10   next();
11 }
```

Differently, semistructured merge identifies that each function is a distinct element, and can produce the desired output without any conflicts, only by superimposing program structure trees:

```
1  var uuid = require('node-uuid');
2
3  function sendText(req, res, next) {
4    res.send(req.params.name || uuid());
5    next();
6  }
7
8  function sendJson(req, res, next) {
9    res.send({ name: req.params.name || uuid() });
10   next();
11 }
```

[4] https://github.com/restify/node-restify

Ordering conflicts, from unstructured merge, do not always happen between function declarations. It can also happen between a statement list and a function declaration. For instance, when a developer adds a function declaration to the same text area where another developers adds one or more statements, unstructured merge reports a conflict, while semistructured merge is able to automatically merge the revisions. Ordering conflicts were also the only false positives added by unstructured merge tools, when considering Java systems, identified by Cavalcanti, Accioly and Borba (2017) in their study. In the context of Java, classes and methods are the declarations most often involved in ordering conflicts.

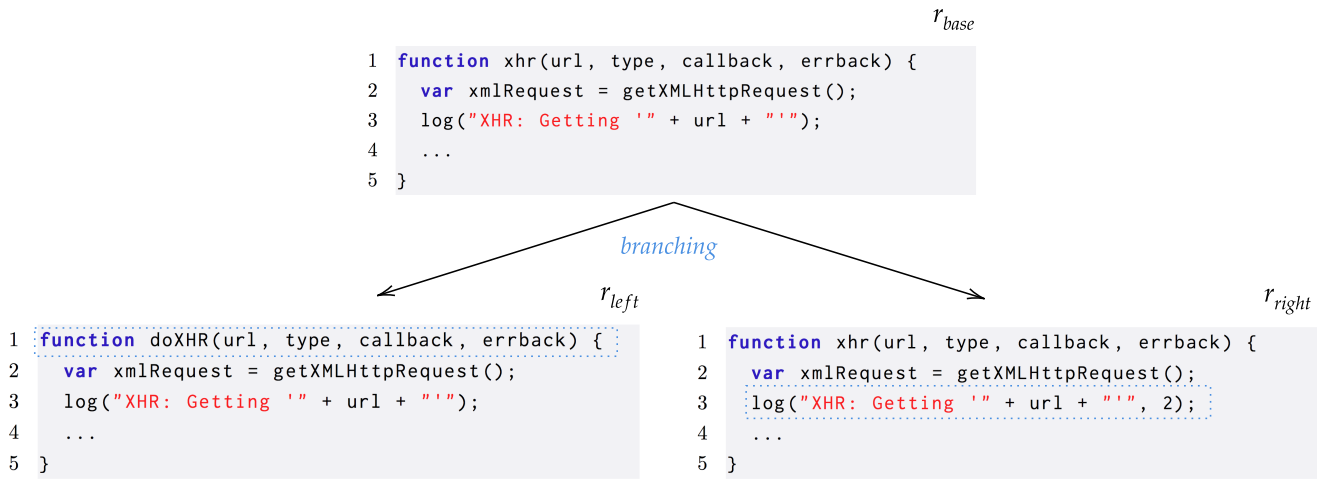
3.2.2 False Positives Added by Semistructured Merge

The subsections that follow describe cases in which semistructured merge tool adds false positives when compared to the textual, line-based unstructured approach.

3.2.2.1 Function Renaming Conflict

As previously noted by Apel et al. (2011) and Cavalcanti, Accioly and Borba (2017), renaming is a challenge for semistructured merge. When an element is renamed in one revision, the semistructured merge algorithm is not aware of this renaming, and it cannot map the renamed element to its version from the base revision. Figure 16 illustrates a merge scenario, based on a case from *Less.js*[5], where this happens. A developer (revision r_{left}) renames a function from `xhr` to `doXHR`, while a second developer (revision r_{right}) edits its body.

Figure 16 – Merge scenario with false positive added by semistructured merge (function renaming conflict)



[5] https://github.com/less/less.js/

Unstructured merge does not report conflicts, because the changes introduced by revisions r_{left} and r_{right} occur in distinct text areas, having a statement as separator of chunks (line 2), and it generates a program that combines contributions from both developers, with a new name and a new body:

```
1  function doXHR(url, type, callback, errback) {
2    var xmlRequest = getXMLHttpRequest();
3    log("XHR: Getting '" + url + "'", 2);
4    ...
5  }
```

Semistructured merge, on the other hand, does not understand that r_{left} renamed `xhr`. Instead, the algorithm interprets the function renaming as a deletion, and assumes that r_{left} deleted a function modified by r_{right}, which supposedly represents an interference between development tasks. Therefore, semistructured merge reports a conflict:

```
1  function xhr(url, type, callback, errback) {
2  <<<<<<< LEFT
3  =======
4    var xmlRequest = getXMLHttpRequest();
5    log("XHR: Getting '" + url + "'", 2);
6    ...
7  >>>>>>> RIGHT
8  }
9
10 function doXHR(url, type, callback, errback) {
11   var xmlRequest = getXMLHttpRequest();
12   log("XHR: Getting '" + url + "'");
13   ...
14 }
```

It is interesting to observe that the renamed function (`doXHR`) is not surrounded by conflict markers, because it is actually just seen, by semistructured merge, as a newly added function. The merge algorithm's assumption is that, instead of renaming a function, r_{left} deleted `xhr` and added `doXHR`. Barros (2018) discusses solutions for a semistructured merge tool to deal with renaming conflicts.

Renaming conflicts are often semistructured merge false positives, but they might indeed represent actual interference between development tasks. For instance, consider a scenario where a developer renames a function, and the other one not only edits its body, but also adds new calls to it. Unstructured merge leads to an invalid program, with calls to an undefined function, whereas semistructured merge soundly does not perform the merge, and reports a conflict.

3.2.2.2 Function Conversion Conflict

Another case of false positive added by semistructured merge comes from the conversion of a function expression assigned to a variable into a function declaration, and vice-versa. Subsection 2.3.2 discussed the difference between those two means of creating a function. False positives might occur when one developer converts a function from a form to the other, and another developer changes that function. This type of conflict is referred in this work as a *function conversion conflict*. In Figure 17, there is an example of merge scenario, extracted from a project called *BitcoinJS*[6], that shows such case. The revision r_{left} converts the variable `processTx`, that holds a function expression, into a function declaration, whereas r_{right} modifies the function expression body.

Once again, unstructured merge does not report conflicts due to the fact that the changes occur in different text areas, having a variable assignment as a separator (line 2), and it produces a valid merge:

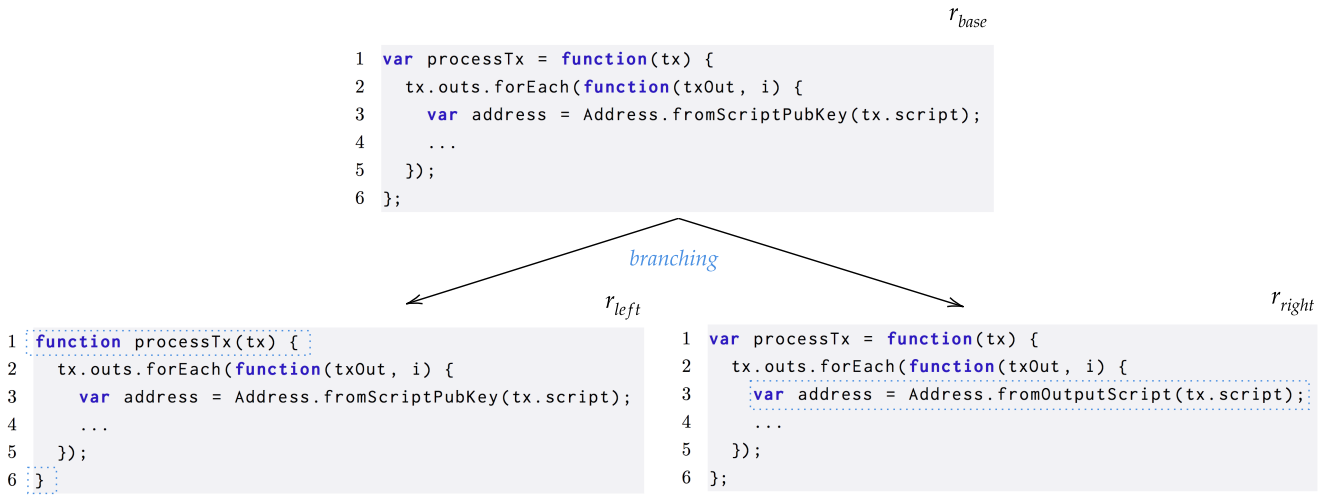
```
1 function processTx(tx) {
2   tx.outs.forEach(function(txOut, i) {
3     var address = Address.fromOutputScript(tx.script);
4     ...
5   });
6 }
```

Semistructured merge, however, reports a conflict for a similar reason as in function renaming described earlier. The merge algorithm— relying on unstructured merge to compose the `StatementList` nodes that contain the function expression assigned to a variable as plain text— understands that textual content of a `StatementList` node was removed by r_{left}, while part of the textual content of the same node was edited by r_{right}:

```
1  <<<<<<< LEFT
2  =======
3  var processTx = function(tx) {
4    tx.outs.forEach(function(txOut, i) {
5      var address = Address.fromOutputScript(tx.script);
6      ...
7    });
8  };
9  >>>>>>> RIGHT
10
11 function processTx(tx) {
12   tx.outs.forEach(function(txOut, i) {
13     var address = Address.fromScriptPubKey(tx.script);
14     ...
15   });
16 }
```

[6] https://github.com/bitcoinjs/bitcoinjs-lib

Figure 17 – Merge scenario with false positive added by semistructured merge (function conversion conflict)



The other way around, having a developer migrating a function declaration into a function expression, whereas another developer changes its body, is also reported as a conflict by semistructured merge, but for a slightly different reason. Semistructured merge reports a conflict because it interprets such conversion as a deletion of function declaration, and assumes that a developer removed a function that was edited by the other one. Conversely, this case might also be a semistructured merge true positive, when the latter developer adds a reference to the function (e.g., in a call) before its declaration. If a function declaration were kept, the program would still work, since a function declaration gets hoisted, but having a function expression assigned to a variable leads to a program that throws an error during runtime. While semistructured tool would soundly not perform the merge and report a conflict, unstructured merge would erroneously merge the contributions, generating an invalid program. An example of such case is provided, as a false negative for unstructured merge, in the next subsection.

3.2.2.3 Function Declaration Displacement Conflict

An additional case of false positive introduced by semistructured merge might arise when a developer moves a function declaration into a sibling node (e.g., a `StatementList`), which causes the semistructured merge algorithm to interpret such change as a deletion of the moved function. A so-called *function declaration displacement conflict* happens when a function declaration is moved in that way by one developer, while it is modified by another one. Usually, such conflict takes place when a developer moves a function declaration into an *Immediately Invoked Function Expression* (IIFE) with the purpose of not polluting the global scope.

Figure 18 – Merge scenario with false positive added by semistructured merge (function declaration displacement conflict)

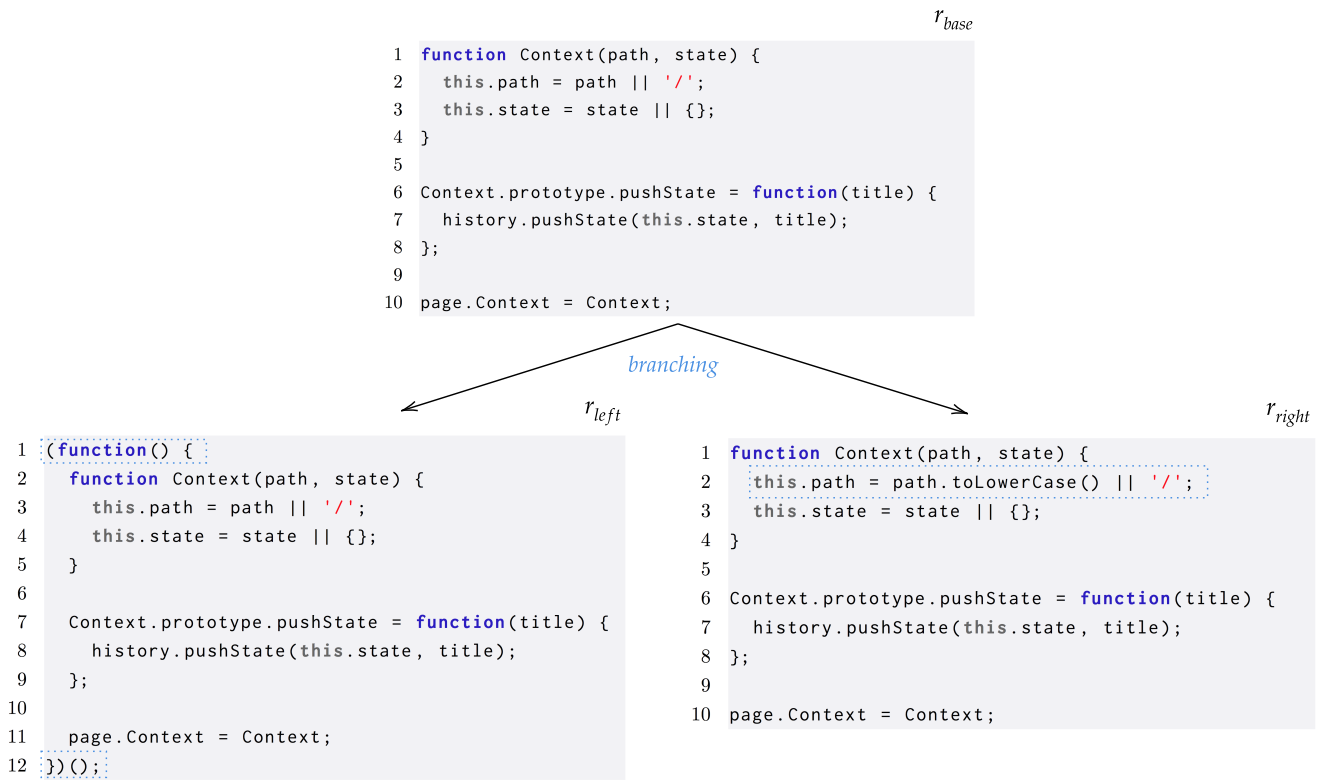


Figure 18 presents an example of this, based on a real case from the *page.js* project.[7] The revision r_{left} moves the function `Context`, which is emulating a class by means of prototype (SILVA et al., 2017), along with statements, into an IIFE. And, in parallel, r_{right} edits the body of `Context`. As in previous scenarios, unstructured merge performs the merge without reporting a conflict, once the changes are made to different text areas (whitespace changes are ignored in the present analysis):

```
(function() {
  function Context(path, state) {
    this.path = path.toLowerCase() || '/';
    this.state = state || {};
  }

  Context.prototype.pushState = function(title){
    history.pushState(this.state, title);
  };

  page.Context = Context;
})();
```

[7] https://github.com/visionmedia/page.js

With our semistructured merge tool, we have a recurring behaviour. When moving `Context`, and further statements, into an anonymous function expression (contained in a `StatementList` node), the function declaration is no longer a node represented in the program structure tree, being just part of textual content of a statement list node. As a consequence, the algorithm understands that r_{left} removes `Context`, whereas r_{right} edits it, thus raising a conflict:

```
function Context(path, state) {
<<<<<<< LEFT
=======
  this.path = path.toLowerCase() || '/';
  this.state = state || {};
>>>>>>> RIGHT
}

(function() {
  function Context(path, state) {
    this.path = path || '/';
    this.state = state || {};
  }

  Context.prototype.pushState = function(title){
    history.pushState(this.state, title);
  };

  page.Context = Context;
})();
```

The other way around, having a developer moving a function declaration from a statement list out of it (e.g., when eliminating the usage of an IIFE), while another developer edits part of the textual content of the statement list node that refers to the function body, also is reported as a conflict by semistructured merge. When moving a function declaration out of a statement list, there will be a new non-terminal node for that function, and the statement list node's textual content will be changed to remove the function declaration. So, when that happens followed by an edition of other developer on the same function (i.e., the same area in the statement list node's textual content), there is a textual conflict in the statement list node.

3.2.2.4 No Longer Existing One-to-one Mapping Conflict

All the aforementioned cases of semistructured merge false positives are added by both `jsFSTMerge v1` and `jsFSTMerge v2`. However, as described and exemplified in the Subsection 3.1.2, there is an extra type of false positive that is exclusively added by `jsFSTMerge v1`, which occurs when one-to-one mappings between `StatementList` nodes are not kept.

3.2.3 False Negatives Added by Unstructured Merge

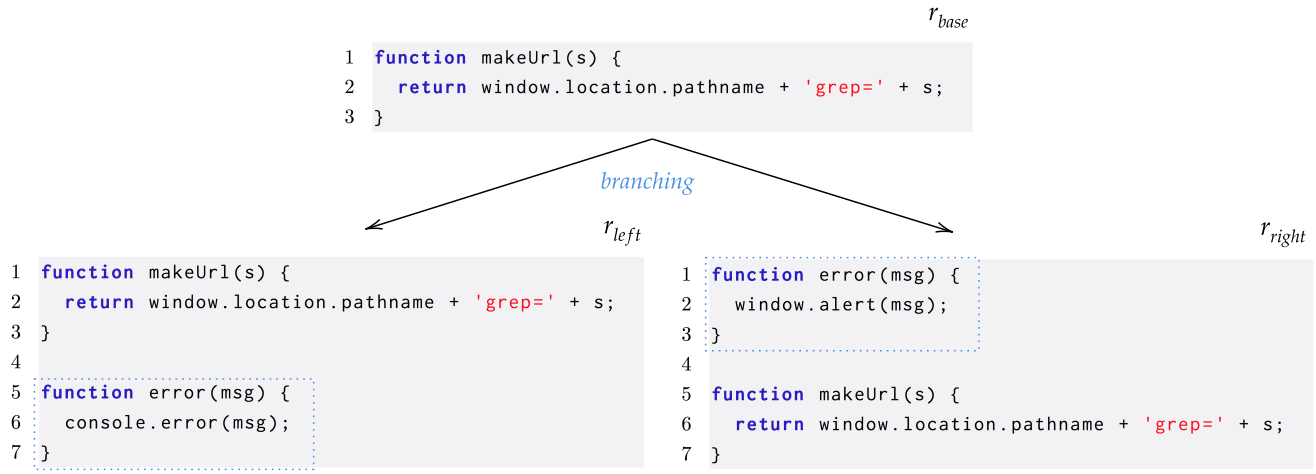
The subsections that follow present scenarios where unstructured merge tools miss actual interference between development tasks, differently from semistructured merge.

3.2.3.1 Adding Duplicated Function Declaration

Similarly to the findings obtained by Cavalcanti, Accioly and Borba (2017) for Java systems, the false negatives added by unstructured merge for JavaScript are mostly caused by not detecting when revisions add duplicated function declarations. As long as function declarations with a common name are added to different areas of the program, an unstructured merge tool reports no conflict. In JavaScript, there is no function overloading; when there are two function declarations with the same name within the same scope, even if they have different formal parameters, the last function defined will override the previously defined one (STEFANOV, 2010). And it does not cause a compilation error, differently from languages such as Java, that does not allow two class methods with same signature. When comparing the impact of duplicated declaration for both JavaScript and Java systems, we observe that not reporting conflicts for JavaScript artifacts potentially cause more trouble, since duplicated function declarations are semantic errors that typically do not raise any warning during a build phase, thus being harder to be detected.

Figure 19 shows an example of merge scenario, inspired on the *Mocha*[8] codebase, where two revisions (r_{left} and r_{right}) add functions with same name (`error`) to different text areas.

Figure 19 – Merge scenario with false negative added by unstructured merge (duplicated function declaration)



Unstructured merge does not raise a conflict, and it generates a program with two functions called `error`. This program does not cause a build error, but it represents

[8] https://github.com/mochajs/mocha

an interference between development tasks, once the developer who introduced the first function will rely on its implementation, and not the overriding one (line 9):

```
1  function error(msg) {
2    window.alert(msg);
3  }
4
5  function makeUrl(s) {
6    return window.location.pathname + 'grep=' + s;
7  }
8
9  function error(msg) { // Overrides function "error"
10   console.error(msg);
11 }
```

In turn, semistructured merge is able to match the functions by name and type via superimposition. Then, it tries to merge their bodies, and correctly identifies a conflict:

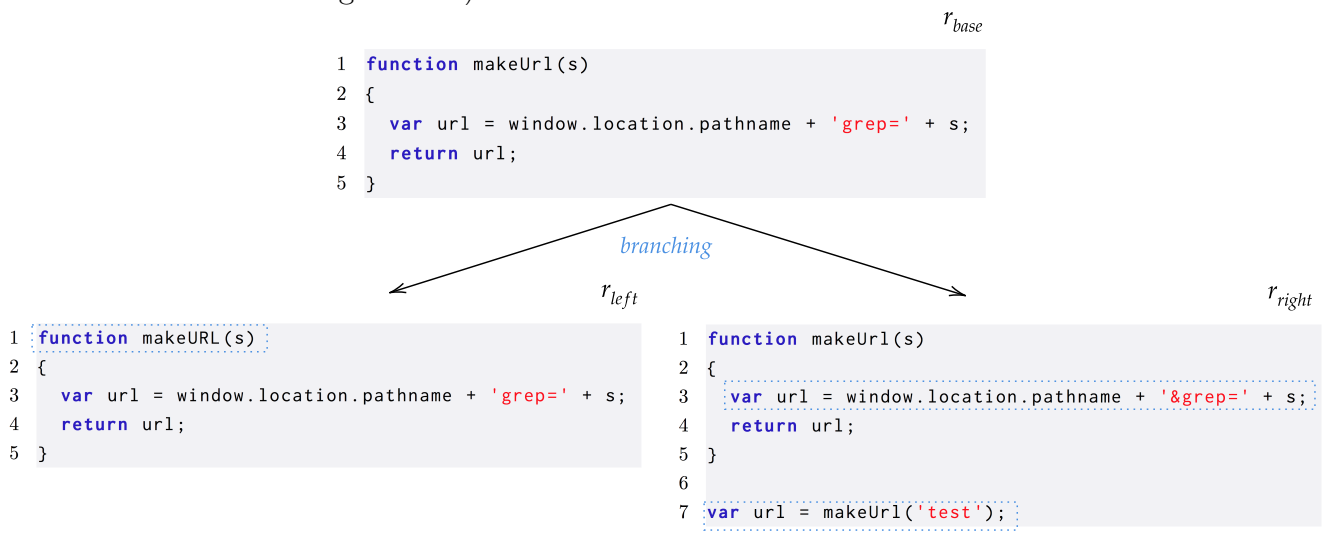
```
1  function error(msg) {
2  <<<<<<< LEFT
3    console.error(msg);
4  =======
5    window.alert(msg);
6  >>>>>>> RIGHT
7  }
8
9  function makeUrl(s) {
10   return window.location.pathname + 'grep=' + s;
11 }
```

3.2.3.2 Adding Call to Renamed Function

Function renaming, as previously explained, can also be a case of false negative added by unstructured merge. This happens when a developer renames a function, and another developer, in addition to changing its body, adds a call to it by referring to the old 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 an error during runtime. Once again, this case was also observed in Java systems (CAVALCANTI; ACCIOLY; BORBA, 2017), but trying to invoke a method that no longer exists, in Java, causes a compilation error, which makes this false negative easier to be identified than in JavaScript.

Figure 20 presents an example of such case, where revision r_{left} renames `makeUrl` to `makeURL`, and r_{right}, besides making changes to the function body, adds a statement with a call to the function using the old name (`makeUrl`).

Figure 20 – Merge scenario with false negative added by unstructured merge (function renaming conflict)



Since changes are made to different areas, having the opening curling brace ("{") as a separator, unstructured merge unsoundly does not report any conflict, and performs the merge. In the merged program, the call to the function causes a `ReferenceError` (line 7):

```
1 function makeURL(s)
2 {
3   var url = window.location.pathname + '&grep=' + s;
4   return url;
5 }
6
7 var url = makeUrl('test'); // ReferenceError: makeUrl is not defined
```

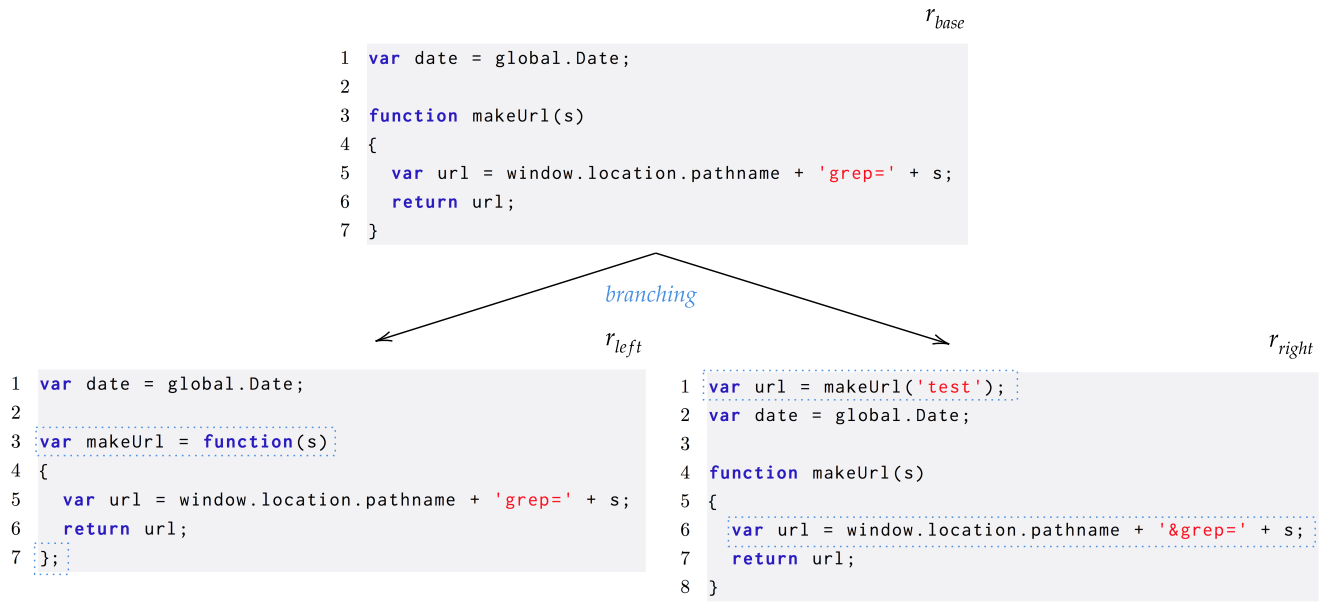
When using semistructured merge, the renaming of `makeUrl` is interpreted as a deletion of that function and the addition of a function called `makeURL`. At the same time, r_{right} edits `makeUrl` body, which causes semistructured merge to report a conflict. Therefore, semistructured merge ends up preventing an error, which would result from a call to a no longer existing function, from being escaped to users:

```
1  function makeUrl(s) {
2  <<<<<<< LEFT
3  =======
4    var url = window.location.pathname + '&grep=' + s;
5    return url;
6  >>>>>>> RIGHT
7  }
8
9  function makeURL(s) {
10   ...
11 }
12
13 var url = makeUrl('test');
```

3.2.3.3 Adding Early Call to No Longer Hoisted Function

Converting a function declaration into a function expression assigned to a variable can also be a false negative added by unstructured merge. As previously noted, 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. Unless new references occur in the same text area, unstructured merge incorrectly merges the contributions, yielding a program that throws errors during runtime. In Figure 21, there is an example of a scenario that illustrates this case. The revision r_{left} converts `makeUrl` into a function expression assigned to a variable, whereas r_{right} modifies the function body, and, also, adds a call to `makeUrl` at the beginning of the program.

Figure 21 – Merge scenario with false negative added by unstructured merge (function conversion conflict)



Unstructured merge does not raise conflicts, because changes are made in different text chunks, but the generated program leads to a `TypeError` when trying to invoke `makeUrl`:

```
1 var url = makeUrl('test'); // TypeError: makeUrl is not a function
2 var date = global.Date;
3
4 var makeUrl = function(s)
5 {
6   var url = window.location.pathname + '&grep=' + s;
7   return url;
8 };
```

Once more, semistructured merge avoids that runtime error by reporting a conflict, due to the fact that the algorithm understands that r_{left} removed a function that was edited by r_{right}:

```
...
var makeUrl = function(s)
{
  ...
};

function makeUrl(s) {
<<<<<<< LEFT
=======
  var url = window.location.pathname + '&grep=' + s;
  return url;
>>>>>>> RIGHT
}
```

3.2.4 False Negatives Added by Semistructured Merge

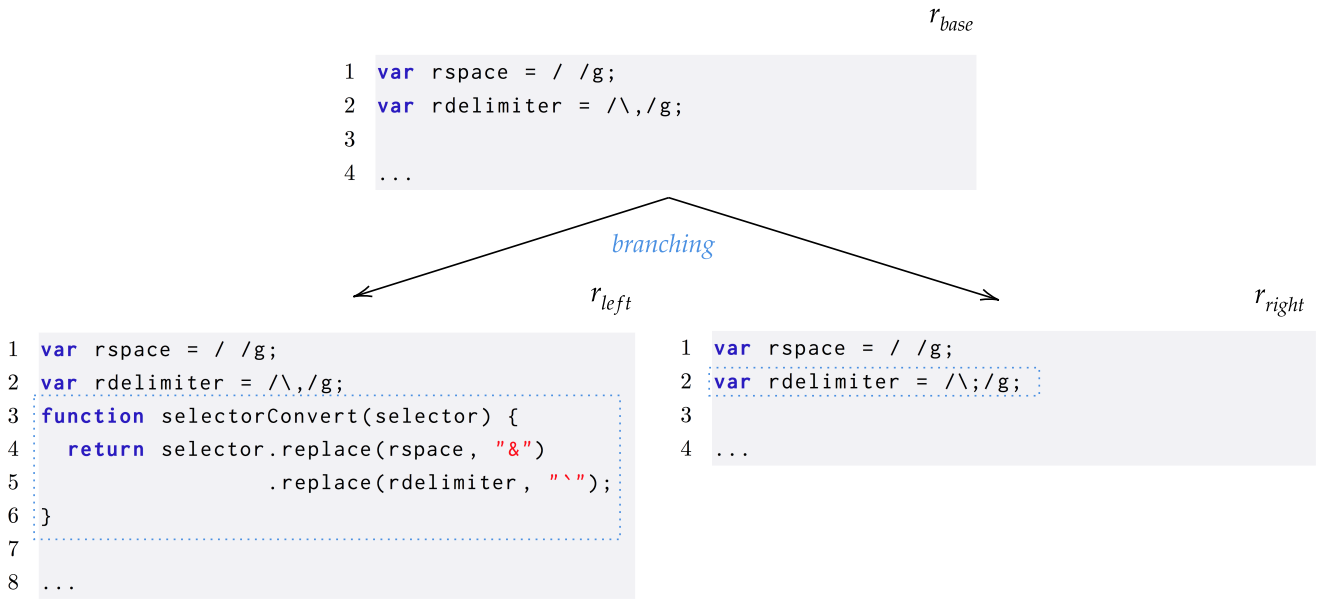
We were not able to find general cases of false negatives added by semistructured merge that conform to a set of recurring syntactic patterns. All the cases are from unstructured merge *accidentally* detecting semantic conflicts that would otherwise not be reported if changes were made in different text areas. Usually, this happens when a developer adds a new element that references an existing one, and, in parallel, another 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 behavioral errors; especially when the second developer is not simply refactoring or optimizing code. Since changes are made to different elements (i.e., different nodes in the generated program structure tree), no conflict is reported by semistructured merge. An example of merge scenario, adapted from a real case observed in the *jQuery* project[9], is shown in Figure 22. In this scenario, the r_{left} adds the function `selectorConvert`, which uses the existing variable `rdelimiter`, while r_{right} changed the value of this variable.

This time, there is no textual separator between changes introduced by the revisions; they are made in consecutive lines. As a result, unstructured merge reports a conflict:

```
var rspace = / /g;
<<<<<<< LEFT
var rdelimiter = /\,/g;
function selectorConvert(selector) {
  return selector.replace(rspace, "&")
                 .replace(rdelimiter, "`");
}
=======
var rdelimiter = /\;/g;
>>>>>>> RIGHT
...
```

[9] https://github.com/jquery/jquery

Figure 22 – Merge scenario with false negative added by semistructured merge (accidental conflict)



On the other hand, from semistructured merge's perspective, the changes are made in different nodes in the program structure tree (r_{left} adding a `FunctionDeclaration` node and r_{right} editing a `StatementList` node), so it does not report any conflict, but the merged revision might lead to a semantic error instead. In the generated program, the function `selectorConvert` now uses a value for `rdelimiter` that is different from what the first developer expects (a semicolon instead of a comma), possibly affecting the program correctness:

```
1 var rspace = / /g;
2 var rdelimiter = /\;/g;
3 function selectorConvert(selector) {
4   return selector.replace(rspace, "&")
5                  .replace(rdelimiter, "`");
6 }
7
8 ...
```

4 EVALUATION

Motivated by the lack of prior investigation of semistructured merge for JavaScript, as well as by encouraging results obtained in previous studies for other languages, we proposed and implemented different versions of semistructured tools for JavaScript as described in the previous chapter. In the present chapter, we analytically evaluate the approach proposed by Apel et al. (2011) to implement a semistructured merge tool for a certain programming language by instantiating the `FSTMerge` architecture. Furthermore, we empirically evaluate our merge tools created on top of `FSTMerge` for JavaScript, comparing them to unstructured merge tools, which are still widely used by industry. In particular, we investigate the following main research questions:

- **RQ1:** *Is the FSTMerge semistructured approach generalizable for JavaScript?*
- **RQ2:** *Could semistructured merge for JavaScript be effective in practice?*

For answering **RQ1**, Section 4.1 discusses implementation decisions that were made, as described in the previous chapter, in order to conceive a well-functioning tool for JavaScript based on the `FSTMerge` infrastructure. In this context, we investigate the generalizability of the `FSTMerge` approach to support an additional programming language by providing annotated grammars, comparing JavaScript to other languages. Moreover, Section 4.1 indicates improvements that can be made to the `FSTMerge` framework to make it more generalizable for JavaScript and other languages that have similar characteristics.

To answer **RQ2**, in turn, Section 4.2 presents an empirical study— based on the research conducted by Cavalcanti, Accioly and Borba (2017)— that assesses whether semistructured merge approach for JavaScript, considering `jsFSTMerge v1` (Subsection 3.1.2) and `jsFSTMerge v2` (Subsection 3.1.3) tools, reduces integration effort (productivity), without negatively impacting the correctness of the merging process (quality), when compared to unstructured merge. This evaluation is performed by reproducing merges from the development history of different GitHub projects that primarily use JavaScript, and analyzing evidences of incidence of false positives and false negatives described in the previous chapter. Although this empirical study is based on a prior research, it is not a replication study in a strict sense, as different tools, code samples, and methods to compute metrics are employed, as explained later on this text.

4.1 IS THE FSTMERGE SEMISTRUCTURED APPROACH GENERALIZABLE FOR JAVASCRIPT?

As described in Section 3.1, the `FSTMerge` architecture consists of two main parts (APEL et al., 2011): 1) a generic merge engine that is capable of identifying and resolving cer-

tain conflicts and 2) an abstract specification, for each supported language, of program elements whose order does not matter and of how they should be merged. This abstract specification is given by an annotated grammar in the `FeatureBNF` format. The assumption made by Apel et al. (2011), upon instantiating `FSTMerge` with a few programming languages, such as Java and C#, is that the semistructured approach implemented by this architecture is generic enough to be applied for other languages. To support a new language, Apel et al. (2011) explain that it is only necessary to annotate an off-the-shelf grammar, based on some publicly available grammar for the language. Additionally, special conflict handlers can be implemented for providing further merging logic for terminal nodes. `FSTMerge` merge engine is supposed to be sufficiently generic, thus not requiring any modification, to provide support for semistructured merge for any language, depending solely on a grammar enriched with information for conflict resolution.

On the contrary of the procedure Apel et al. (2011) carried out to instantiate `FSTMerge` for Java and C#, simply providing an annotated off-the-shelf grammar, we had to introduce adaptations to both grammar and `FSTMerge` merge engine to have an *effective* semistructured merge tool for JavaScript:

- **Modification of grammar**. The implementation of `jsFSTMerge v0` used a parser generated from an annotated grammar for JavaScript written according to the specification of ES5 (ECMA, 2015). Using this off-the-shelf grammar, however, led to the problems discussed in Subsection 3.1.1, which mainly concern the definition of individual statements at the same level as function declarations. A solution to solve some of these problems, including erroneous matching of single statements, was modifying the grammar to group consecutive statements into a new element: `StatementList`.

- **Modification of merge engine**. With a new version of the grammar that uses `StatementList` to group statements, there is no matching and ordering issues with superimposition when there is only one child node of its type (i.e., statements in a continuous sequence, without being separated by function declarations), because the matching, in this case, is trivial and permuting a single `StatementList` node with function declarations does not change program semantics. On the other hand, when there is more than one `StatementList` subnode, which is a far more often case, semistructured merge, from the original `FSTMerge` engine implementation, is not able to neither properly match statement lists nor guarantee that their order is preserved. To overcome those issues, changes were made to the merge engine to first assign identifiers (their position relative to other statement lists within a given scope) to statement lists (`jsFSTMerge v1`; see Subsection 3.1.2), and then to join statement lists node into a single one (`jsFSTMerge v2`; see Subsection 3.1.3).

Once it was not feasible to obtain a well-functioning semistructured merge tool for JavaScript without having to introduce changes to the ES5 grammar and the `FSTMerge`

merge engine, we claim that the `FSTMerge` semistructured approach is not effectively generalizable for JavaScript.[1] As a whole, `FSTMerge` semistructured approach can be considered generic in the sense that a tool for any programming language can be created by annotating its off-the-shelf grammar, but the resulting tool may not produce correct merges, not being an effective merge tool that could be used in practice.

It is noteworthy to remember that `FSTMerge` architecture allows conflict handlers to deal with particular cases of terminal node merging (e.g., merge of *implements* list in Java), which were not used in any version of `jsFSTMerge` explored in this thesis. Nevertheless, such handlers are not able to deal with issues concerning matching and ordering of statements, which were only solved by modifying input grammar and merge engine, so the lack of generalizability still persists regardless of the availability of conflict handlers.

In general, the main characteristics of JavaScript that prevent the `FSTMerge` semistructured approach, in its pure form (i.e., without requiring modifications of grammar or merge engine), from working in an effective manner are the following:

1. having a syntax that allows elements whose order matters (statements) at the same level as elements whose order is arbitrary (function declarations); and

2. not having unique names for elements whose order matters (statements).

These same characteristics, or slight variations of them, can be found in other programming languages, making `FSTMerge` semistructured approach not completely suitable for them as well, unless adaptations are made.

An example of programming language that shares such characteristics is PHP. In this language, as well as in JavaScript, statements, which cannot get a proper unique name from grammar annotation, appear at the same level as function declarations and, also, class declarations. Function (and class) declarations are commutative and associative, whereas statements are not, so the same matching and ordering problems would arise when implementing a semistructured merge tool for the language by following a pure approach to instantiate `FSTMerge`. Listing 4.1 shows an example of a program written in PHP, which has statements, a variable declaration (line 3) and call expressions (lines 9-10), at the same level as function declarations (lines 5-7, 12-14). As in JavaScript, functions, in PHP, can also be invoked before its definition, as we can see in line 10.

Python is another programming language that has similar features to JavaScript. The language syntax allows statements at the same level as function and class declarations. Listing 4.2 shows a program written in Python that is quite similar to the one shown in Listing 4.1, with variable declaration and function calls along with function declarations. However, differently from JavaScript and PHP, functions can be called only after its definition. When trying to invoke a not yet defined function in line 7, an error is raised. This makes function declarations (and the same goes for classes) not completely

[1] This claim applies to ES5, as well as newer versions of JavaScript, due to their backward compatibility.

commutative and associated with respect to statements. While function declarations can be safely permuted among themselves, they cannot be permuted with statements without possibly introducing runtime errors. Nonetheless, decisions made in this work could still be leveraged to have a semistructured merge tool for Python based on `FSTMerge` with adaptations on grammar and merge engine, as discussed later in this section.

Listing 4.1 – Example of program in PHP with statements and function declarations at the same level

```
1  $x = 'test';
2
3  function f1($val) {
4    echo 'f1: ' . $val;
5  }
6
7  f1($x); // "f1: test"
8  f2($x); // "f2: test"
9
10 function f2($val) {
11   echo 'f2: ' . $val;
12 }
```

Listing 4.2 – Example of program in Python with statements and function declarations at the same level

```
1  x = "test"
2
3  def f1(val):
4    print("f1: " + val)
5
6  f1(x) # "f1: test"
7  f2(x) # NameError: name 'f2' is not defined
8
9  def f2(val):
10   print("f2: " + val)
```

There are programming languages for which a semistructured merge tool based on a pure `FSTMerge` approach—simply with an off-the-shelf annotated grammar and possible conflict handlers— often produces semantically correct programs and has proven to present better results than unstructured merge; we have Java as an example (APEL et al., 2011; CAVALCANTI; ACCIOLY; BORBA, 2017). This happens, mainly, because those languages do not allow statements at the same level as commutative and associative declarations (in the case of Java, class and method declarations). In spite of that, such languages can still share, for specific program elements, the previously mentioned characteristics which lead a pure `FSTMerge` approach to not work well for JavaScript. Considering

Java, for example, a class can have blocks of code, called *static blocks*, that are executed at the time of loading their respective class for use (ARNOLD; GOSLING; HOLMES, 2005). Interestingly, static blocks, in Java, are similar to top-level statements in JavaScript, once the order of them in a class, with respect to other static blocks, may be relevant for a program execution and they do not have a unique name.

Listing 4.3 – Example of class in Java with static blocks

```
1  class Test {
2    public static String x = "test";
3
4    public static void f1(String val) {
5      System.out.println("f1: " + val);
6    }
7
8    static {
9      f1(x); // "f1: test"
10     f2(x); // "f2: test"
11   }
12
13   public static void f2(String val) {
14     System.out.println("f2: " + val);
15   }
16
17   static {
18     x = "test2";
19   }
20
21   static {
22     f1(x); // "f1: test2"
23   }
24 }
```

Listing 4.3 illustrates this better by providing an example of a Java class, similar to other examples, with three static blocks. From the moment `Test` class (line 1) is loaded, the three static blocks are executed in sequence. The first static block (lines 8-11) invokes two static methods defined, respectively, before and after the block, which shows that the order of methods can be rearranged without affecting the static block's code execution. The second static block (lines 17-19) changes the value of a static variable, and then the third block (lines 21-23) calls a method that reads its new value. If the last two static blocks were swapped, which could happen during superimposition process, the third block would generate a different result, since it would read a variable not yet changed by the second block. This indicates that, differently from what happens to methods, the order of static blocks does matter. As a consequence, the same problems identified in this work when instantiating `FSTMerge` approach for JavaScript can happen to a semistructured tool

for Java, when it comes to static blocks. The tool may not be able to properly match static blocks, because they do not have proper unique names from grammar annotation, and the superimposition might change the order of static blocks, thus possibly changing the behaviour of a program. But, in general, most Java classes have at most one static block, so these issues tend to be rare (CAVALCANTI; ACCIOLY; BORBA, 2017). JavaScript programs, in turn, often have more than one group of statements among function declarations.

In these three programming languages— PHP, Python, and Java—, there are non-uniquely named elements that are sensitive to order (statements for PHP and Python, and static blocks for Java) at the same level as elements that are not sensitive (class and function declarations for PHP and Python, and method declarations for Java). Not only for these languages, but also for others that have similar characteristics, the decisions made in this work, when designing `jsFSTMerge v1` and `jsFSTMerge v2` to deal with matching and ordering issues, can inspire adaptations to semistructured merge implementations based on `FSTMerge`. First, for PHP, consecutive statements could also be grouped into statement lists, and they could either be renamed (if following the approach for `jsFSTMerge v1`) or, to avoid further matching issues, joined into a single one (if following the approach for `jsFSTMerge v2`). Second, for Python, similar steps could be taken, but there is a need to deal with the fact that functions (and classes) are allowed to be used only after their definition. A simple solution for this, based on what was done for `jsFSTMerge v2`, is joining sibling statement lists into a single node, and then modifying superimposition to guarantee that this node is always the last one among its siblings, appearing after all declaration nodes. Finally, for Java, differently from previous cases, there is no need of adapting the annotated grammar; a static block is already an element that groups consecutive statements. The only modification that could be made is one to the `FSTMerge` merge engine to either assign sequential names to static blocks (as in `jsFSTMerge v1`) or join them into a single static block (as in `jsFSTMerge v2`). The latter avoids additional false positives as mentioned in Subsection 3.1.3.

Even though modifications to grammar and merge engine can solve semistructured merging issues for JavaScript and other languages, making `FSTMerge` approach more generalizable for them is relevant for semistructured merge adoption, as long as support for additional languages would be more easily plugged in. Improvements can be made to `FSTMerge` architecture to better handle program elements that need to be represented as a node in the program structure tree, but that also need to have their order preserved with respect to siblings of the same type. A suggestion is to make grammar annotation more flexible to not only define which elements will be represented as non-terminal and terminal nodes, but also to define which elements are sensitive to order and which ones are not. In the current architecture, both aspects are tangled, as it simply assumes that any element represented as a node (whose production rule is either annotated with `@FSTNonTerminal` or `@FSTTerminal`) have an arbitrary order. For instance, a new parameter (in addition to

`name` and `merge`) could be integrated into `FeatureBNF` format to indicate whether a node should have its order preserved or not. `FSTMerge` engine, then, should use this information to perform a merge respecting this property. As a result, `FSTMerge` would have a better generality by more effectively supporting many programming languages, which was the original intention of Apel et al. (2011) when designing this architecture.

> `FSTMerge` semistructured merge approach, in its pure form, is not effectively generalizable for JavaScript, because of its limitation in representing nodes whose order matters and which do not have unique names as siblings of other nodes whose order is arbitrary. This lack of effective generalizability is also expected to happen to other programming languages, in which statements occur at the same level as commutative and associative declarations.

4.2 COULD SEMISTRUCTURED MERGE FOR JAVASCRIPT BE EFFECTIVE IN PRACTICE?

Currently, industry mainly employs unstructured merge tools to integrate code contributions when using version control systems (KHANNA; KUNAL; PIERCE, 2007). And, more specifically, Git has become the standard system for version control in software development, due to a number of reasons, such as flexibility of distributed repositories and rising popularity of GitHub (BIRD et al., 2009), as discussed in Section 2.1. Git uses, by default, an unstructured tool, based on the *diff3* algorithm, to perform three-way merge. When analyzing the feasibility of a semistructured merge tool to be adopted by industry, replacing unstructured ones, many factors should be taken into account, including usability. Usability can be a major barrier to adoption of any software engineering tool, once programmers will not use a tool if they cannot easily get the result they need from it (FAVRE; ESTUBLIER; SANLAVILLE, 2003).

The different versions of `jsFSTMerge` developed in this work can be executed in a standalone fashion, by just running an executable file, but they can also be automatically integrated with Git, without requiring further configuration. With this integration, a semistructured merge tool is run every time the user invokes a `git merge` command. Additionally, `jsFSTMerge` produces merged revisions with conflict markers in the same format as in the ones generated by a native merge tool from Git. Hence, programmers can continue using Git and, when merging JavaScript revisions, our tool is used in a transparent manner behind Git, while unstructured merge is still employed for programs written in other languages. Usability of `jsFSTMerge`, therefore, is as smooth as possible due to this seamless integration with Git, allowing programmers to keep their typical usage of the version control system.

Besides usability, other factors play critical roles in making a semistructured tool for JavaScript a competitive alternative to traditional textual, line-based tools. This work, in an effort to answer **RQ2** (*Could semistructured merge for JavaScript be effective in practice?*), focuses on two of such factors: 1) integration effort reduction, which leads to an improvement of productivity for developers, and 2) correctness of the merging process, which is associated with the quality of software produced by a merge tool. To evaluate these aspects, we conducted an empirical study, based on a different study conducted by Cavalcanti, Accioly and Borba (2017), which focus on Java systems. We want to investigate their original research questions, which are sub-questions of RQ2 in our study that targets JavaScript systems instead:

- **RQ2.1:** *When compared to unstructured merge, does semistructured merge reduce unnecessary integration effort by reporting fewer spurious conflicts?*
- **RQ2.2:** *When compared to unstructured merge, does semistructured merge compromise integration correctness by missing more non spurious conflicts?*

As in the study conducted by Cavalcanti, Accioly and Borba (2017), we tackle those two research questions by first reproducing merges from the development history of a number of projects from GitHub. Then, we compute *added false positives* and *added false negatives* by unstructured (considering `KDiff3`) and semistructured (considering both `jsFSTMerge v1` and `jsFSTMerge v2`), metrics that were presented in Subsection 3.2, as means of relatively comparing merge tools with respect to development productivity and software quality from their use. In particular, we use added false positives by the analyzed merge tools as a metric to try to answer **RQ2.1**, whereas we use added false negatives to try to answer **RQ2.2**.

There is a methodological difference, which is important to highlight, between this work and the original study with respect to the manner added false positives and false negatives are computed. In Cavalcanti, Accioly and Borba (2017), these metrics are, in fact, approximations of the actual values, because the number of merge scenarios and conflicts analyzed in their work is high enough to impair a manual verification of every conflict that was added either by unstructured or semistructured merge. In our study, on the other hand, the volume of conflicts to be analized is considerably smaller, enabling manual analysis and classification of false positives and false negatives, so we managed to obtain exact values instead of approximations. Further discussion about the differences in the method from which these metrics are computed is presented in Subsection 4.2.1.2.

4.2.1 Evaluation Design

In this empirical study, to compute metrics from real-world JavaScript projects for answering RQ2.1 and RQ2.2, we use a setup based on the one designed by Cavalcanti, Accioly and Borba (2017) with three steps:

- **Mining step**. First, we select GitHub projects that use JavaScript, and then we employ tools to mine their repositories, collecting a number of merge scenarios.
- **Execution step**. Second, we use unstructured and semistructured merge tools to merge collected scenarios, finding candidates of false positives and false negatives.
- **Analysis step**. Finally, we process the list of candidates of false positives and false negatives, filtering part of the conflicts out and manually classifying the remaining list as false positive or negative for each merge tool.

The subsections that follow explain in more detail the steps of this empirical study.

4.2.1.1 Mining Step

With the goal of selecting relevant JavaScript projects, we applied different strategies to find meaningful and diverse projects on GitHub. Initially, similarly to the method Cavalcanti, Accioly and Borba (2017) used to find Java projects, we searched for the top 100 projects that primarily use JavaScript with the highest number of stars on GitHub, which is a metric of project activity and relevance (BORGES; VALENTE, 2018). In addition to this list of 100 projects, we considered other JavaScript projects that got traction in the last years in terms of popularity among developers (GRIEF; RAMBEAU, 2018). From this new list of projects, we discarded the ones that are primarily using ES6 and newer versions of JavaScript, otherwise `jsFSTMerge v1` and `jsFSTMerge v2` would produce the same result as unstructured merge, since an ES6 program would not be properly parsed and a textual merge for the entire artifact would be invoked as a fallback. Subsequently, we selected 50 projects— the same number of projects selected in the study led by Cavalcanti, Accioly and Borba (2017)— from the resulting list of projects, considering number of lines of code, number of developers, and number of commits in order to obtain a diverse range of projects.

Table 1 presents the list of 50 JavaScript projects which are analyzed in this study. We have not systematically selected projects to achieve representativeness (NAGAPPAN; ZIMMERMANN; BIRD, 2013), but we believe that our sample presents a reasonable degree of diversity with respect to, at least, source code size, number of collaborators, number of commits and, also, domain. For example, our sample contains projects that range from Web frameworks to database drivers. Most of them have a focus on Web development, due to the nature of JavaScript, but still covering different aspects of it (e.g., template engines, CSS pre-processors, asynchronous request libraries, etc). Moreover, our sample has varying source sizes and number of developers contributing to the project. For instance, `impress.js` has only 817 lines of code (LOC)— albeit of its small size, it is not a toy project—, whereas `faker.js` has 989,523 LOC. Likewise, `whistle` has only 9 collaborators, while `AngularJS` has 1,599 developers who have contributed to the project.

Table 1 – List of JavaScript projects used as subject systems

Name	URL	LOC	Collaborators	Commits	Merge Commits	Analysed Merge Scenarios
Ace	https://github.com/ajaxorg/ace	212,097	341	7,439	199	195
AngularJS	https://github.com/angular/angular.js	116,483	1,599	8,951	34	34
Async	https://github.com/caolan/async	12,878	216	1,707	372	372
BitcoinJS	https://github.com/bitcoinjs/bitcoinjs-lib	6,812	60	2,403	375	375
Bluebird	https://github.com/petkaantonov/bluebird	39,097	201	2,055	216	216
Bower	https://github.com/bower/bower	9,951	210	2,712	427	427
Bowser	https://github.com/lancedikson/bowser	1,853	65	625	129	129
Brackets	https://github.com/adobe/brackets	382,360	356	17,702	88	88
Chance	https://github.com/chancejs/chancejs	3,800	100	841	195	194
d3	https://github.com/d3/d3	40,633	123	4,157	198	159
director	https://github.com/flatiron/director	5,027	53	747	77	77
Dox	https://github.com/tj/dox	2,418	34	469	55	55
faker.js	https://github.com/Marak/faker.js	989,523	149	984	179	178
fetch	https://github.com/github/fetch	1,706	51	587	117	117
Flux	https://github.com/facebook/flux	2,314	111	388	114	114
GitBook	https://github.com/GitbookIO/gitbook	4,246	85	2,377	207	205
i18next	https://github.com/i18next/i18next	1,995	115	1,284	247	229
impress.js	https://github.com/impress/impress.js	1,144	59	362	59	59
Intro.js	https://github.com/usablica/intro.js	817	74	629	169	169
Istanbul	https://github.com/gotwarlost/istanbul	4,953	78	572	126	126
Jasmine	https://github.com/jasmine/jasmine	24,743	181	1,915	285	285
jQuery	https://github.com/jquery/jquery	38,069	273	6,362	248	245
jquery-pjax	https://github.com/defunkt/jquery-pjax	13,483	62	508	98	98
JSHint	https://github.com/jshint/jshint	46,360	235	2,081	337	337
Konva	https://github.com/konvajs/konva	43,073	107	2,459	262	257
Less.js	https://github.com/less/less.js	126,312	217	2,871	458	443
Mocha	https://github.com/mochajs/mocha	15,727	397	3,072	187	186
Mousetrap	https://github.com/ccampbell/mousetrap	16,137	22	370	36	36
mustache.js	https://github.com/janl/mustache.js	1,295	90	731	134	134
Nightmare	https://github.com/segmentio/nightmare	3,719	111	987	267	267
node_redis	https://github.com/NodeRedis/node_redis	5,551	131	1,251	200	187
node-restify	https://github.com/restify/node-restify	2,999	186	1,610	305	294
numbers.js	https://github.com/numbers/numbers.js	3,241	26	392	92	92
page.js	https://github.com/visionmedia/page.js	2,305	82	628	142	128
Paper.js	https://github.com/paperjs/paper.js	13,658	64	7,177	412	411
Phaser	https://github.com/photonstorm/phaser	396,498	400	12,123	199	198
PM2	https://github.com/Unitech/pm2	21,977	205	4,483	197	193
Pug	https://github.com/pugjs/pug	13,358	208	2,526	459	451
Q	https://github.com/kriskowal/q	8,333	70	894	152	148
Request	https://github.com/request/request	2,791	284	2,263	200	185
RequireJS	https://github.com/requirejs/requirejs	25,068	101	1,405	157	157
reveal.js	https://github.com/hakimel/reveal.js	9,004	227	2,202	389	389
socket.io	https://github.com/socketio/socket.io	1,806	155	1,714	309	290
StatsD	https://github.com/etsy/statsd	2,936	170	934	287	287
Stylus	https://github.com/stylus/stylus	3,501	156	3,908	189	188
three.js	https://github.com/mrdoob/three.js	190,818	1,031	26,208	93	92
Underscore.js	https://github.com/jashkenas/underscore	9,126	258	2,439	238	238
WebTorrent	https://github.com/webtorrent/webtorrent	4,493	116	2,392	306	306
whistle	https://github.com/avwo/whistle	3,022	9	7,498	63	63
Zepto.js	https://github.com/madrobby/zepto	1,594	183	1,516	242	242
Total		**2,891,104**	**9,837**	**161,910**	**10,526**	**10,345**
Mean		**57,822**	**197**	**3,238**	**211**	**207**
Standard Deviation		**158,984**	**256**	**4,665**	**112**	**111**

The next move, after selecting JavaScript projects used as subject systems in the study, is using a tool to mine their corresponding GitHub repositories and extract three-way merge scenarios (see Section 2.2) from the entire history of each repository. In this

study, we adopted `GitMergesMiner`[2] as a tool for this. `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 (i.e., a commit that was created by a `git merge` command and has two parents) has an attribute called `isMerge` with a true value (CAVALCANTI; ACCIOLY; BORBA, 2017). Therefore, to identify merge scenarios, we simply query the ID of all merge commits, checking commits with `isMerge` set as a true. 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).

In total, we extracted 10,526 merge scenarios from the 50 selected JavaScript projects. In the execution and analysis steps, we consider only the JavaScript files from these merge scenarios. A JavaScript project, specially when it is targeted to a Web browser as runtime environment, typically includes files in other formats (e.g., HTML and CSS). Nonetheless, we process only JavaScript files when measuring integration effort and correctness of the merging process. Also, during the execution and analysis steps, we discard merge scenarios that involve JavaScript files with elements not supported by our ES5 grammar (e.g., class declaration). Although we targeted only projects that use the ES5 version of JavaScript, we included a few projects that use, in specific revisions, constructs available only in ES6, which leads to semistructured parsing errors. As can be observed in Table 1, the number of merge scenarios that were analyzed was actually 10,345, meaning that we discarded 191 merge scenarios, which represents only 1.8% of the total number of merge scenarios.

4.2.1.2 Execution and Analysis Steps

After collecting sample projects and merge scenarios as described in the previous step, in the execution step, we run unstructured and semistructured merge tools to reproduce merges of the collected scenarios. These tools take three revisions from each merge scenario (base, left, and right) as input and attempt to merge their files. We used `KDiff3`, mentioned in Subsection 3.2, as unstructured merge tool to be evaluated in this study. As semistructured merge tools, we evaluate `jsFSTMerge v1` and `jsFSTMerge v2`. This way, we are able to compare these two implementations of semistructured merge approach for JavaScript between themselves and compare them to the unstructured merge approach.

The analysis step consists of identifying and computing occurrences of the added false positives and false negatives described in Subsection 3.2. For this, we collect all conflicts reported by the merge tools from execution step and we initially categorize them, by using scripts to automate this process, into three classes: 1) conflict reported by both unstructured and semistructured merge, 2) conflict reported only by unstructured merge, and 3) conflict reported only by semistructured merge (`jsFSTMerge v1` or `jsFSTMerge v2`). To determine if two conflicts, reported by two different merge approaches, are the

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

same, we check if both revision 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 added false positives and false negatives.

Before manually classifying conflicts as added false positives or false negatives for each merge approach, we filter the list of candidates to discard conflicts that are not in JavaScript files (e.g., conflicts in a HTML file), and conflicts that happen due to textual differences in spacing or comments which are reported by unstructured merge, but not by our semistructured merge tools. We decided to not consider the latter, which were manually identified, because such conflicts are not inherently related to the semistructured merge approach, but related to how the unstructured merge is invoked from the semistructured merge to handle composition of terminal nodes. In the case of our semistructured tools, for instance, the unstructured merge that is performed on terminal nodes ignores spacing changes, differently from the default behavior from `KDiff3`, therefore, to avoid bias on the results due to these types of conflicts, they are discarded from this analysis. After filtering out conflicts from the list of candidates of false positives and false negatives, we finally proceed to manually classify them into specific types of false positive and false negative according to the characteristics described in Subsection 3.2, thus precisely computing the occurrence of each type. Table 2 shows the types of false positives and false negatives that can arise when comparing unstructured and semistructured tools for JavaScript.

In Cavalcanti, Accioly and Borba (2017), as mentioned earlier, differently from this work, the metrics to compute added false positives and added false negatives for each approach are approximations of the actual values. They compute the underestimated number of false positives and false negatives added by unstructured merge and the overestimated number of false positives and false negatives added by semistructured merge. This approximation is relevant for their work due to infeasibility of manually analyzing each potential false positive or false negative, which was possible in this work because the obtained number of conflicts added only by unstructured or semistructured merge was considerably lower.

The different types of false positives and false negatives identified by Cavalcanti, Accioly and Borba (2017), when comparing unstructured and semistructured merge approaches for Java systems, are mostly equivalent to the types we identified in this work for JavaScript. Table 2 presents the identified types conflicts for Java (CAVALCANTI; ACCIOLY; BORBA, 2017) along with the ones for JavaScript. Ordering conflicts are false positives for unstructured merge in both programming languages. Renaming conflicts can be either false positives for semistructured merge or false negative for unstructured merge

in the same manner in JavaScript and Java, with the fundamental 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 positives for semistructured merge can be seen as special cases of renaming conflicts, because they are all caused by the same root problem: transformation of function declarations that triggers deletion of nodes— while they are edited by another developer— in program structure trees. A JavaScript false positive for semistructured merge which is an exception for this, not being related to renaming conflict, is the one that occurs only in `jsFSTMerge v1`: the conflicts due to no longer existing one-to-one mapping between statement list nodes (see Subsection 3.1.2). Regarding false negatives for semistructured merge, accidental conflicts happen in both Java and JavaScript for the same reason (changes in consecutive lines that cause unstructured tool to report a conflict, which happens to avoid a semantic error to escape), but in Java, there are additional cases that are specific to the language. One of these cases, the usage of static blocks, is discussed in Section 4.1. Once static blocks do not have unique names, they might cause matching problems during superimposition, leading to extra false negatives.

Table 2 – Types of false positives and negatives identified for JavaScript and Java

Merge Approach	JavaScript		Java	
	False Positives	False Negatives	False Positives	False Negatives
Unstructured	1. Ordering conflict	1. Duplicated function declaration 2. Call to renamed function 3. Early call to no longer hoisted function	1. Ordering conflict	1. Duplicated declaration 2. Reference to renamed declaration
Semistructured	1. Function renaming conflict 2. Function conversion conflict 3. Function declaration displacement conflict 4. No longer existing one-to-one mapping conflict	1. Accidental conflict	1. Renaming conflict	1. Accidental conflict 2. Type ambiguity error 3. Duplicated static block

4.2.2 Evaluation Results

In this empirical study, we analyzed a total of 10,345 merge scenarios obtained from 50 JavaScript projects. Table 3 presents overall results, for each one of the three merge tools considered in this work, about the number of reported conflicts, merge scenarios with conflicts, added false positives and added false negatives. As we can see in this table, `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. `jsFSTMerge v2`, in turn, shows a greater reduction of reported conflicts when compared to unstructured merge, reporting 866 conflicts, a reduction of 6%. In the study carried out by Cavalcanti, Accioly and Borba (2017), it was observed a reduction of 24% of reported conflicts when using a semistructured merge tool based on `FSTMerge` for Java; it is worth noting that this tool leveraged special conflict handlers to solve specific cases.

Concerning merge scenarios with conflicts, `jsFSTMerge v1` and `jsFSTMerge v2` also presented a reduction when compared to unstructured merge, but at a lower rate, which means that a number of different conflicts occur in the same merge scenario, impairing

development productivity in a smaller degree than if they were reported in different merge scenarios. Moreover, we can see that the the number of added false positives decreases from unstructured merge to semistructured merge, being the lowest when using `jsFSTMerge v2`. Detailed results of added false positives and false negatives, for each conflict type and sample project, are available in Appendix A.

Table 3 – Comparing results of unstructured and semistructured merge tools

	`KDiff3`	`jsFSTMerge v1`	`jsFSTMerge v2`
Reported Conflicts	918	884 (-3.85%)	866 (-6%)
Merge Scenarios with Conflicts	582	566 (-2.75%)	557 (-4.3%)
Added False Positives	58	25	7
Added False Negatives	0	1	1

In this study, we rely on the same general assumption made by Cavalcanti, Accioly and Borba (2017) in their research that the higher the number of added false positives, the greater the integration effort, since developers have to spend more time in resolving spurious conflicts. At the same time, we also generally assume that the higher the number of added false negatives, the weaker the guarantee of correctness on the merging process, since more actual interferences are not detected. However, as different conflicts might demand different resolution effort (MENS, 2002; PRUDENCIO et al., 2012; SANTOS; MURTA, 2012), we conduct a qualitative analysis of conflicts to better understand the effort required to resolve each type of conflict, as discussed later in this section.

Regarding performance, we observed a similar execution time relation between unstructured and semistructured merge tools as the one reported by Cavalcanti, Accioly and Borba (2017), who observed that semistructured merge for Java is, on average, 30 times slower than unstructured one. In general, semistructured merge for JavaScript, considering both `jsFSTMerge v1` and `jsFSTMerge v2`, is also slower than unstructured merge, but not prohibitive slower. For example, to reproduce 294 merge scenarios from the *node-restify* project, `jsFSTMerge v1` took 42.16 seconds, `jsFSTMerge v2` took 36.66 seconds, and `KDiff3` took 1.98 seconds. These large differences can be explained by the complexity of superimposing program structure trees, but they certainly can be reduced by means of an industrial strength implementation of our tools, as there are many ways to optimize them (e.g., by exploring parallelization). However, in practice, this difference is often non prohibitive. In the example of *node-restify*, the slowest semistructured merge tool spent, on average, less than 1 second per merge scenario.

In the subsections that follow, we present descriptive statistics related to the research questions RQ2.1 and RQ2.2, which investigate in more detail the impact of using semistructured merge tools, as an alternative to unstructured ones, on the development productivity and merge correctness. We analyze the frequency of added false positives and added false negatives by each merge approach.

4.2.2.1 When compared to unstructured merge, does semistructured merge reduce unnecessary integration effort by reporting fewer spurious conflicts?

For answering RQ2.1, we need to compare the number of false positives added by unstructured merge tool (`KDiff3`) and semistructured merge tools (`jsFSTMerge v1` and `jsFSTMerge v2`). When using unstructured merge (`KDiff3`), our results show that, in our aggregated sample, $0.25\% \pm 0.56\%$ (*average* $\pm$ *standard deviation*) of the merge scenarios have at least one added false positive, and that $7.51\% \pm 16.59\%$ of the reported conflicts are added false positives. When using `jsFSTMerge v1`, we observe that $0.18\% \pm 0.57\%$ of its merge scenarios have at least one added false positive, and that $5.47\% \pm 13.94\%$ of the reported conflicts are added false positives. And, finally, when using `jsFSTMerge v2`, we have that $0.05\% \pm 0.23\%$ of its merge scenarios have at least one added false positive, and that $1.61\% \pm 5.7\%$ of the reported conflicts are added false positives.

For a per project view, Figure 23 presents a violin plot that indicates that semistructured merge, considering both `jsFSTMerge v1` and `jsFSTMerge v2` tools, tends to have fewer merge scenarios with added false positives than unstructured merge. In particular, we can see that the 3rd quartile (upper bound of the black box) of the unstructured tool is higher than zero, meaning that at least 25% of projects of our sample had one or more merge scenarios with added false positive when using `KDiff3`, while the 3rd quartile for semistructured merge tools is zero (no black box visible). When comparing `jsFSTMerge v1` and `jsFSTMerge v2`, we can observe that the latter adds far fewer false positives.

Figure 23 – Per project distribution of percentage of added false positives in terms of merge scenarios

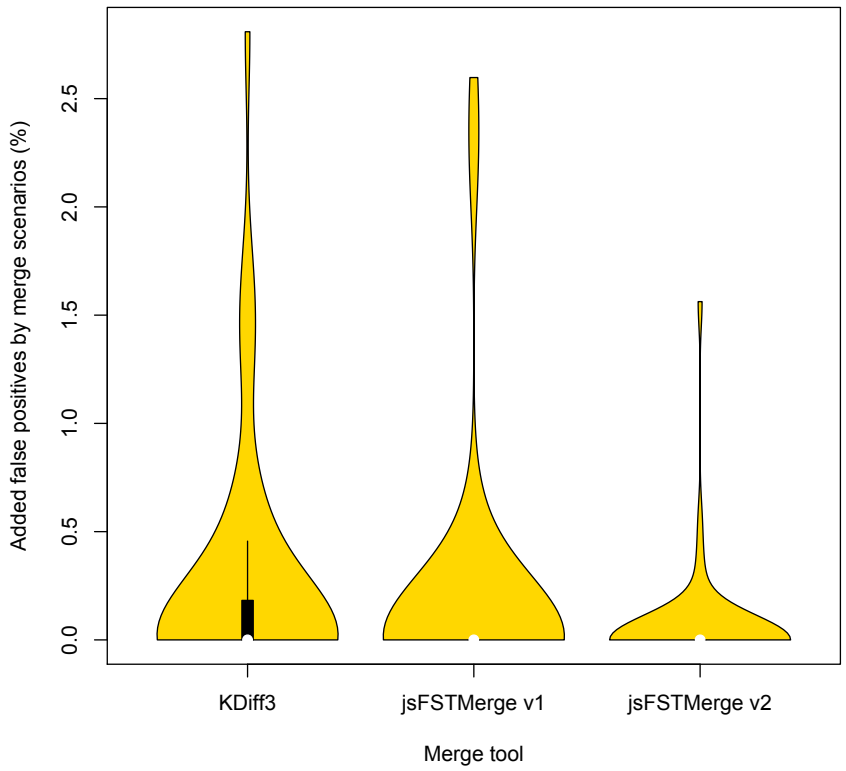
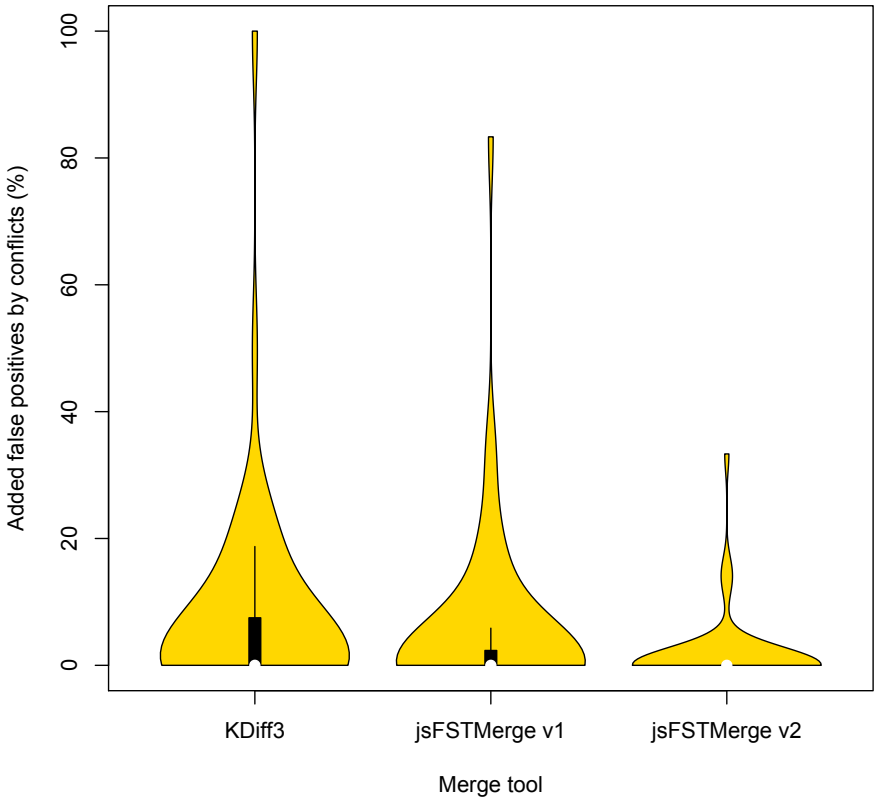


Figure 24 shows a violin plot similar to the previous one, but illustrating the distribution of rate for added false positives in terms of conflicts. A recurring distribution can be observed, with semistructured merge tools having a lower rate of reported conflicts accounted as added false positive than unstructured merge. The maximum percentage for `KDiff3` is 100% and it comes from *impress.js*, as it presented only one conflict which happens to be an ordering conflict.

Figure 24 – Per project distribution of percentage of added false positives in terms of conflicts



Both plots, shown in Figure 24 and Figure 23, suggest that there is no significant difference between the rate of false positives added by `KDiff3` and the ones added by `jsFSTMerge v1`, whereas we can see a difference between `KDiff3` and `jsFSTMerge v2` that seems to be more significant.

To better investigate the statistical significance of differences in added false positives among the evaluated merge tools, we conducted Wilcoxon Signed-Rank tests given that our data are paired, deviate from normality, and come from the same sample (WILCOXON; WILCOX, 1964). First, comparing `jsFSTMerge v1` and `jsFSTMerge v2`, our test shows that there is statistically significant difference, both in terms of merge scenarios and in terms of conflicts (*p-value* equals to, respectively, 0.0295 and 0.00296 < 0.05). Conversely, when comparing `KDiff3` and `jsFSTMerge v1`, a Wilcoxon Signed-Rank test shows, as the plots indicated, that there is no significant difference in percentages of merge scenarios with added false positives (*p-value* equals to 0.204 > 0.05) and reported conflicts accounted

as added false positives (*p-value* equals to $0.275 > 0.05$). And, finally, comparing `KDiff3` and `jsFSTMerge v2`, our test shows that there is indeed 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 $0.003 < 0.05$).

Given we are performing three pairwise comparisons (`jsFSTMerge v1` vs. `jsFSTMerge v2`, `KDiff3` vs. `jsFSTMerge v1` and `KDiff3` vs. `jsFSTMerge v2`), corrections to control for inflation of type-1 error probability become necessary to have a confidence level for the entire family of simultaneous tests (KUTNER; NACHTSHEIM; NETER, 2005). One way of doing that is by applying a Bonferroni correction, which consists of dividing the *p-value* by the total number of paired comparisons performed (MILLER, 1966); in our case, we have $p\text{-}value = 0.05/3 = 0.0166$. Considering this corrected *p-value*, the only differences that remain strongly significant are the ones regarding number of conflicts between `jsFSTMerge v1` and `jsFSTMerge v2` (*p-value* equals to $0.00296 < 0.0166$), and `KDiff3` and `jsFSTMerge v2` (*p-value* equals to $0.003 < 0.0166$). Nevertheless, Bonferroni corrections are overly conservative, so the once significant differences with respect to merge scenarios might still have statistical power.

An additional statistical analysis carried out was a Mann-Whitney's U test (MANN; WHITNEY, 1947) to evaluate differences in added false positives in terms of conflicts which were reported as significant from the previous tests. We found a significant large effect size for the difference between `KDiff3` and `jsFSTMerge v2` (U equals to 199.5, Z equals to 3.24, *p-value* < 0.05, and r equals to 0.46). Furthermore, we found a significant medium effect size for the difference between `jsFSTMerge v1` and `jsFSTMerge v2` (U equals to 55, Z equals to 1.85, *p-value* < 0.05, and r equals to 0.26).

4.2.2.2 When compared to unstructured merge, does semistructured merge compromise integration correctness by missing more non spurious conflicts?

As shown in Table 3, no false negative was identified when using unstructured merge, whereas only one false negative was identified, from our sample, for the semistructured merge tools. This single case is an accidental conflict detected in a merge scenario from the *jQuery* project, as seen in Tables 12, 13, 14, and 15 (Appendix A). This conflict was reported by unstructured merge because developers made changes on consecutive lines in which one developer changed the value of a variable accessed by a function; this is the real case that inspired the example presented in Subsection 3.2.4.

Regarding false negatives for unstructured merge, we have identified different cases which might occur when merging JavaScript programs, as presented in Subsection 3.2.3. Nevertheless, the lack of detected false negatives when performing unstructured merge on our sample is an evidence that such cases might not often occur in real-world projects. The same applies to false negatives added by semistructured merge, once we detected only one occurrence. As a result, considering our sample, there is no statistically significant

difference between unstructured and semistructured merge for JavaScript when it comes to compromising integration correctness.

4.2.3 Discussion

In this subsection, we provide an interpretation for the results obtained in this empirical study to try to answer our research questions, and we also present practical insights about semistructured merge for JavaScript, which can be transferred to other languages.

4.2.3.1 Integration Effort

Both of the semistructured merge tools evaluated in this work, `jsFSTMerge v1` and `jsFSTMerge v2`, reduced the overall number of reported conflicts, when compared to unstructured merge (`KDIff3`), but only `jsFSTMerge v2` introduced fewer false positives with statistical significance. With respect to the number of spurious conflicts that are reported, which cause unnecessary integration effort, `jsFSTMerge v2` effectively presents better results than unstructured merge. And we can also argue that `jsFSTMerge v1`, at least, does not present worse ones. However, in order to properly measure the integration effort reduction caused by the use of a merge tool over another one, we need to analyze the nature of the false positives added by each tool.

Considering an unstructured merge tool, the only case of added false positive, as shown in Table 2, are ordering conflicts. When an ordering conflict involves two function declarations that are 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). The main productivity loss that unstructured merge tools cause, in this case, is the disruption of the development workflow, because the programmer responsible for doing the code integration still needs to shift focus to the conflict resolution, although the conflict itself is simple. As noted by Bird and Zimmermann (2012), regardless of the conflict nature, conflicts still hamper productivity because resolving conflicts might be a time-consuming and error prone activity. And, conversely, ordering conflicts are not always simple to resolve. We also identified what Cavalcanti, Accioly and Borba (2017) called *crosscutting conflicts*, which are ordering conflicts that 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 another function declaration, and, from our sample, they appear to be more often than the simpler conflicts involving only function declarations. Listing 4.4 presents an example of crosscutting conflict observed in a merge scenario from the *stylus* project. We believe that this type of conflict is more difficult to understand and resolve, because it involves different syntactic structures, which might demand more effort from developers.

Listing 4.4 – Example of crosscutting ordering conflict

```
<<<<<<< LEFT
// JS API

describe('JS API', function(){
  it('define a variable with object as hash',
    ...
  });
  ...
});
=======
}

function readDir(dir, ext){
  ext = ext || '.styl';
  return fs.readdirSync(dir).filter(function(file){
    return ~file.indexOf(ext);
  }).map(function(file){
    return file.replace(ext, '');
  });
}

...
>>>>>>> RIGHT
```

Regarding semistructured merge, let us only consider the false positives added both by `jsFSTMerge v1` and `jsFSTMerge v2`. These false positives behave similarly because they are all caused by transformation of function declaration that triggers a deletion of an edited node. When such conflict is reported, it may be initially hard to understand it, because it involves duplicated elements, as we can seen in the examples provided in Subsection 3.2.2. Usually, there is a conflict in an old element (e.g., function declaration) that has a new implementation along with a new element that has an old implementation, whereas there should be only one element. To better illustrate this, see the example of function renaming conflict shown in Subsection 3.2.2.1. In this example, the merge produced by semistructured merge contains a conflict in a function that has the old name along with a function that has the new name and the old body. Even though this type of conflict might cause some confusion to programmers, they tend to be quite easy to resolve, once the combination of contributions from both revisions is straightforward, apart of possible differences in indentation. In the case of function conflict renaming, for instance, it is simply necessary to select the new name from a revision and the new body from another revision. Similar conflict resolutions can be applied for the other types of false positive.

In our sample, semistructured merge, considering both `jsFSTMerge v1` and `jsFSTMerge v2`, reduced the overall number of reported conflicts, and introduced fewer false positives than unstructured merge. Therefore, semistructured merge for JavaScript, specially when using `jsFSTMerge v2`, reduces unnecessary integration effort by reporting fewer spurious conflicts. And, although false positives added by both unstructured and semistructured merge may be hard to understand, the ones added by semistructured merge tend to be easier to resolve.

4.2.3.2 Correctness

We were not able to obtain statistically significant results with respect to false negatives from our empirical study. In our sample, there is minor difference between the number of false negatives introduced exclusively by one of the merge tools, i.e., unstructured and semistructured merge, in practical terms, present the same false negatives. Considering our sample, it is safe to say that semistructured merge does not compromise integration correctness, but, due to lack of statistical significance and potential external validity issues, we cannot guarantee that this holds for other systems.

Despite the frequency of false negatives added by unstructured and semistructured merge approaches seeming to be quite low, we still can analyze how much effort they demand. The false negatives added by unstructured merge, described in Subsection 3.2.3, are issues that do not cause compilation errors, possibly escaping errors to users. We consider that such errors are hard to detect during code integration (in Java, differently, such false negatives are easier to detect as compilation errors guide developers toward the location of the problem (CAVALCANTI; ACCIOLY; BORBA, 2017)). Likewise, the false negatives added by semistructured merge, accidental conflicts as discussed in Subsection 3.2.4, are also hard to detect because they involve behavioural errors that might not be detected during build and testing phases. The dynamic nature of JavaScript (e.g., allowing multiple functions with the same name, but the last ones overriding the first ones) imposes additional challenges when handling behavioural errors.

In our sample, the number of merges leading to false negatives added by unstructured and semistructured merge tools are insignificant. As a result, we have, for our sample, that semistructured merge does not compromise integration correctness, but we cannot guarantee the same for other systems. Nevertheless, we believe that false negatives that can be added by both unstructured and semistructured merge are hard to detect and resolve.

4.2.3.3 `jsFSTMerge v1` or `jsFSTMerge v2`?

The only difference between the results from `jsFSTMerge v1` and `jsFSTMerge v2` comes from the incidence of a type of false positive that occurred only in the former version of the tool. This false positive originates from the lack of maintenance of one-to-one mapping of `StatementList` nodes, as explained in Subsection 3.1.2. All the other metrics and cases, for all sample projects and merge scenarios, are exactly the same for both tools.

In summary, `jsFSTMerge v2` significantly reduced the number of reported conflicts and added false positives, when compared to `jsFSTMerge v1`, thus reducing the integration effort by reporting fewer spurious conflicts. And, at the same time, `jsFSTMerge v2` has the same correctness result as `jsFSTMerge v1`. Consequently, we argue that `jsFSTMerge v2` is far superior to `jsFSTMerge v1` when considering both metrics of development integration effort and correctness of the merging process.

On the other hand, `jsFSTMerge v2` has a drawback when compared to `jsFSTMerge v1`, because it rearranges statement lists, grouping them together, for a certain syntactic level, thus modifying the original format of the code. 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. Reformatting the code, though, might be considered beneficial— in case developers do not already adopt some tool for similar purpose— because it enforces a consistent and organized code structure, having, for example, a group of function declarations followed by a group of statements. In fact, we observed that many JavaScript programs already follow a format which consists of statements at the top of the program that import needed dependencies (similar to `import` declarations in Java), followed by function declarations, and, then, followed by the remaining statements. For example, consider this program:

```
1  // "Import statements"
2  var uuid = require('node-uuid');
3
4  // Function declarations
5  function sendText(req, res) {
6    res.send(req.params.name || uuid());
7  }
8
9  function sendJson(req, res) {
10   res.send({ name: req.params.name || uuid() });
11 }
12
13 // Remaining statements
14 modules.exports = {
15   sendText: sendText,
16   sendJson: sendJson
17 };
```

An improvement that can be made to `jsFSTMerge v2` to combine the best of both worlds, minimizing integration effort while enforcing a good code structure, is introducing changes to the grammar and merge engine to identify these types of statements that use the `require` function (or `import` when targeting ES6). This way, the semistructured merge could be able to produce revisions according to this format, while still avoiding matching issues between `StatementList` nodes.

Alternatively, to completely preserve the original format of the revisions, another idea of improvement— which is more generic than the aforementioned one— for `jsFSTMerge v2` is introducing special markers around statement lists that are joined before superimposition. These markers can indicate the original position of each statement list, by using information such line number in the revision file. An extra step can 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 positions. The result is that the semistructured merge would be able to keep the original format of the revisions, while still avoiding spurious conflicts that happen as a result of one-to-one statement list relationships not being maintained.

4.2.3.4 Unstructured or Semistructured Merge?

Considering the quantitative metrics we used to relatively compare integration effort and correctness of merging process, the number of false positives and false negatives indicate that the semistructured merge, particularly when taking `jsFSTMerge v2` into account, is a better merge approach than unstructured merge. When also analyzing the ease of dealing with cases of false positive and negatives added by each approach, semistructured merge might have some advantage for having false positives that are easier to be solved.

However, there are other factors that influence the adoption of a merge tool in an industrial context. For example, the code reformatting introduced by `jsFSTMerge v2` can be seen, by developers, as a negative aspect for choosing semistructured merge. Additionally, the development phase of a project is also important in this decision, because whether there are too many function conversions, function renaming and other types of refactoring, the developers would have to deal with too many false positives if using semistructured merge.

Performance might also be taken into account when deciding which merge tool to use. As previously mentioned, semistructured merge, due to the complexity of merging trees and possible usage of conflict handlers, is slower than unstructured merge, but generally not prohibitive slower. For instance, the difference between a execution time of 0.05 seconds and 1 second to operate on a merge scenario is often irrelevant for developers in their daily routines, especially because the number of merges performed by a developer tend do be just a few during a day of work.

Another factor to consider is the adoption of JavaScript idioms, which involve the usage of function expressions, that make semistructured merge to provide better or worse results, as discussed in the next subsection. In any case, semistructured merge surely has potential to be a more effective merge tool than unstructured tools. Code formatting, for instance, can be improved as modifications, such as the ones discussed in the previous subsection, are integrated to the tool. Moreover, even though performance does not seem to be an actual barrier for adoption, there is still room for improvement. For example, among many optimizations that could be employed, our approach could explore parallelization, instead of merging files sequentially.

4.2.3.5 When Is Semistructured Merge Better for JavaScript?

The effectiveness of semistructured merge for JavaScript depends mainly on the usage of function declarations in such a manner that they are represented as nodes in a program structure tree. This basically happens when a function declaration does not appear inside a statement, to avoid being represented as textual content of a terminal node. The more a JavaScript program uses function declarations to organize its logic, the more effective the semistructured merge approach is in solving ordering conflicts. If a revision contains JavaScript programs written entirely outside of functions, only by means of statements, semistructured merge would behave exactly as unstructured merge.

In this context, an important discussion is the usage of function expressions assigned to variables as an alternative to function declarations (see Subsection 2.3.2). If a program is written only by using function expressions, in the end, the program structure tree would have only one `StatementList` node, not leveraging the potential of semistructured merge. To better exploit semistructured merge approach, a program needs to use function declarations over function expressions assigned to variables when defining functions. Besides having a better reduction of integration effort, the hoisting of function declarations can avoid runtime errors that arise when using function expressions.

Moreover, there are JavaScript idioms that involve wrapping parts of a program, or even an entire program, into a function expression. For example, a pattern that is used to avoid polluting global scope, as discussed in Subsection 2.3.2, is the usage of an Immediately Invoked Function Expression (IIFE) that might wrap an entire program. Listing 4.5 presents an example of a JavaScript program that consists of an IIFE that contains function declarations. Although there are function declarations in this program (`sendText` and `sendJson`), they are inside a statement. As a consequence, when parsing this program with a semistructured merge tool, they are not represented as nodes; instead, they are just part of an opaque leaf. If two revisions are created from this program, with each one adding a new function declaration below `sendJson`, the merge of such revisions by `jsFSTMerge v1` or `jsFSTMerge v2` would result in a conflict in the same way a unstructured merge tool would.

Listing 4.5 – Example of Function Declarations Inside an Immediately Invoked Function Expression

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

    function sendText(req, res) {
      res.send(req.params.name || uuid());
    }

    function sendJson(req, res) {
      res.send({ name: req.params.name || uuid() });
    }

    modules.exports = {
      sendText: sendText,
      sendJson: sendJson
    };
})();
```

It is important to remember, as discussed in Subsection 2.3.1, that ECMAScript edition 6 (ES6) introduced support for classes, which are, in fact, syntactic sugar over an existing prototype-based pattern that uses functions (ECMA, 2015). Nonetheless, the usage of classes and methods might imply, depending on how an input grammar is annotated when implementing a semistructured merge tool, in more elements represented as nodes in program structure trees, possibly leading to more effective reduction of reported spurious conflicts. Considering a future version of `jsFSTMerge`, with a suitable support for ES6, programs that adopt classes likely will exploit semistructured merge in a more beneficial manner than the ones that neither use classes nor function declarations outside of statements.

4.2.4 Threats to Validity

In this subsection, we discuss the limitations and threats that affect the validity of our empirical study, according to guidelines proposed by Wohlin et al. (2012).

4.2.4.1 Construct Validity

In this study, we measure integration effort mainly based on the amount of false positives added by each merge approach. A quantitative analysis, however, may fail in capturing the actual effort necessary to resolve conflicts. For example, an ordering conflict that involves different types of elements, as discussed in Subsection 4.2.3.1, might require more effort than resolving several function renaming conflicts. To alleviate this concern, in addition to quantitative comparison, we have manually analyzed every added false positive and

false negative, from our sample, to have a better understanding of their impact on the effort in resolving conflicts.

4.2.4.2 Internal Validity

Our approach to collect merge scenarios, based on mining public Git repositories, might represent a threat to the internal validity of our study. Since we analyzed repositories cloned from GitHub, which support commands to rewrite development history, such as `rebase` and `cherry-pick`, we may have lost merge scenarios in which developers had to resolve conflicts. This can explain why the number of conflicts obtained from the mining step in this work was considerably small when compared to the number of collected merge scenarios.

An additional internal validity threat lies in the manual analysis of conflicts that was performed by the authors not only to classify conflicts as false positives or false negatives, but also to discard conflicts that occurred as a result of changes in spaces or comments. This manual analysis is laborious and potentially error-prone, but, in order to avoid classification mistakes, the authors carefully double-checked each analyzed conflict in order to determine whether the given classification were correct or whether the conflict were soundly discarded.

Finally, the fact that we discarded merge scenarios that involve JavaScript files not supported by our semistructured merge implementations could also be considered a threat to the internal validity of the study, because we could be biasing the results in favour of semistructured merge approach. But, in fact, these discarded scenarios represent only 1.8% of the total scenarios, so this portion would not significantly affect our the results.

4.2.4.3 External Validity

Given that the semistructured merge tools developed in this work only support the ES5 version of JavaScript, we ended up restricting the selection of projects to be evaluated, which may be a threat to external validity of this work, once we try to scope JavaScript systems in general. However, most, if not all, false positives and false negatives identified and analyzed in this work are also likely to occur in projects written in newer versions of JavaScript and, going further, in other languages that share similar characteristics with JavaScript, such as Python and PHP.

Moreover, even when considering only projects that use ES5, one could argue that our results may not be generalized for enterprise projects because we used open-source projects in the evaluation. To mitigate this threat, we applied methods, as explained in Subsection 4.2.1.1, to select projects with a good degree of diversity with respect to number of programmers, source code size, and domain. This way, we believe that our results can be applied to other systems, enterprise or not, that use the ES5 version of JavaScript.

5 CONCLUSIONS

In this chapter, we draw our conclusions regarding semistructured merge for JavaScript and `FSTMerge` semistructured approach in general. We summarize the outcome of this thesis in Section 5.1, and we highlight our main contributions in Section 5.2. Section 5.3 presents works that are related to the context of this thesis. Finally, we outline some ideas for future work in Section 5.4.

5.1 SUMMARY

Version control systems support collaborative software development by allowing developers to work on the same project in parallel. However, the integration of independent code contributions might involve conflicting changes, which may demand a substantial effort from programmers for their resolution. In order to reduce such effort, unstructured merge tools, which are the current state of practice in software merging, try to automatically solve conflicts by relying on textual similarity. Conversely, semistructured approach tries to go further by exploiting the syntactic structure of elements involved in a conflict, representing some elements of a software artifact as nodes in a program structure tree, and performing a recursive operation to merge these trees.

Previous studies (APEL et al., 2011; CAVALCANTI; ACCIOLY; BORBA, 2015; CAVALCANTI; ACCIOLY; BORBA, 2017) suggest that semistructured merge might indeed reduce integration effort when compared to unstructured approach, but the industry still heavily relies on the latter. These studies focus mostly on Java, one of the programming languages supported by available tools built on top of `FSTMerge`, a semistructured merge engine developed by Apel et al. (2011). `FSTMerge` requires, as input, an annotated grammar for each supported language. Although the results of such studies can be transferred to other programming languages (e.g., C# and C++), it was uncertain whether semistructured merge would be similarly effective for scripting languages (often used by developers), particularly JavaScript, due to their dynamic and flexible nature. The lack of effective semistructured merge tools for popular languages is a barrier for adoption of semistructured merge in practice, and JavaScript is the most popular programming language for Web applications, also gaining popularity in server-side development since the release of Node.js.

Given that no prior research has been conducted to investigate how effective semistructured merge can be for JavaScript, and considering the relevance of this language for the industry, this thesis proposes and implements three versions of a semistructured merge tool, using the `FSTMerge` architecture as a framework: `jsFSTMerge v0`, `jsFSTMerge v1`, and `jsFSTMerge v2`; all of them supporting the ECMAScript edition 5 (ES5) version of

JavaScript.

The first version, `jsFSTMerge v0`, was an attempt to conceive a well-functioning merge tool by instantiating the `FSTMerge` semistructured approach in a pure fashion, by simply annotating an off-the-shelf grammar. But, since JavaScript syntax allows individual statements at the top-level of a program along with function declarations, the resulting tool based on `FSTMerge` often incorrectly merges programs, reporting too many spurious conflicts, and missing actual conflicts. This failed attempt shed light on the lack of generalizability of the `FSTMerge` approach not only for JavaScript, but also for similar programming languages, such as PHP and Python. We found that, unless adaptations are made to the input grammar and merge engine, **FSTMerge approach is not able to generate an effective merge tool for languages that allow statements at the same level as commutative and associative declarations** (e.g., function declarations in JavaScript). This happens because `FSTMerge` has limitations in representing nodes whose order matters and which do not have unique identifiers as siblings of nodes whose order is arbitrary.

To overcome matching and ordering issues that are present in `jsFSTMerge v0`, preventing it from producing correct merges, we first adapted the JavaScript input grammar to group consecutive statements into a single node, and, then, we introduced, separately, two different changes to the merge engine. The first one, generating `jsFSTMerge v1`, consisted of assigning identifiers to statement lists according to their position with respect to other statement lists within a given scope. This tool can consistently produce valid merges, but it presents an extra type of false positive due to the fact it relies on the maintenance of one-to-one mapping between statement lists. To avoid this type of false positive, while still solving matching and ordering issues, we tried a second improvement in the `FSTMerge` engine, which produced `jsFSTMerge v2`. With this version, statement lists, in a given scope, are joined into a single node, assuring that the matching between statement lists always works as expected. A drawback of `jsFSTMerge v2` is that it might change the original format of the code, since it rearranges group of statements to be in a sequence.

We conducted an empirical study to compare `jsFSTMerge v1` and `jsFSTMerge v2` between themselves and, more importantly, compare them to an unstructured merge tool. By reproducing 10,526 merge scenarios from 50 JavaScript projects, we compared these tools with respect to the occurrence of false positives and false negatives added exclusively by each tool, carrying out a relative comparison. The incidence of false positives is associated with the integration effort demanded by developers when using the respective merge tool, while the incidence of false negatives represents a negative impact to the correctness of the merging process.

We observed that `jsFSTMerge v2` significantly reduced, in our sample, the number of added false positives when compared to `jsFSTMerge v1`, entirely as a result of avoiding

mapping issues between statement lists. Furthermore, `jsFSTMerge v1` and `jsFSTMerge v2` presented exactly the same false negatives, so we can say that the latter, in general, is superior to the former with regards to integration effort reduction and integration correctness. When comparing `jsFSTMerge v2` to an unstructured merge tool, we observed that the **semistructured merge tool reduced the total number of reported conflicts by 6%, significantly reducing the number of added false positives, while not significantly compromising integration correctness**.

Even though the reduction of reported conflicts when using `jsFSTMerge v2`, compared to unstructured merge, is lower than that reported in previous studies for Java and C#, this reduction is still significant and might represent a valuable gain in productivity for developers. Also, this reduction might be higher when software artifacts often use function declarations outside of statements. It is noteworthy to mention that the merge tools used in these other studies avail themselves of special conflict handlers, which were not exploited in this thesis, and they could help to improve the effectiveness of our tools. Nevertheless, semistructured merge seems to be a promising alternative to traditional unstructured merge when working with JavaScript, and it has a considerable room for improvement.

5.2 CONTRIBUTIONS

We summarize our contributions as follows:

- **Implementation of different versions of `jsFSTMerge`**, a semistructured merge tool for JavaScript built on top of `FSTMerge` architecture.
- **Identification of different types of false positives and false negatives** that can be added by unstructured and semistructured merge tools for JavaScript.
- **Analytic evaluation of `FSTMerge` generalizability** for implementing effective semistructured merge tools, for a given programming language, that can be used in practice.
- **Empirical evaluation of `jsFSTMerge v1` and `jsFSTMerge v2`**, comparing them to unstructured merge with respect to how often each tool reports a conflict, and how often each tool misses an actual interference between development tasks.

5.3 RELATED WORK

As mentioned in Section 2.2, a number of studies propose different development tools and strategies to automatically resolve conflicts that arise during merging of revisions, better supporting collaborative development. An example, which was the main research topic of this study, is the semistructured merge approach, proposed by Apel et al. (2011). In this thesis, we evaluate the semistructured approach for JavaScript, and we show evidence

that it can reduce the number of reported conflicts, as previously reported for other programming languages. On the other hand, structured and semantic merge approaches have been proposed to also use information on the language a program is written in to solve conflicts.

Examples of structured merge tools, which incorporate full structural information about a program, include Westfechtel (1991), Buffenbarger (1995), Grass (1992), and Apiwattanapong, Orso and Harrold (2007) (see Subsection 2.2.2). By adopting a hybrid approach, Apel, Lessenich and Lengauer (2012) propose a semistructured merge tool that tunes that 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 language, such as Binkley, Horwitz and Reps (1995), which propose a merge algorithm based on program dependence graphs. In a similar direction, Niu, Easterbrook and Sabetzadeh (2005) propose a domain-independent approach for merging that captures relationships between the structural elements of the programs by also exploiting program dependence graphs. Finally, some merge approaches require that the software artifacts to be merged include a formal semantics (BERZINS, 1994).

These studies have a strong emphasis on the number of reported conflicts. In our work, in turn, we are worried about false positives and false negatives added by different merge approaches not only quantitatively, but also qualitatively. Regarding added false positives, Prudencio et al. (2012) suggest that integration effort can be measured as the number of extra steps (creation, deletion or modification of software artifacts) that a developer needs to carry out in order to soundly integrate changes from independent code contributions. In this context, Santos and Murta (2012) indicate a correlation between this 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 (2013) 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, there are studies which conduct empirical evaluations that provide evidences on the frequency and impact of conflicts. For instance, Brun et al. (2011) and (KASI; SARMA, 2013), similarly to what was performed in our work, reproduced merge scenarios from different GitHub repositories to measure the frequency in which merge scenarios result in conflicts. Likewise, Zimmermann (2007) reproduces merge scenarios from projects using CVS. These works indicate that conflicts, in fact, occur frequently. Cavalcanti, Accioly and Borba (2017) and our study complement such studies by collecting evidences about how often conflicts that demand unnecessary integration occur, as well as how often actual interferences are undetected by different merge tools; in our context, considering JavaScript programs. Regarding added false negatives, Brun et al. (2011) and Kasi and Sarma (2013) investigate the frequency of merge scenarios that presented build or test

failures, which can be seen as consequence of undetected interferences introduced in the merging process. We identified false negatives for JavaScript added by unstructured merge tools, relatively comparing to `jsFSTMerge`, that might not cause build errors, because of the flexibility of the language (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.

5.4 FUTURE WORK

There are many opportunities to improve semistructured merge for JavaScript and the `FSTMerge` semistructured approach in general. Possible ideas for future work include the following:

- **Support for ECMAScript 6 (ES6)**. In this work, we decided to focus on ES5 both because it is still the latest version of JavaScript that is supported by all available Web browsers, and because it is a fully compatible subset of newer versions of the JavaScript, while providing a grammar that is less complex and easier to work with. However, it is crucial, considering the rising adoption of ES6 and newer versions of JavaScript, to extend `jsFSTMerge` to support them. ES6, in particular, introduced classes that can make semistructured merge more effective, once more program elements might be represented as nodes in program structure trees, better leveraging semistructured approach.

- **Improvement of code formatting**. The code reformatting that `jsFSTMerge v2` causes, by rearranging statement lists, can be considered as a barrier of adoption for some developers. Further changes can be added to the input grammar and merge engine to identify special types of statements (e.g., statements that import dependencies), to make the tool generate a merged revision that is as similar as possible to the original revisions. Alternatively, changes can be added to the merge engine to introduce markers around statement lists before joining them, annotating positional information which enables a recovery of the original format of the code. Moreover, improvements can be added to the tool to better handle changes to indentation and comments, as they are important elements for code appearance— this is a general issue concerning `FSTMerge` approach.

- **Improvement of conflict resolution**. First, dedicated handlers can be employed to deal with specific cases of conflicts that occur in terminal nodes. For example, conflicts in statements that import dependencies could be solved by a conflict handler, instead of simply applying unstructured merge, possibly avoiding spurious conflicts. Additionally, Cavalcanti, Accioly and Borba (2017) proposes an improved semistructured tool that employs conflict handlers to use unstructured merge as a

sort of oracle to reduce false positives and false negatives of semistructured merge. Similar approach can be applied to `jsFSTMerge`.

- **Improvement of `FSTMerge` architecture**. As an alternative to modifications to `FSTMerge` engine, to properly handle elements whose order does not matter and that do not have unique names, we could extend the `FeatureBNF` annotation system to allow identification of nodes that are sensitive to order changes among siblings of the same type. Not only `jsFSTMerge` would benefit from this, no longer requiring custom changes to the engine, but, more importantly, other programming languages could benefit as well. Adaptations to input grammar may still be necessary when the granularity of certain elements is too fine (e.g., statements in JavaScript, PHP, and Python). Nonetheless, such support for specification of ordering information, during grammar annotation, would make `FSTMerge` approach more generalizable for additional programming languages, potentially fostering industrial adoption of semistructured merge.

REFERENCES

ALWIS, B.; SILLITO, J. Why are software projects moving from centralized to decentralized version control systems? In: *Proceedings of the 2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering*. [S.l.: s.n.], 2009.

APEL, S.; HUTCHINS, D. A calculus for uniform feature composition. *ACM Transactions on Programming Languages and Systems*, v. 3, n. 5, 2010.

APEL, S.; KäSTNER, C.; LENGAUER, C. Featurehouse: Language-independent, automated software composition. In: *Proceedings of the 31st International Conference on Software Engineering*. [S.l.: s.n.], 2009.

APEL, S.; LENGAUER, C. Superimposition: A language independent approach to software composition. In: *Proceedings of the 7th International Conference on Software Composition*. [S.l.: s.n.], 2008.

APEL, S.; LENGAUER, C.; MöLLER, B.; KäSTNER, C. An algebraic foundation for automatic feature-based program synthesis. *Science of Computer Programming*, v. 75, n. 11, 2010.

APEL, S.; LESSENICH, O.; LENGAUER, C. Structured merge with auto-tuning: Balancing precision and performance. In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. [S.l.]: ACM, 2012.

APEL, S.; LIEBIG, J.; BRANDL, B.; LENGAUER, C.; KäSTNER, C. 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*. [S.l.: s.n.], 2011.

APIWATTANAPONG, T.; ORSO, A.; HARROLD, M. J. Jdiff: A differencing technique and tool for object-oriented programs. *Automated Software Engineering*, Kluwer Academic Publishers, v. 14, n. 1, 2007.

ARNOLD, K.; GOSLING, J.; HOLMES, D. *The Java programming language*. [S.l.]: Addison-Wesley, 2005.

BARROS, G. *Enhancement in conflict detection and resolution when using semistructured merge*. Bachelor's Thesis — Universidade Federal de Pernambuco, 2018.

BERZINS, V. On merging software extensions. *Acta Informatica*, v. 23, n. 6, Nov 1986.

BERZINS, V. Software merge: Semantics of combining changes to programs. *ACM Transactions on Programming Languages and Systems*, ACM, 1994.

BINKLEY, D.; HORWITZ, S.; REPS, T. Program integration for languages with procedure calls. *ACM Transactions on Software Engineering and Methodology*, ACM, 1995.

BIRD, C.; RIGBY, P. C.; BARR, E. T.; HAMILTON, D. J.; GERMAN, D. M.; DEVANBU, P. The promises and perils of mining git. In: *Proceedings of the 6th Working Conference on Mining Software Repositories*. [S.l.: s.n.], 2009.

BIRD, C.; ZIMMERMANN, T. Assessing the value of branches with what-if analysis. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. [S.l.]: ACM, 2012.

BORGES, H.; HORA, A.; VALENTE, M. T. Understanding the factors that impact the popularity of github repositories. In: *32nd IEEE International Conference on Software Maintenance and Evolution*. [S.l.: s.n.], 2016.

BORGES, H.; VALENTE, M. T. What's in a github star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software*, v. 146, Dec 2018.

BRINDESCU, C.; CODOBAN, M.; SHMARKATIUK, S.; DIG, D. How do centralized and distributed version control systems impact software changes? In: *Proceedings of the 36th International Conference on Software Engineering*. [S.l.: s.n.], 2014.

BRUN, Y.; HOLMES, R.; ERNST, M. D.; NOTKIN, D. Proactive detection of collaboration conflicts. In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. [S.l.]: ACM, 2011.

BUFFENBARGER, J. Syntactic software merging. In: *Selected Papers from the ICSE SCM-4 and SCM-5 Workshops, on Software Configuration Management*. [S.l.]: Springer-Verlag, 1995.

CAVALCANTI, G.; ACCIOLY, P.; BORBA, P. Assessing semistructured merge in version control systems: A replicated experiment. In: ACM (Ed.). *Proceedings of the 9th International Symposium on Empirical Software Engineering and Measurement*. [S.l.: s.n.], 2015.

CAVALCANTI, G.; ACCIOLY, P.; BORBA, P. Evaluating and improving semistructured merge. In: ACM (Ed.). *Proceedings of the ACM on programming languages*. [S.l.: s.n.], 2017. v. 28, n. 5.

CONRADI, R.; WESTFECHTEL, B. Version models for software configuration management. *ACM Computing Surveys*, v. 30, n. 2, 1998.

CVS. 2018. <https://www.nongnu.org/cvs/>.

ECMA. *ECMA-262: ECMAScript Language Specification. Edition 5.1*. 2011. <https://www.ecma-international.org/ecma-262/5.1/>.

ECMA. *ECMAScript Language Specification, 6th edition)*. 2015. <http://www.ecma-international.org/publications/standards/Ecma-402.htm>.

ESTUBLIER, J. Software configuration management: a roadmap. *ACM Transactions on Software Engineering and Methodology*, v. 14, n. 4, 2000.

ESTUBLIER, J.; GARCIA, S. Process model and awareness in scm. In: *Proceedings of the 12th International Workshop on Software Configuration Management*. [S.l.: s.n.], 2005.

ESTUBLIER, J.; LEBLANG, D.; HOEK, A. van der; CONRADI, R.; CLEMM, G.; TICHY, W.; ; WIBORG-WEBER, D. Impact of software engineering research on the practice of software configuration management. In: *International Conference on Software Engineering*. [S.l.: s.n.], 2005.

FAVRE, J.; ESTUBLIER, J.; SANLAVILLE, R. Tool adoption issues in a very large software company. In: SPRINGER (Ed.). *Proceedings of 3rd International Workshop on Adoption Centric Software Engineering*. [S.l.: s.n.], 2003.

FLANAGAN, D. *JavaScript: The Definitive Guide*. [S.l.]: O'Reilly, 2011.

FREDERICKSON, B. *Ranking Programming Languages by GitHub Users*. 2018. <https://www.benfrederickson.com/ranking-programming-languages-by-github-users/>.

GALLABA, K.; MESBAH, A.; BESCHASTNIKH, I. Don't call us, we'll call you: Characterizing callbacks in javascript. In: *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. [S.l.: s.n.], 2015.

GIT. 2018. <https://git-scm.com/>.

GITHUB. 2018. <https://github.com/features>.

GOGUEN, J. A.; MESEGUER, J. Security policies and security models. In: IEEE COMPUTER SOCIETY. *IEEE Symposium on Security and Privacy*. [S.l.], 1982.

GOUSIOS, G. The ghtorrent dataset and tool suite. In: *Proceedings of the 10th Working Conference on Mining Software Repositories*. [S.l.: s.n.], 2013.

GRASS, J. E. Cdiff: A syntax directed differencer for c++ programs. In: *Proceedings of the USENIX C++ Conference*. [S.l.]: USENIX Association, 1992.

GRIEF, S.; RAMBEAU, M. *2018 JavaScript Rising Stars*. 2018. <https://risingstars.js.org/2018/en/>.

GRINTER, R. E. Using a configuration management tool to coordinate software development. In: *Proceedings of Conference on Organizational Computing Systems*. [S.l.]: ACM, 1995.

HORWITZ, S.; PRINS, J.; REPS, T. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems*, ACM, v. 11, n. 3, 1989.

HUNT, J. W.; MCILROY, M. D. *An algorithm for differential file comparison. Computer Science*. [S.l.], 1975.

KALLIAMVAKOU, E.; GOUSIOS, G.; BLINCOE, K.; SINGER, L.; GERMAN, D. M.; DAMIAN, D. The promises and perils of mining github. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. [S.l.: s.n.], 2014.

KASI, B. K.; SARMA, A. Cassandra: Proactive conflict minimization through optimized task scheduling. In: *Proceedings of the 35th International Conference on Software Engineering*. [S.l.]: IEEE Press, 2013.

KDIFF3. 2018. <http://kdiff3.sourceforge.net/>.

KHANNA, S.; KUNAL, K.; PIERCE, B. C. A formal investigation of diff3. In: SPRINGER-VERLAG (Ed.). *Proceedings of the 27th International Conference on Foundations of Software Technology and Theoretical Computer Science*. [S.l.: s.n.], 2007.

KUTNER, M.; NACHTSHEIM, C.; NETER, J. *Applied Linear Statistical Models*. [S.l.]: McGraw-Hill Irwin, 2005.

MANN, H.; WHITNEY, D. On a test of whether one of two random variables is stochastically larger than the other. *Annals of Mathematical Statistics*, 1947.

MARTINSEN, J.; GRAHN, H.; ; ISBERG, A. A comparative evaluation of javascript execution behavior. In: SPRINGER (Ed.). *Proceedings of the 11th international conference on Web engineering*. [S.l.: s.n.], 2011.

MENS, T. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, v. 28, n. 5, 2002.

MERCURIAL. 2018. <https://www.mercurial-scm.org/>.

MILLER, R. *Simultaneous Statistical Inference*. [S.l.]: Springern, 1966.

MOZILLA. *MDN Web Docs*. 2018. <https://developer.mozilla.org>.

NAGAPPAN, M.; ZIMMERMANN, T.; BIRD, C. Diversity in software engineering research. In: *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*. [S.l.]: ACM, 2013.

NEDERLOF, A.; MESBAH, A.; DEURSEN, A. V. Software engineering for the web: the state of the practice. In: *36th International Conference on Software Engineering*. [S.l.: s.n.], 2014.

NIU, N.; EASTERBROOK, S.; SABETZADEH, M. A category-theoretic approach to syntactic software merging. In: *Proceedings of the 21st IEEE International Conference on Software Maintenance*. [S.l.]: IEEE, 2005.

O'SULLIVAN, B. Making sense of revision-control systems. *Communications of the ACM*, v. 52, n. 9, 2009.

PEREIRA, C. R. *Introduction to Node.js*. [S.l.]: Apress, 2012.

PERRY, D.; SIY, H.; VOTTA, L. Parallel changes in large-scale software development: an observational case study. *ACM Transactions on Software Engineering and Methodology*, v. 10, n. 3, 2001.

POWELL, T. *Ajax: The Complete Reference*. [S.l.]: McGraw-Hill, Inc., 2008.

PRUDENCIO, J. G.; MURTA, L.; WERNER, C.; CEPEDA, R. To lock, or not to lock: That is the question. *Journal of Systems and Software*, v. 85, n. 2, 2012.

RCS. 2018. <https://www.gnu.org/software/rcs/>.

RICHARDS, G.; HAMMER, C.; BURG, B.; VITEK., J. The eval that men do: A large-scale study of the use of eval in javascript applications. In: SPRINGER (Ed.). *Proceedings the European Conference on Object-Oriented Programming*. [S.l.: s.n.], 2011.

RIGBY, P. C.; BARR, E. T.; BIRD, C.; GERMAN, D. M.; DEVANBU, P. Collaboration and governance with distributed version control. *ACM Transactions on Software Engineering and Methodology*, 2009.

SANTOS, R.; MURTA, L. Evaluating the branch merging effort in version control systems. In: *Proceedings of the 26th Brazilian Symposium on Software Engineering*. [S.l.]: IEEE Computer Society, 2012.

SILVA, L. H.; VALENTE, M. T.; BERGEL, A.; ANQUETIL, N.; ETIEN, A. Identifying classes in legacy javascript code. *Journal of Software: Evolution and Process*, Kluwer Academic Publishers, v. 1, n. 1, 2017.

SOURCEFORGE. 2018. <https://sourceforge.net/>.

STEFANOV, S. *JavaScript Patterns*. [S.l.]: O'Reilly, 2010.

SUBVERSION. 2018. <https://subversion.apache.org/>.

TICHY, W. F. Tools for software configuration management. In: *Proceedings of the International Workshop on Software Version and Configuration Control*. [S.l.: s.n.], 1988.

TILKOV, S.; VINOSKI, S. Node.js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, v. 14, n. 6, 2010.

WESTFECHTEL, B. Structure-oriented merging of revisions of software documents. In: *Proceedings of the 3rd International Workshop on Software Configuration Management*. [S.l.: s.n.], 1991.

WILCOXON, F.; WILCOX, R. A. *Some rapid approximate statistical procedures*. [S.l.]: Lederle Laboratories, 1964.

WOHLIN, C.; RUNESON, P.; HöST, M.; OHLSSON, M. C.; REGNELL, B. *Experimentation in Software Engineering*. [S.l.]: Springer, 2012.

ZAKAS, N. C. *Understanding ECMAScript 6: The Definitive Guide for JavaScript Developers*. [S.l.]: No Starch Press, 2016.

ZIMMERMANN, T. Mining workspace updates in cvs. In: *Proceedings of the Fourth International Workshop on Mining Software Repositories*. [S.l.]: IEEE Computer Society, 2007.

APPENDIX A – DETAILED RESULTS OF INTEGRATION EFFORT AND CORRECTNESS STUDY

Chapter 4 describes an empirical study to compare integration effort and correctness of unstructured and semistructured merge approaches for JavaScript systems. As unstructured tool, we used `KDiff3` in this study. As semistructured merge tools, we evaluated two versions of `jsFSTMerge` developed in this work. Online links to source code, available on GitHub, of both unstructured and semistructured tools are provided below:

- **KDiff3**: https://github.com/KDE/kdiff3
- **jsFSTMerge v1**: https://github.com/AlbertoTrindade/jsFSTMerge/tree/version-1
- **jsFSTMerge v2**: https://github.com/AlbertoTrindade/jsFSTMerge

In this appendix, we present detailed results of integration effort (in terms of added false positives) and correctness (in terms of added false negatives) for each selected JavaScript project and three merge tools considered in this work.

Table 4 – False positives added by unstructured merge with respect to merge scenarios

Project	**Merge Scenarios**	**Merge Scenarios with Ordering Conflicts**	**(%)**
Ace	195	1	0.51
AngularJS	34	0	0
Async	372	2	0.54
BitcoinJS	375	0	0
Bluebird	216	0	0
Bower	427	0	0
Bowser	129	0	0
Brackets	88	0	0
Chance	194	0	0
d3	159	0	0
director	77	0	0
Dox	55	0	0
faker.js	178	5	2.81
fetch	117	0	0
Flux	114	0	0
GitBook	205	0	0
i18next	229	1	0.44
impress.js	59	1	1.69
Intro.js	169	0	0
Istanbul	126	0	0
Jasmine	285	0	0
jQuery	245	1	0.41
jquery-pjax	98	0	0
JSHint	337	0	0
Konva	257	0	0
Less.js	443	3	0.68
Mocha	186	0	0
Mousetrap	36	0	0
mustache.js	134	0	0
Nightmare	267	0	0
node_redis	187	1	0.53
node-restify	294	1	0.34
numbers.js	92	0	0
page.js	128	0	0
Paper.js	411	1	0.24
Phaser	198	0	0
PM2	193	0	0
Pug	451	0	0
Q	148	0	0
Request	185	0	0
RequireJS	157	2	1.27
reveal.js	389	6	1.54
socket.io	290	0	0
StatsD	287	4	1.39
Stylus	188	1	0.53
three.js	92	0	0
Underscore.js	238	0	0
WebTorrent	306	0	0
whistle	63	0	0
Zepto.js	242	0	0
Total	**10,345**	**30**	**0.29**
Mean			**0.26**
Standard Deviation			**0.56**

Table 5 – False positives added by unstructured merge with respect to conflicts

Project	**Conflicts**	**Ordering Conflicts**	**(%)**
Ace	16	3	18.75
AngularJS	17	1	5.88
Async	26	2	7.69
BitcoinJS	7	0	0
Bluebird	8	0	0
Bower	22	0	0
Bowser	22	0	0
Brackets	11	0	0
Chance	25	5	20
d3	81	0	0
director	14	0	0
Dox	6	0	0
faker.js	54	9	16.67
fetch	6	0	0
Flux	8	0	0
GitBook	5	0	0
i18next	28	1	3.57
impress.js	1	1	100
Intro.js	23	0	0
Istanbul	2	0	0
Jasmine	58	0	0
jQuery	31	0	0
jquery-pjax	6	0	0
JSHint	37	6	16.22
Konva	58	4	6.90
Less.js	62	4	6.45
Mocha	2	0	0
Mousetrap	1	0	0
mustache.js	8	0	0
Nightmare	21	1	4.76
node_redis	2	1	50
node-restify	7	1	14.29
numbers.js	19	1	5.26
page.js	12	1	8.33
Paper.js	55	1	1.82
Phaser	4	0	0
PM2	2	0	0
Pug	4	0	0
Q	23	0	0
Request	2	0	0
RequireJS	8	2	25.00
reveal.js	39	9	23.08
socket.io	9	0	0
StatsD	19	4	21.05
Stylus	5	1	20
three.js	8	0	0
Underscore.js	5	0	0
WebTorrent	4	0	0
whistle	7	0	0
Zepto.js	18	0	0
Total	**918**	**58**	**6.32**
Mean			**7.51**
Standard Deviation			**16.59**

Table 6 – False positives added by `jsFSTMerge v1` with respect to merge scenarios

Project	Merge Scenarios	Merge Scenarios with Function Renaming Conflicts	Merge Scenarios with Function Conversion Conflicts	Merge Scenarios with Function Declaration Displacement Conflicts	Merge Scenarios with No Longer Existing One-to-one Mapping Conflicts	(%)
Ace	195	0	0	0	0	0
AngularJS	34	0	0	0	0	0
Async	372	0	0	0	0	0
BitcoinJS	375	0	1	0	0	0.27
Bluebird	216	0	0	0	0	0
Bower	427	0	0	0	0	0
Bowser	129	0	0	0	0	0
Brackets	88	0	0	0	0	0
Chance	194	0	0	0	0	0
d3	159	0	0	0	0	0
director	77	0	0	0	2	2.60
Dox	55	0	0	0	0	0
faker.js	178	0	0	0	0	0
fetch	117	0	0	0	0	0
Flux	114	0	0	0	0	0
GitBook	205	0	0	0	1	0.49
i18next	229	0	0	0	0	0
impress.js	59	0	0	0	0	0
Intro.js	169	0	0	0	0	0
Istanbul	126	0	0	0	0	0
Jasmine	285	0	0	0	0	0
jQuery	245	0	0	1	2	1.22
jquery-pjax	98	0	0	0	0	0
JSHint	337	0	0	0	1	0.30
Konva	257	0	0	0	0	0
Less.js	443	1	0	0	2	0.68
Mocha	186	0	0	0	0	0
Mousetrap	36	0	0	0	0	0
mustache.js	134	0	0	0	0	0
Nightmare	267	0	0	0	1	0.37
node_redis	187	0	0	0	4	2.14
node-restify	294	1	0	0	0	0.34
numbers.js	92	0	0	0	0	0
page.js	128	0	0	2	1	2.34
Paper.js	411	0	0	0	0	0
Phaser	198	0	0	0	0	0
PM2	193	0	0	0	1	0.52
Pug	451	0	0	0	0	0
Q	148	0	0	0	0	0
Request	185	0	1	0	0	0.54
RequireJS	157	0	0	0	0	0
reveal.js	389	0	0	0	0	0
socket.io	290	0	0	0	0	0
StatsD	287	0	0	0	0	0
Stylus	188	0	0	0	0	0
three.js	92	0	0	0	0	0
Underscore.js	238	0	0	0	0	0
WebTorrent	306	0	0	0	1	0.33
whistle	63	0	0	0	0	0
Zepto.js	242	0	0	0	0	0
Total	**10,345**	**2**	**2**	**3**	**16**	**0.22**
Mean						**0.24**
Standard Deviation						**0.59**

Table 7 – False positives added by `jsFSTMerge v1` with respect to conflicts

Project	Conflicts	Function Renaming Conflicts	Function Conversion Conflicts	Function Declaration Displacement Conflicts	No Longer Existing One-to-one Mapping Conflicts	(%)
Ace	13	0	0	0	0	0
AngularJS	16	0	0	0	0	0
Async	24	0	0	0	0	0
BitcoinJS	8	0	1	0	0	12.50
Bluebird	8	0	0	0	0	0
Bower	22	0	0	0	0	0
Bowser	22	0	0	0	0	0
Brackets	11	0	0	0	0	0
Chance	20	0	0	0	0	0
d3	81	0	0	0	0	0
director	16	0	0	0	2	12.50
Dox	6	0	0	0	0	0
faker.js	45	0	0	0	0	0
fetch	6	0	0	0	0	0
Flux	8	0	0	0	0	0
GitBook	6	0	0	0	1	16.67
i18next	27	0	0	0	0	0
impress.js	0	0	0	0	0	0
Intro.js	23	0	0	0	0	0
Istanbul	2	0	0	0	0	0
Jasmine	58	0	0	0	0	0
jQuery	33	0	0	1	2	9.09
jquery-pjax	6	0	0	0	0	0
JSHint	32	0	0	0	1	3.13
Konva	54	0	0	0	0	0
Less.js	61	1	0	0	2	4.92
Mocha	2	0	0	0	0	0
Mousetrap	1	0	0	0	0	0
mustache.js	8	0	0	0	0	0
Nightmare	22	0	0	0	2	9.09
node_redis	6	0	0	0	5	83.33
node-restify	7	1	0	0	0	14.29
numbers.js	18	0	0	0	0	0
page.js	14	0	0	2	1	21.43
Paper.js	54	0	0	0	0	0
Phaser	4	0	0	0	0	0
PM2	3	0	0	0	1	33.33
Pug	4	0	0	0	0	0
Q	23	0	0	0	0	0
Request	3	0	1	0	0	33.33
RequireJS	6	0	0	0	0	0
reveal.js	30	0	0	0	0	0
socket.io	9	0	0	0	0	0
StatsD	15	0	0	0	0	0
Stylus	4	0	0	0	0	0
three.js	8	0	0	0	0	0
Underscore.js	5	0	0	0	0	0
WebTorrent	5	0	0	0	1	20
whistle	7	0	0	0	0	0
Zepto.js	18	0	0	0	0	0
Total	**884**	**2**	**2**	**3**	**18**	**2.83**
Mean						**5.47**
Standard Deviation						**13.94**

Table 8 – False positives added by `jsFSTMerge v2` with respect to merge scenarios

Project	Merge Scenarios	Merge Scenarios with Function Renaming Conflicts	Merge Scenarios with Function Conversion Conflicts	Merge Scenarios with Function Declaration Displacement Conflicts	Merge Scenarios with No Longer Existing One-to-one Mapping Conflicts	(%)
Ace	195	0	0	0	0	0
AngularJS	34	0	0	0	0	0
Async	372	0	0	0	0	0
BitcoinJS	375	0	1	0	0	0.27
Bluebird	216	0	0	0	0	0
Bower	427	0	0	0	0	0
Bowser	129	0	0	0	0	0
Brackets	88	0	0	0	0	0
Chance	194	0	0	0	0	0
d3	159	0	0	0	0	0
director	77	0	0	0	0	0
Dox	55	0	0	0	0	0
faker.js	178	0	0	0	0	0
fetch	117	0	0	0	0	0
Flux	114	0	0	0	0	0
GitBook	205	0	0	0	0	0
i18next	229	0	0	0	0	0
impress.js	59	0	0	0	0	0
Intro.js	169	0	0	0	0	0
Istanbul	126	0	0	0	0	0
Jasmine	285	0	0	0	0	0
jQuery	245	0	0	1	0	0.41
jquery-pjax	98	0	0	0	0	0
JSHint	337	0	0	0	0	0
Konva	257	0	0	0	0	0
Less.js	443	1	0	0	0	0.23
Mocha	186	0	0	0	0	0
Mousetrap	36	0	0	0	0	0
mustache.js	134	0	0	0	0	0
Nightmare	267	0	0	0	0	0
node_redis	187	0	0	0	0	0
node-restify	294	1	0	0	0	0.34
numbers.js	92	0	0	0	0	0
page.js	128	0	0	2	0	1.56
Paper.js	411	0	0	0	0	0
Phaser	198	0	0	0	0	0
PM2	193	0	0	0	0	0
Pug	451	0	0	0	0	0
Q	148	0	0	0	0	0
Request	185	0	1	0	0	0.54
RequireJS	157	0	0	0	0	0
reveal.js	389	0	0	0	0	0
socket.io	290	0	0	0	0	0
StatsD	287	0	0	0	0	0
Stylus	188	0	0	0	0	0
three.js	92	0	0	0	0	0
Underscore.js	238	0	0	0	0	0
WebTorrent	306	0	0	0	0	0
whistle	63	0	0	0	0	0
Zepto.js	242	0	0	0	0	0
Total	**10,345**	**2**	**2**	**3**	**0**	**0.07**
Mean						**0.07**
Standard Deviation						**0.24**

Table 9 – False positives added by `jsFSTMerge v2` with respect to conflicts

Project	Conflicts	Function Renaming Conflicts	Function Conversion Conflicts	Function Declaration Displacement Conflicts	No Longer Existing One-to-one Mapping Conflicts	(%)
Ace	13	0	0	0	0	0
AngularJS	16	0	0	0	0	0
Async	24	0	0	0	0	0
BitcoinJS	8	0	1	0	0	12.50
Bluebird	8	0	0	0	0	0
Bower	22	0	0	0	0	0
Bowser	22	0	0	0	0	0
Brackets	11	0	0	0	0	0
Chance	20	0	0	0	0	0
d3	81	0	0	0	0	0
director	14	0	0	0	0	0
Dox	6	0	0	0	0	0
faker.js	45	0	0	0	0	0
fetch	6	0	0	0	0	0
Flux	8	0	0	0	0	0
GitBook	5	0	0	0	0	0
i18next	27	0	0	0	0	0
impress.js	0	0	0	0	0	0
Intro.js	23	0	0	0	0	0
Istanbul	2	0	0	0	0	0
Jasmine	58	0	0	0	0	0
jQuery	31	0	0	1	0	3.23
jquery-pjax	6	0	0	0	0	0
JSHint	31	0	0	0	0	0
Konva	54	0	0	0	0	0
Less.js	59	1	0	0	0	1.69
Mocha	2	0	0	0	0	0
Mousetrap	1	0	0	0	0	0
mustache.js	8	0	0	0	0	0
Nightmare	20	0	0	0	0	0
node_redis	1	0	0	0	0	0
node-restify	7	1	0	0	0	14.29
numbers.js	18	0	0	0	0	0
page.js	13	0	0	2	0	15.38
Paper.js	54	0	0	0	0	0
Phaser	4	0	0	0	0	0
PM2	2	0	0	0	0	0
Pug	4	0	0	0	0	0
Q	23	0	0	0	0	0
Request	3	0	1	0	0	33.33
RequireJS	6	0	0	0	0	0
reveal.js	30	0	0	0	0	0
socket.io	9	0	0	0	0	0
StatsD	15	0	0	0	0	0
Stylus	4	0	0	0	0	0
three.js	8	0	0	0	0	0
Underscore.js	5	0	0	0	0	0
WebTorrent	4	0	0	0	0	0
whistle	7	0	0	0	0	0
Zepto.js	18	0	0	0	0	0
Total	**866**	**2**	**2**	**3**	**0**	**0.81**
Mean						**1.61**
Standard Deviation						**5.7**

Table 10 – False negatives added by unstructured merge with respect to merge scenarios

Project	Merge Scenarios	Merge Scenarios with Duplicated Function Declaration	Merge Scenarios with Call to Renamed Function	Merge Scenarios with Early Call to No Longer Hoisted Function	(%)
Ace	195	0	0	0	0
AngularJS	34	0	0	0	0
Async	372	0	0	0	0
BitcoinJS	375	0	0	0	0
Bluebird	216	0	0	0	0
Bower	427	0	0	0	0
Bowser	129	0	0	0	0
Brackets	88	0	0	0	0
Chance	194	0	0	0	0
d3	159	0	0	0	0
director	77	0	0	0	0
Dox	55	0	0	0	0
faker.js	178	0	0	0	0
fetch	117	0	0	0	0
Flux	114	0	0	0	0
GitBook	205	0	0	0	0
i18next	229	0	0	0	0
impress.js	59	0	0	0	0
Intro.js	169	0	0	0	0
Istanbul	126	0	0	0	0
Jasmine	285	0	0	0	0
jQuery	245	0	0	0	0
jquery-pjax	98	0	0	0	0
JSHint	337	0	0	0	0
Konva	257	0	0	0	0
Less.js	443	0	0	0	0
Mocha	186	0	0	0	0
Mousetrap	36	0	0	0	0
mustache.js	134	0	0	0	0
Nightmare	267	0	0	0	0
node_redis	187	0	0	0	0
node-restify	294	0	0	0	0
numbers.js	92	0	0	0	0
page.js	128	0	0	0	0
Paper.js	411	0	0	0	0
Phaser	198	0	0	0	0
PM2	193	0	0	0	0
Pug	451	0	0	0	0
Q	148	0	0	0	0
Request	185	0	0	0	0
RequireJS	157	0	0	0	0
reveal.js	389	0	0	0	0
socket.io	290	0	0	0	0
StatsD	287	0	0	0	0
Stylus	188	0	0	0	0
three.js	92	0	0	0	0
Underscore.js	238	0	0	0	0
WebTorrent	306	0	0	0	0
whistle	63	0	0	0	0
Zepto.js	242	0	0	0	0
Total	**10,345**	**0**	**0**	**0**	**0**
Mean					**0**
Standard Deviation					**0**

Table 11 – False negatives added by unstructured merge with respect to conflicts

Project	Merge Scenarios	Duplicated Function Declaration	Call to Renamed Function	Early Call to No Longer Hoisted Function	(%)
Ace	195	0	0	0	0
AngularJS	34	0	0	0	0
Async	372	0	0	0	0
BitcoinJS	375	0	0	0	0
Bluebird	216	0	0	0	0
Bower	427	0	0	0	0
Bowser	129	0	0	0	0
Brackets	88	0	0	0	0
Chance	194	0	0	0	0
d3	159	0	0	0	0
director	77	0	0	0	0
Dox	55	0	0	0	0
faker.js	178	0	0	0	0
fetch	117	0	0	0	0
Flux	114	0	0	0	0
GitBook	205	0	0	0	0
i18next	229	0	0	0	0
impress.js	59	0	0	0	0
Intro.js	169	0	0	0	0
Istanbul	126	0	0	0	0
Jasmine	285	0	0	0	0
jQuery	245	0	0	0	0
jquery-pjax	98	0	0	0	0
JSHint	337	0	0	0	0
Konva	257	0	0	0	0
Less.js	443	0	0	0	0
Mocha	186	0	0	0	0
Mousetrap	36	0	0	0	0
mustache.js	134	0	0	0	0
Nightmare	267	0	0	0	0
node_redis	187	0	0	0	0
node-restify	294	0	0	0	0
numbers.js	92	0	0	0	0
page.js	128	0	0	0	0
Paper.js	411	0	0	0	0
Phaser	198	0	0	0	0
PM2	193	0	0	0	0
Pug	451	0	0	0	0
Q	148	0	0	0	0
Request	185	0	0	0	0
RequireJS	157	0	0	0	0
reveal.js	389	0	0	0	0
socket.io	290	0	0	0	0
StatsD	287	0	0	0	0
Stylus	188	0	0	0	0
three.js	92	0	0	0	0
Underscore.js	238	0	0	0	0
WebTorrent	306	0	0	0	0
whistle	63	0	0	0	0
Zepto.js	242	0	0	0	0
Total	**918**	**0**	**0**	**0**	**0**
Mean					**0**
Standard Deviation					**0**

Table 12 – False positives added by `jsFSTMerge v1` with respect to merge scenarios

Project	**Merge Scenarios**	**Merge Scenarios with Accidental Conflicts**	**(%)**
Ace	195	0	0
AngularJS	34	0	0
Async	372	0	0
BitcoinJS	375	0	0
Bluebird	216	0	0
Bower	427	0	0
Bowser	129	0	0
Brackets	88	0	0
Chance	194	0	0
d3	159	0	0
director	77	0	0
Dox	55	0	0
faker.js	178	0	0
fetch	117	0	0
Flux	114	0	0
GitBook	205	0	0
i18next	229	0	0
impress.js	59	0	0
Intro.js	169	0	0
Istanbul	126	0	0
Jasmine	285	0	0
jQuery	245	1	0.41
jquery-pjax	98	0	0
JSHint	337	0	0
Konva	257	0	0
Less.js	443	0	0
Mocha	186	0	0
Mousetrap	36	0	0
mustache.js	134	0	0
Nightmare	267	0	0
node_redis	187	0	0
node-restify	294	0	0
numbers.js	92	0	0
page.js	128	0	0
Paper.js	411	0	0
Phaser	198	0	0
PM2	193	0	0
Pug	451	0	0
Q	148	0	0
Request	185	0	0
RequireJS	157	0	0
reveal.js	389	0	0
socket.io	290	0	0
StatsD	287	0	0
Stylus	188	0	0
three.js	92	0	0
Underscore.js	238	0	0
WebTorrent	306	0	0
whistle	63	0	0
Zepto.js	242	0	0
Total	**10,345**	**1**	**0.01**
Mean			**0.01**
Standard Deviation			**0.06**

Table 13 – False positives added by `jsFSTMerge v1` with respect to conflicts

Project	Conflicts	Accidental Conflicts	(%)
Ace	13	0	0
AngularJS	16	0	0
Async	24	0	0
BitcoinJS	8	0	0
Bluebird	8	0	0
Bower	22	0	0
Bowser	22	0	0
Brackets	11	0	0
Chance	20	0	0
d3	81	0	0
director	16	0	0
Dox	6	0	0
faker.js	45	0	0
fetch	6	0	0
Flux	8	0	0
GitBook	6	0	0
i18next	27	0	0
impress.js	0	0	0
Intro.js	23	0	0
Istanbul	2	0	0
Jasmine	58	0	0
jQuery	33	1	3.03
jquery-pjax	6	0	0
JSHint	32	0	0
Konva	54	0	0
Less.js	61	0	0
Mocha	2	0	0
Mousetrap	1	0	0
mustache.js	8	0	0
Nightmare	22	0	0
node_redis	6	0	0
node-restify	7	0	0
numbers.js	18	0	0
page.js	14	0	0
Paper.js	54	0	0
Phaser	4	0	0
PM2	3	0	0
Pug	4	0	0
Q	23	0	0
Request	3	0	0
RequireJS	6	0	0
reveal.js	30	0	0
socket.io	9	0	0
StatsD	15	0	0
Stylus	4	0	0
three.js	8	0	0
Underscore.js	5	0	0
WebTorrent	5	0	0
whistle	7	0	0
Zepto.js	18	0	0
Total	**884**	**1**	**0.11**
Mean			**0.06**
Standard Deviation			**0.43**

Table 14 – False positives added by `jsFSTMerge v2` with respect to merge scenarios

Project	**Merge Scenarios**	**Merge Scenarios with Accidental Conflicts**	**(%)**
Ace	195	0	0
AngularJS	34	0	0
Async	372	0	0
BitcoinJS	375	0	0
Bluebird	216	0	0
Bower	427	0	0
Bowser	129	0	0
Brackets	88	0	0
Chance	194	0	0
d3	159	0	0
director	77	0	0
Dox	55	0	0
faker.js	178	0	0
fetch	117	0	0
Flux	114	0	0
GitBook	205	0	0
i18next	229	0	0
impress.js	59	0	0
Intro.js	169	0	0
Istanbul	126	0	0
Jasmine	285	0	0
jQuery	245	1	0.41
jquery-pjax	98	0	0
JSHint	337	0	0
Konva	257	0	0
Less.js	443	0	0
Mocha	186	0	0
Mousetrap	36	0	0
mustache.js	134	0	0
Nightmare	267	0	0
node_redis	187	0	0
node-restify	294	0	0
numbers.js	92	0	0
page.js	128	0	0
Paper.js	411	0	0
Phaser	198	0	0
PM2	193	0	0
Pug	451	0	0
Q	148	0	0
Request	185	0	0
RequireJS	157	0	0
reveal.js	389	0	0
socket.io	290	0	0
StatsD	287	0	0
Stylus	188	0	0
three.js	92	0	0
Underscore.js	238	0	0
WebTorrent	306	0	0
whistle	63	0	0
Zepto.js	242	0	0
Total	**10,345**	**1**	**0.01**
Mean			**0.01**
Standard Deviation			**0.06**

Table 15 – False positives added by `jsFSTMerge v2` with respect to conflicts

Project	Conflicts	Accidental Conflicts	(%)
Ace	13	0	0
AngularJS	16	0	0
Async	24	0	0
BitcoinJS	8	0	0
Bluebird	8	0	0
Bower	22	0	0
Bowser	22	0	0
Brackets	11	0	0
Chance	20	0	0
d3	81	0	0
director	14	0	0
Dox	6	0	0
faker.js	45	0	0
fetch	6	0	0
Flux	8	0	0
GitBook	5	0	0
i18next	27	0	0
impress.js	0	0	0
Intro.js	23	0	0
Istanbul	2	0	0
Jasmine	58	0	0
jQuery	31	1	3.23
jquery-pjax	6	0	0
JSHint	31	0	0
Konva	54	0	0
Less.js	59	0	0
Mocha	2	0	0
Mousetrap	1	0	0
mustache.js	8	0	0
Nightmare	20	0	0
node_redis	1	0	0
node-restify	7	0	0
numbers.js	18	0	0
page.js	13	0	0
Paper.js	54	0	0
Phaser	4	0	0
PM2	2	0	0
Pug	4	0	0
Q	23	0	0
Request	3	0	0
RequireJS	6	0	0
reveal.js	30	0	0
socket.io	9	0	0
StatsD	15	0	0
Stylus	4	0	0
three.js	8	0	0
Underscore.js	5	0	0
WebTorrent	4	0	0
whistle	7	0	0
Zepto.js	18	0	0
Total	**884**	**1**	**0.11**
Mean			**0.06**
Standard Deviation			**0.43**