Semantic conflict detection via dynamic analysis

Amanda Moraes
Centro de Informática
Universidade Federal de Pernambuco
Brazil
ascm@cin.ufpe.br

Paulo Borba
Centro de Informática
Universidade Federal de Pernambuco
Brazil
phmb@cin.ufpe.br

Léuson Da Silva
Polytechnique Montreal
Canadá
leuson-mario-pedro.da-silva@polymtl.ca

Abstract

During collaborative software development, a semantic conflict may occur when the individual behavior expected by different developers is no longer preserved after merging their branches. While potential semantic conflicts are not captured via textual merge tools, different approaches have already been proposed based on static analysis or automated test generation to verify behavioral changes given a merge scenario. However, these approaches share some limitations regarding scalability and reporting false positives and negatives. Trying to address these limitations, in this work, we assess the detection of conflicts by focusing on overriding assignments in JavaScript code through dynamic analysis. Dynamic analysis allows us to detect changes involving writing operations to the same state element at runtime and does not need assertions about the under-analysis code, such as in test-based approaches, requiring only the final version of the merged code (post-merge) to be executed. To evaluate our approach, besides translating test cases from related works, we empirically analyze merge scenarios from 50 JavaScript open-source projects hosted on GitHub, from which we correctly detected one scenario of overriding assignment representing a potential semantic conflict with no false positives.

Keywords: Semantic Conflict Detection, Dynamic Analysis, Overriding Assignment, JavaScript

1 Introduction

In the software development process, adopting tools to manage different versions of projects' files facilitates the evolution and maintenance of the contributions implemented simultaneously by multiple authors over time [4]. However, when contributions from different branches are merged, textual conflicts may be reported by widely used version control tools, which occurs due to changes made to the same region or in consecutive lines of a file [1, 8].

According to Horwitz et al. [15], merge tools that are based on textual differential analysis (e.g., Git and Diff3 [11, 16]) have limited usefulness in the context of software projects as they do not provide guarantees regarding the behavior of the program under integration [3] and do not analyze its content as instructions, only as a textual structure. Behavior, therefore, is a key aspect to semantically analyze whether the merge result of a program is in disagreement with the original individual contributions that originated it. Unlike textual conflicts, behavioral semantic conflicts represent the behavioral differences between a program's pre- and post-merge versions, impacting how the software works when executed (either during test suites or even in production environments). The motivation to detect this kind of conflict is due to the divergence of behavior that occurs even when the integration is textually and syntactically valid [7], resulting in a scenario that incurs costs for software development when not noticed before execution.

The expected behavior intended by a software contributor is hard to assess and, in this context, can be approximated into program specifications about how it should work, while its unexpected change (potential semantic conflict) can be approximated by the concept of interference. According to Horwitz et al. [15], two contributions interfere with each other in an unplanned way if the specifications individually satisfied by each of them are not satisfied by the program that integrates them. Often, the specifications of contributions or the intentions of each contributor participating in an integration scenario are not explicitly defined in the code, but they might be assumed based on documentation [30], comments [28], or implicitly expressed due to the legibility of the code under integration [22]. Hence, the detection of those interferences could reduce potential semantic conflicts [1] and their associated costs, such as broken features or production debugging due to errors introduced with the interfering integrations.

In light of the above, previous works have proposed tools for detecting semantic conflicts based on the concept of interference via static analysis [2, 26]. Such approaches suffer from low scalability of *System Dependence Graphs*, which is adopted to represent programs under analysis, given their computational complexity. Da Silva et al. [5] take a different route by exploring the generation of unit tests. For that, the authors present SAM, a semantic merge tool executed on the four versions of a merge scenario (base, left, right, and merge commits). Conversely, SAM demands the implementation of assertions about third-party code and, therefore, tends to point to more false negative cases relying on the testing coverage achieved and the values used in the resulting test cases.

[1]In this work behavioral semantic conflicts will be simply called semantic conflicts, as other classifications such as static and syntactic will not be addressed

In this study, we delve into the detection of semantic conflicts through dynamic analysis. If compared to the unit test generation approach, the presented solution does not require assertions about the software to be executed and is only expected to run on one merge scenario version (the merge commit one), while the tests must be ran over left, right, base and merge commit to detect interferences. Considering the also mentioned static analysis approach, because the dynamic analysis occurs at runtime, the complexity of the execution is coupled to the original input complexity except for the semantic conflict detection algorithm, which leads to a unique control flow path execution (in static analysis many could be taken into account) that reduces false positives. For that, we explore the Jalangi2 framework to analyze JavaScript code [25] and detect interferences resulting from overriding assignments (OA), similar to the static analysis proposed by Barbosa et al. [2] for Java code.

To validate the proposed dynamic analysis, we translated more than 60 test cases representing positive and negative cases for overriding assignment, which were adapted from the tests defined by Barbosa et al. [2] for its static analysis of overriding assignment in Java code. Furthermore, we perform an empirical study exploring the overriding assignment detection through a sample of 159 merge scenarios extracted from JavaScript open-source repositories. Although most of the samples were considered negatives according to the proposed analysis, our results show the potential of our approach by reporting the detection of one true positive interference with no associated false positives. As contributions to our work, we highlight:

- A new approach to detect semantic conflicts due to overriding assignments based on dynamic analysis;
- A dataset of merge scenarios, with and without semantic conflicts in JavaScript;
- We provide our scripts and related data, supporting replications and running future studies.

The rest of the paper is organized as follows. Section 2 motivates the problem under investigation here, while Section 3 presents our proposed dynamic analysis, focusing on the overriding assignment. In Section 4, we present the methodology steps for our empirical evaluation and its associated results, while discussing them in Section 5. Threats to the validity of our study are described in Section 6 and related works are discussed in Section 7. Finally, in Section 8, we present our conclusions.

2 Concepts and Motivation

The behavioral semantic conflicts this work addresses can be illustrated with the example in Figure 1 (based on [2]). Consider there is a method called `generateReport` whose body contains excerpts inserted by the developers of *Left* and *Right* as a result of merging their branches, respectively. From now on, a *merge* will refer to 3-way merge [10], meaning it creates a merge commit to integrate the changes proposed by two divergent branches with a common ancestor (the base commit). The two contributing branches that originated the merge may be referred to as *Left* and *Right* from here on.

```
class Text {
    constructor() {
        this.text = "";
        this.fixes = 0;
        this.comments = 0;
    }
    generateReport() {
        countDuplicatedWhitespaces(); // Left
        countComments();
        countDuplicatedWords(); // Right
    } ...
```

Figure 1. Example of semantic conflict after the merging of branches *Left* and *Right*

Additionally, assume that, in Figure 1, the *countDuplicatedWhiteSpaces* method invoked by *Left* writes to the `fixes` attribute of the class, which means the number of corrections to be applied due to spaces in white duplicates of `text`. In the same way, the call to `countDuplicatedWords` added by *Right* has a similar behavior by also writing on the attribute `fixes` regarding the duplicate words. In the end, we can assume that the result of the merge between the two branches does not satisfy what one of them (*Left*) implicitly intended with its contribution, as the `fixes` attribute will be last updated according to the definition of just one parent (*Right*).

The analyzed example corresponds to a semantic conflict, whose interference between contributions arises from an overriding assignment (OA). Barbosa et al. [2] mention that there may be OA when changes (additions and modifications) to one of the branches may semantically (i.e., its execution) involve a write operation to a state element that is also associated with a write operation involved in the changes made by the other branch, with no previous existing write operation occurring to the same element between them. In other words, if, in the example of Figure 1, the `countComments` method also wrote to the `fixes` attribute, there would be no conflict as `fixes` would already be known to be overwritten for *Left* before the merging operation. Such an observation is valid as the call for `countComments` is an instruction originating from the merge base, which already existed before any modification of *Left* or *Right*.

According to Misra [18], the state of a program is defined by the values of its variables, whether local or global. During execution, variables translate into memory addresses, and in this sense, arrays and objects, along with their associated elements, fields, and other variables, collectively contribute to the state of the program. As a result, the state elements that will be considered to identify overriding assignments in this study are those that undergo an explicit assignment operation at the code level, and that can be represented not only by variables but also accessed from them (such as via indexes and keys), which are independent and individual states despite those that reference them.

Additionally, we propose an interprocedural dynamic analysis of OA, which is an aspect illustrated by Figure 1. By the definition of overriding assignments, modifications or additions from the integrated branches must **involve** a write operation to a common state element semantically. In the case at hand, it is evident that the addition of two method calls led to conflicting assignments. In the same way, the insertion of explicit writings would also do it.

To summarize, the semantic conflict addressed here consists of unplanned interferences between contributions that cause the behavioral result of the integration to not satisfy some individual specification of them. The solution presented in the following sections aims to dynamically identify OA as a subset of the potential semantic conflicts that can occur in merge scenarios in JavaScript code, due to the availability of tools to build dynamic analyses for the language, that has been a leader in the rankings of open-source software repositories [12].

3 Detecting Interference Using Dynamic Analysis

This section presents our approach for detecting semantic interferences arising from overriding assignments using dynamic analysis on JavaScript code. Currently, the code associated with our approach is available on GitHub [19], where we provide instructions on how to run the analysis and reproduce the study experiments.

3.1 Dynamic Analysis

Our approach is built on the Jalangi2 framework [25], which is based on the discontinued Jalangi tool [27]. Such framework provides an API for a simplified implementation of custom dynamic analyses to be run on JavaScript files based on callbacks that can be optionally defined according to the instructions of interest. Jalangi2 proposes to instrument a script (and its dependencies) defining events associated with program instructions that invoke those pre-defined callbacks at runtime; for example, it executes the `declare` callback when a variable is declared. To perform such an analysis on a script, we start a Node.js [9] process providing the path of the JavaScript input file, additional paths associated to required Jalangi2 native scripts or auxiliary analyses, and extra (optional) parameters as arguments.

All callbacks have as a parameter a static and unique identifier in the file scope that allows the mapping of the current instruction to a location in the original file (path, range of lines, and columns). This functionality is important for the analysis that this study proposes because by knowing the lines where the instruction is located, it is possible to know if it corresponds to any line modified by the branches involved in a merge scenario. Therefore, the instruction's location can be compared with the merge commit metadata provided as an extra parameter to the process so that assignments, function calls, and any other operations can be linked to the contributing branch that added or modified them.

A more detailed diagram of the artifacts involved in performing our custom dynamic overriding assignment analysis is represented in Figure 2. In this representation, one of the extra parameters is the path to a JSON file containing metadata about the merge scenario, which will be accessed by the analysis (*LINE_TO_PARENT_MAP.json*). Throughout the instrumented script callbacks calls, we update two sets that store the identified assignments from *Left* and *Right* (as well as other auxiliary data structures) in pursuit to track the interfering writes and their current context according to the algorithm to be described in Section 3.2. Finally, there is the step of reporting the results at the end of the diagram flow, which refers to interference or error events logging. In successful executions, they are obtained from a list of interference objects filled by the proposed analysis and reported via standard output (or `stdout`, such as the terminal console where the process is running).

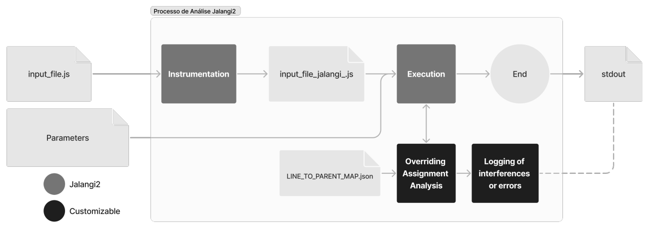


Figure 2. Executing overriding assignment detection analysis with Jalangi2

Our OA dynamic analysis output is event-based and partially illustrated in Figure 3. Initially, the possible events are *ERROR* or *OVERRIDING_ASSIGNMENT*, which can be expanded in future work. For the *OVERRIDING_ASSIGNMENT* event, the *body* key brings as value a short textual description about the detected event. Additionally, the `interference` carries within it the two conflicting assignments along with the identifier of the state element (*targetIdentifier*) that has been overwritten (in the format of *ID_LABEL*, so that elements with the same name are not mismatched). Each assignment from the reported interference contains their location in the original file, the target name and identifier of the written element, as well as the branch to which it is associated, with *L* being equivalent to *Left* and *R* to *Right*. Eventually, if any assignment occurs within a function or method call stack which started with an invocation from some branch, the assignment will also have an object that represents the stack (`functionCallStack`) at the time it was identified, in order to help tracking and verifying the detected interfering changes in the original file.

```
{
  "uuid": "a842cd04-b6e3-462a-847e-3634989a798e",
  "events": [
    {
      "type": "OVERRIDING_ASSIGNMENT",
      "label": "Overriding Assignment Conflict",
      "body": {
        "description": "Interference detected on 33_x ...",
        "interference": {
          "previousAssignment": {
            ...
          },
          "currentAssignment": {
            "id": 33,
            "name": "x",
            "location": "(.../ifWithInvokeConflictSample/index.js:32:33:32:43)",
            "branch": "R",
            "isObject": true,
            "functionCallStack": [
              {
                "id": 417,
                "name": "",
                "location": "(.../ifWithInvokeConflictSample/index.js:27:37:27:44)",
                "branch": "R",
                "beforeInvoke": true
              }
            ]
          },
          "targetIdentifier": "33_x"
  ...
}
```

Figure 3. Example of OA analysis standard output result

3.2 Algorithm

The proposed overriding assignment algorithm was adapted from the algorithm for OA detection based on static analysis defined by Barbosa et al. [2]. The resulting algorithm for this study is represented by the Algorithm 1, which we discuss next.

The script `s` is assumed to be a sequence of events captured by the Jalangi2 callbacks defined in the custom dynamic analysis, such as variables declarations and function calls. We also assume that each branch that participated in the merge operation that generated the current script has a set to store the assignments involved by their additions or modifications, but the same state element cannot have an assignment in both sets simultaneously. Such restriction characterizes the interference we are pursuing, as it means that the same target has already been assigned in previous operations by the other merge branch. When it occurs, the past assignment is removed from its set and the new interference is registered. This way, the overwritten element will always remain associated with the contributing branch whose assignment to it is most recent while we keep track of the interfering ones, see lines 4 to 7 of the Algorithm 1.

If an instruction not mapped to any branch also performs an assignment (which means it is an operation that had not been changed by *Left* or *Right* since the base commit of the merge scenario), the corresponding element must be removed from any assignment set to which it is associated (`removeAssignmentFromAllBranches`). Furthermore, if a function call associated with one of the branches is identified (lines 10 to 11), a stack of function calls starts to be filled until the call that started it is finished and removed from the stack at the starting point (lines 16 to 17). This stack allows a write operation involved by the invoked methods to be properly associated to the branch that caused the assignment (line 3, `¬isFunctionCallStackEmpty`).

Algorithm 1: Overriding Assignment Detection

```
   Data: A script $s$
   Result: A list of overriding assignment interferences
1  for $event \in s$ do
2      if $isWrite(event) \lor isPutFieldPre(event)$ then
3          if $isFromSomeBranch(event) \lor$
             $\neg isFunctionCallStackEmpty()$ then
4              $addAssignmentToCurrentBranch(event)$
5              if $hasAssignmentFromOtherBranch(event)$ then
6                  $updateInterferences(event)$
7                  $removeAssignmentFromOtherBranch(event)$
8          else
9              $removeAssignmentFromAllBranches(event)$
10     if $isInvokeFunPre(event)$ then
11         if $isFromSomeBranch(event) \lor$
             $\neg isFunctionCallStackEmpty()$ then
12             if $isArrayInplaceMethod(event)$ then
13                 $handleAssignedIndices(event)$
14             else
15                 $functionCallStack.push(event)$
16     if $isInvokeFun(event)$ then
17         $functionCallStack.pop()$
18     if $isEndExecution(event)$ then
19         $logResults()$
```

4 Evaluation and Results

An overriding assignment analysis has already been proposed by Barbosa et al. [2], and test cases were made available with assertions about the presence or absence of interferences resulting from OA. Such a test suite allows us to verify the correctness of our proposed algorithm, and its use is explained further in Section 4.1. In addition to the test-based approach, evaluating the solution in the face of uncontrolled scopes is equally important. This way, Section 4.2 explains in more detail the empirical study in which the presented dynamic analysis was performed on merge scenarios that occurred in open-source JavaScript projects.

4.1 State Of The Art Evaluation

The first evaluation strategy adopted for our proposed approach consists of translating the defined test cases originally developed for the static analysis of overriding assignment proposed by Barbosa et al. [2]. In total, there are 71 test cases to check the presence or absence of OA interferences in Java code [14] which represent a diverse set of scenarios missing or containing the interfering assignments to be detected.

Initially, each Java class representing a test case is submitted to a transpilation step to JavaScript, made with JSweet on its online platform[17] to obtain the corresponding JavaScript file for each test case. For specific examples demonstrating the transpilation result, refer to the *test_cases* sub-directory within the solution repository [19].

By default, when the Java class has a static `main` method, the JavaScript code produced already contains the invocation of the corresponding `main` function, which is essential for the dynamic analysis to actually go through the target instructions of the case to be tested. Given this demand, for classes that did not have the static `main` method, they were explicitly added by the authors. Out of 71 Java class files, 2 are categorized as divergences (in behavior) resulting from transpilation: one resulted in JavaScript code with problems due to execution errors and initialization values, while another error was caused by importing non-native language functions that could not be transpiled.

Additionally, the classes that implement each test case also contain annotations in the form of Java comments that indicate which statements or lines are changes from the branch *Left* or *Right* of a supposed merge. With these annotations and the corresponding JavaScript file in hand, the equivalent mapping for the transpiled code was manually generated for each case into a JSON file whose keys represent modified or added lines and the values represent the branches.

Finally, after obtaining the translated scripts and the map of changes per branch, we start a conceptual verification of the expected result for each test case, considering the definition of OA used in this research. Out of 69 cases without transpilation errors, 14 showed conceptual divergence, while 11 of them were related to the related work considering as a combined state the array elements or object fields and their referencing elements (without treating them as independent state components). The expected result is adapted only if there is conceptual divergence to ultimately be used in the final test suite implemented with Jest framework, whose responsibility is to perform our dynamic analysis on each script that represents a test case. All resulting artifacts are available in the [19] solution repository.

Table 1 describes the results achieved with this test-based evaluation strategy. For all analyzed cases, mainly on the 55 that were not subjected to any adaptation, the implemented analysis correctly indicated the presence or absence of overriding assignments.

Table 1. Results obtained with test cases on the overriding assignment algorithm

Transpilation status	No. of test cases	Precision
Transpilation divergence	2	-
Conceptually adapted	14	100%
Success	55	100%

4.2 Empirical Evaluation

The second adopted strategy to evaluate the dynamic analysis for detecting potential semantic conflicts due to overriding assignments consists of an empirical study to detect interferences in merge scenarios extracted from open source projects that use JavaScript as the main programming language. The steps followed for the proposed evaluation are represented in Figure 4. We describe below the methodology, the obtained results and the average runtime overhead of the evaluated samples.

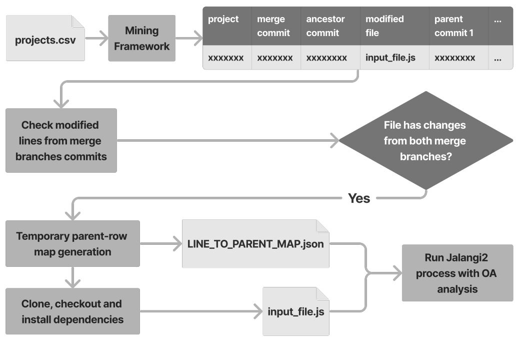

projects.csv
Mining Framework
project
merge commit
ancestor commit
modified file
parent commit 1
...
xxxxxxx
xxxxxxx
xxxxxxxx
input_file.js
xxxxxxxx
...
Check modified lines from merge branches commits
File has changes from both merge branches?
Yes
Temporary parent-row map generation
LINE_TO_PARENT_MAP.json
Clone, checkout and install dependencies
input_file.js
Run Jalangi2 process with OA analysis

Figure 4. Steps and artifacts of the empirical study to evaluate the solution in merge scenarios extracted from open source projects

4.2.1 Collecting Data. Initially, the projects' list made available by Tavares et al. [29] was reused for this study as they presented 50 GitHub repositories with JavaScript code that were selected according to popularity, activity, and preferably using version 5 of ECMAScript; Jalangi2 framework only ensures full support up to ECMAScript 5.1 specification features.

With the defined list of repositories (*projects.csv* in Figure 4), it is provided as input for the *MiningFramework* [13] to extract samples of modified files from the projects 3-way merge commits. The tool then outputs a table where each row represents one modified file path from some of the mined merges along with their parents commits and ancestor commit. Therefore, an additional step in the process of Figure 4 checks whether each reported sample has lines modified by both *Left* and *Right* for the given merge scenario and file, otherwise it can be discarded from the dataset. This step outputs the mapping between lines and branches considering the post-merge file state. This way we ensure our overriding assignment analysis will be performed assuming the correct modified line ranges of files containing possibly interfering changes. However, we do not make any further processing over each sample file, so we cannot ensure they will run successfully as an individual script, nor that they have sufficient code coverage to have their instructions executed.

Finally, with the extracted scenarios and merge metadata available, every sample is post-processed to generate a temporary *JSON* artifact that maps the file's modified lines to each branch of the corresponding merge (*Left* or *Right*), as shown in Figure 4. After this, we clone the GitHub repository, check out to the desired commit and install the dependencies (assuming it is a Node.js project with a *package.json* file). Finally, the resulting script is provided as input, along with the lines-to-branches mapping path as an extra parameter, to a Node.js process that executes the proposed OA analysis.

We achieved a dataset consisting of 452 cases, from which 293 could not be analyzed due to execution errors, leaving then 159 samples executed (65%). In order to evaluate the solution, the 159 scenarios will be considered the base total set, of which 1 had an overriding assignment event detected by our dynamic analysis, see Figure 5. Both the input dataset and their associated outputs are available at the solution implementation repository [19].

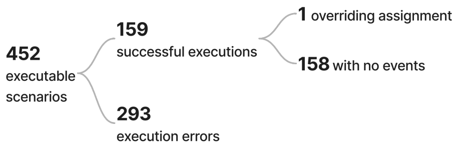


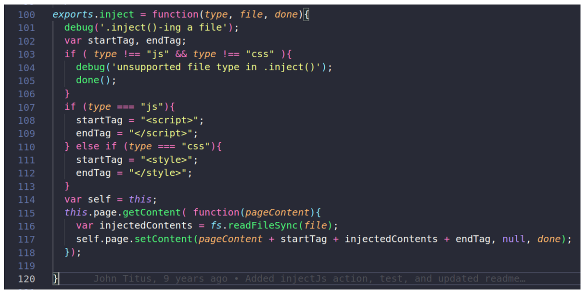
Figure 5. Distribution of projects samples by OA analysis execution result

4.2.2 Results. The implemented OA dynamic analysis identified 1 positive case for an overriding assignment situation which represents a semantic conflict. Hence, there are no false positives to be discussed. The only and true positive case occurred in the Nightmare project (merge commit *3713f3db*, with the modified file *lib/actions.js*) [2]. Figures 6 (line 100) and 7 (line 226) show that *Left* and *Right* added an assignment to the same state element (`inject` attribute of the `exports` object) at different locations in the file (the commits *9f5ffa73* and *786f978* are associated to *Left* and *Right* changes on those assignments, they occurred between the mentioned merge commit and their common ancestor). Because the branch *Left* will have the *inject* function definition discarded due to the addition made by *Right* in a further line to define the same variable as a different function, it causes that element of state to dynamically assume a non expected value by at least one of the merged branches after their integration.

The negative cases represent the major 158 samples from our total set of modified files originated from merge scenarios. To better understand their results, because we do not have a labeled set of samples, we implemented an extra dynamic assignment analysis decoupled from branches: to check how many program state elements (or targets) were assigned more than once along the file execution (counting up to 5 targets), regardless of which assignments were impacted by merge contributions.

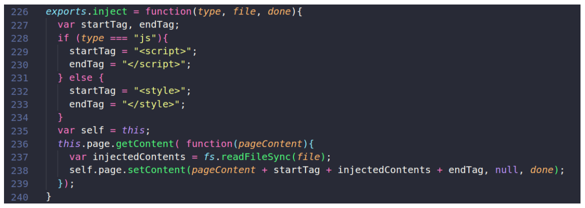
The results show that 132 out of 159 samples (representing 83% of the obtained negative cases) did not contain any target assigned more than once during script execution. Hence,

[2]Available at: https://github.com/segment-boneyard/nightmare/commit/3713...



```
100  exports.inject = function(type, file, done){
101    debug('.inject()-ing a file');
102    var startTag, endTag;
103    if ( type !== "js" && type !== "css" ){
104      debug('unsupported file type in .inject()');
105      done();
106    }
107    if (type === "js"){
108      startTag = "<script>";
109      endTag = "</script>";
110    } else if (type === "css"){
111      startTag = "<style>";
112      endTag = "</style>";
113    }
114    var self = this;
115    this.page.getContent( function(pageContent){
116      var injectedContents = fs.readFileSync(file);
117      self.page.setContent(pageContent + startTag + injectedContents + endTag, null, done);
118    });
119
120  }       John Titus, 9 years ago • Added injectJs action, test, and updated readme…
```

Figure 6. Code snippet modified by commit *9f5ffa73* from branch *Left* of merge commit *3713f3dbd* from the Nightmare project



```
226  exports.inject = function(type, file, done){
227    var startTag, endTag;
228    if (type === "js"){
229      startTag = "<script>";
230      endTag = "</script>";
231    } else {
232      startTag = "<style>";
233      endTag = "</style>";
234    }
235    var self = this;
236    this.page.getContent( function(pageContent){
237      var injectedContents = fs.readFileSync(file);
238      self.page.setContent(pageContent + startTag + injectedContents + endTag, null, done);
239    });
240  }
```

Figure 7. Code snippet modified by commit *786f978* from *Right* branch of merge commit *3713f3dbd* from the Nightmare project

```
Interference detected on 5_inject:
Branch L at (.../Nightmare/lib/actions.js:100:1:120:2)
Branch R at (.../Nightmare/lib/actions.js:226:1:240:2)
```

Figure 8. Description of interference detected by OA analysis

they can be considered true negatives for our OA analysis as an overriding assignment scenario requires at least one write operation from both *Left* and *Right* branches to the same state element. The remaining 27 cases (from which 1 is our true positive), presented at least one overwritten target, which makes an overriding assignment possible, but not guaranteed, as it depends on the changes applied by *Left* and *Right* to truly represent interfering contributions and be characterized as such.

A summary of the overall results achieved on the 159 successfully executed samples for our empirical evaluation is represented in Figure 9. The 100% rate of positives correctly identified and minimum 83% of true negative cases tell us that we actually had a total of 27 samples that could possibly contain overriding assignments, from which 1 (3,7%) was correctly detected by the proposed solution and the remaining 26 were not verified but pointed as negative cases for OA.

4.2.3 Runtime Overhead. In our study, we also evaluated the execution time overhead of the OA dynamic analysis for the 159 successfully executed samples from previous section on a standardized cloud computing environment, as a mean

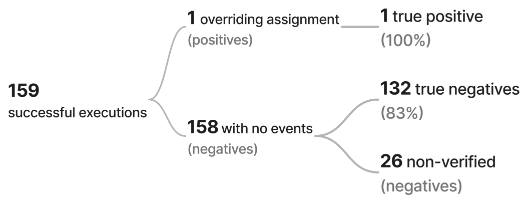


Figure 9. Distribution of successfully executed projects samples by OA analysis result

of assessing the solution performance under reproducible deployment conditions. The featured specification for the environment included three Amazon EC2 instances of type t2.large, 8 GiB of RAM, equipped with 2 vCPUs and Ubuntu 24.04 amd64 as operational system. In each instance, we recorded the elapsed execution time across the mentioned dataset for each of the following categories separately: the original JavaScript file; the instrumented script made by Jalangi2 (only with its native analyses); and finally, the instrumented script with the proposed OA analysis (for the last two we do not consider the time elapsed for instrumentation).

The average execution time of the 3 listed modalities on the dataset is shown in Figure 10. The original scripts exhibit the lowest average execution time (*91 ms*), followed by the instrumented version (*97 ms*) and finally the OA custom analysis (*132 ms*). Figure 11 shows the distribution of the OA analysis relative overhead for the 159 executed samples, with a median of 164,5% increase in execution time and average 172% (55% of interquartile range). Given this value, we also analyzed the intermediate instrumentation version overhead: Figure 11 shows that it represents the major time increase over the original scripts, with a median of 103,5%, while our OA analysis has a median relative overhead of only 34% compared to the instrumented version.

To summarize, in most cases (75%) the applied dynamic analysis framework combined with the OA custom analysis remained in the range that at least doubles the original execution time without tripling it.

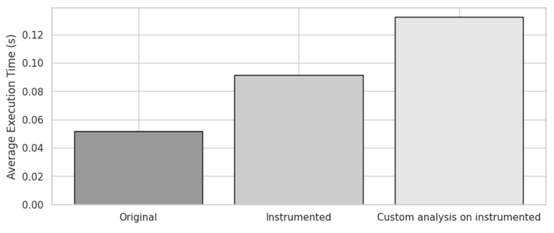


Figure 10. Average execution time for: the original script; its instrumented version; its instrumented version with the proposed custom dynamic analysis

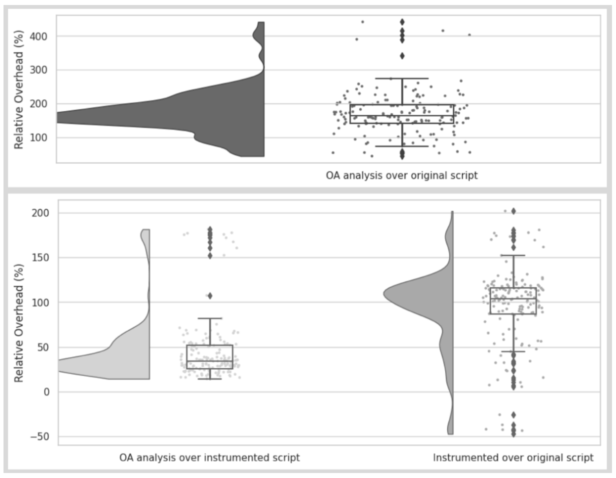


Figure 11. Relative time overhead distribution

5 Discussion

The results achieved with the evaluation of the overriding assignment dynamic analysis based on test cases reinforce the algorithm reliability to detect OA scenarios according to its adopted conceptual definition. The proposed implementation, hence, serves as a pilot for the inclusion of more algorithms that represent semantic interferences in code integration scenarios.

Moreover, the empirical study on real merge scenarios proves the potential for detecting semantic conflicts through the dynamic detection of overriding assignments, as results show that we have correctly detected the 1 positive sample for OA, out of a total of 27 possibly containing it - from Section 4.2.2, we can tell that the major part of our negative cases are actually true negatives as they did not contain any overriden target during execution. Also, the proposed analysis has limited complexity (coupled to the the original program) and overall impact represented by an average execution time increase of 172%, making it suitable to be applied in real code integration pipelines.

Considering the negative rate for OA over the 159 total executions (99%), some of the possible reasons why they represent the majority of samples is that our dynamic analysis is restricted to what is actually invoked in their containing program. Hence, there might be cases where there are little or no instructions being exercised when running their files as input. This scenario reinforce the potential for improving the tool positive rate through the adoption of techniques that are capable of increasing the invoked portion of code given an input script that is possibly not ready to be an entry point. Such an approach, if applied, would still detect interferences dynamically, while it would also take advantage of manipulating the under analysis code to traverse more instructions possibly containing semantic conflicts.

6 Threats to Validity

Internal Validity. When proposing the overriding assignment interference detection via dynamic analysis on JavaScript code, not only the target programming language is a novelty in the field of semantic conflict detection, but the adopted dynamic analysis framework too. As mentioned in Section 4.2.1, Jalangi2 presents a restriction: it provides full support for ECMAScript versions up to 5.1, and may support some ECMAScript 6 features. As a way of addressing this limitation, we employed a chronological filter (using the ES6 launch year, 2015, as the upper limit) on the mined merge scenarios in Section 4.2.1. This way, we could assess a better successful execution rate to focus on the analysis results.

Another limitation comes from the proposed algorithm evaluation method, which is based on test cases originally implemented to evaluate overriding assignment detection in Java classes methods. Considering they had to be transpiled to JavaScript programs, we standardized the process by applying a third party tool [17], leaving only the lines-to-branch mapping process as a manual step.

External Validity. Along with the OA analysis, we also propose a new dataset of open source JavaScript merge scenarios. Because we depend on runtime events to potentially detect an interference, the evaluation of our solution is hence restricted to the diversity of the collected scenarios as well as to the presence of entry point or individually executable files into them. Additionally, the dataset is not labeled according to our definition of semantic conflicts and overriding assignments, but was made available to ensure the study is reproducible and open to future enhancements.

7 Related Work

The dynamic semantic conflicts treated in this study are based on the definition and concept of interference introduced by Horwitz et al. [15], who proposed a formal technique for integrating states of a code without them interfering with each other through the evaluation of differences between their versions. However, along with other works [24][23], approaches based on static techniques and complex structures are applied, such as program slicing and Program Dependence Graphs. In the current work, the approach adopted is dynamic, and the complexity is coupled to the complexity of the code, in addition to only one version being used to identify interfering contribution with its execution.

Nguyen et al. [20] initially focused on exploring conflicts that occur in plugin-based systems and proposed the Varex, a tool to detect them with the variability-aware technique. The same technique was proposed by Nguyen et al. [21], but for detecting merge semantic conflicts with the Semex tool. Semex uses variability-aware execution of tests on a file that is the combination of contributions from the branches of a potential merge by inserting conditional blocks into it. Similar to this work, Semex performs a dynamic analysis, but in return it intervenes in the control flow of the source code and uses existing tests in the projects to execute and detect possible interferences.

Still in the context of semantic conflicts, Da Silva et al. [6] also proposed the detection of interference between versions of a code integration scenario dynamically. From the generation of unit tests, a comparison of results is carried out with their execution to identify changes in behavior across all commits related to a 3-way merge scenario. In addition to being a test-based solution, the work proposes executing more than one version of the code, while in this study, only the merge commit version is executed.

Finally Santos et al. [26] defined algorithms that characterize interferences, such as overriding assignment, but for Java code. Barbosa et al. [2] specifically proposed static OA analysis for Java classes with two approaches: intraprocedural and interprocedural. The defined algorithm was the main reference for implementing the dynamic analysis of this study, which resulted in an interprocedural adaptation to run on JavaScript code.

8 Conclusions

The dynamic analysis of overriding assignment presented in this research corresponds to a tool capable of detecting unplanned interference between contributions integrated into a program. Our approach had its algorithm verified with tens of test cases adapted from related literature, and its evaluation on real integration scenarios extracted from open source projects detected only true positives while resulted in a minimum of 85% true negative rate. As this analysis occurs at runtime, its positive rate can be expanded in future work with approaches that intervene in the source code to help more instructions defined in the input script to be actually executed and analyzed. In order to achieve this goal, applying fuzzers to insert invokes and reusing parameters or tests from the repositories where the merge took place are techniques that might be adopted. Extending the tool with algorithms beyond detecting overriding assignment is also expected, so the analyses will cover more patterns of unplanned interference between developers.

Finally, given its dynamic behavior, positive performance, and limited complexity, the solution also has the potential for being applied into integration pipelines or integration proposals in code repositories as a mean to actively anticipate the conflicts and reduce the impact of behavioral interfering changes during collaborative software development.

9 Acknowledgements

We would like to thank INES (National Software Engineering Institute) and the Brazilian research funding agencies CNPq (grant 309235/ 2021-9), FACEPE (grants IBPG-0546-1.03/15 and APQ/0388-1.03/14), and CAPES for partially supporting this work.

References

[1] Paola Accioly, Paulo Borba, and Guilherme Cavalcanti. 2018. Understanding semi-structured merge conflict characteristics in open-source java projects. *Empirical Software Engineering* 23 (2018), 2051–2085.

[2] Matheus Barbosa, Paulo Borba, Rodrigo Bonifácio, and Galileu Santos. 2022. Semantic conflict detection with overriding assignment analysis. *SBES 2022: XXXVI Brazilian Symposium on Software Engineering* (2022).

[3] Yuriy Brun, Reid Holmes, Michael D Ernst, and David Notkin. 2013. Early detection of collaboration conflicts and risks. *IEEE Transactions on Software Engineering* 39, 10 (2013), 1358–1375.

[4] Reidar Conradi and Bernhard Westfechtel. 1998. Version models for software configuration management. *ACM Computing Surveys (CSUR)* 30, 2 (1998), 232–282.

[5] Léuson Da Silva, Paulo Borba, Toni Maciel, Wardah Mahmood, Thorsten Berger, João Moisakis, Aldiberg Gomes, and Vinícius Leite. 2024. Detecting semantic conflicts with unit tests. *Journal of Systems and Software* (2024), 112070.

[6] Léuson Da Silva, Paulo Borba, Wardah Mahmood, and Thorsten Berger. 2020. Detecting Semantic Conflicts via Automated Behavior Change Detection. *IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2020). https://doi.org/10.1109/ICSME46990.2020.00026

[7] Léuson Da Silva, Paulo Borba, and Arthur Pires. 2022. Build conflicts in the wild. *Journal of Software: Evolution and Process* 34, 4 (2022), e2441.

[8] GitHub Docs. 2024. Resolver um conflito de merge usando a linha de comando. https://docs.github.com/pt/pull-requests/collaborating-with-pull-requests/addressing-merge-conflicts/resolving-a-merge-conflict-using-the-command-line

[9] OpenJS Foundation. [n. d.]. Node.js — Run JavaScript Everywhere. https://nodejs.org/en

[10] Git. 2024. Branches no Git - O básico de Ramificação (Branch) e Mesclagem (Merge). https://git-scm.com/book/pt-br/v2/Branches-no-Git-O-b%C3%A1sico-de-Ramifica%C3%A7%C3%A3o-Branch-e-Mesclagem-Merge

[11] Git. 2024. Resolver um conflito de merge usando a linha de comando. GitHub. https://git-scm.com/

[12] Innovation Graph. 2024. Programming Languages Global Metrics. https://innovationgraph.github.com/global-metrics/programming-languages

[13] Software Productivity Group. [n. d.]. MiningFramework. GitHub. https://github.com/spgroup/miningframework

[14] Software Productivity Group. [n. d.]. Static Analyses Algorithms for Detecting Semantic Conflicts. GitHub. https://github.com/spgroup/conflict-static-analysis/tree/2f481eb58b/src/test/java/br/unb/cic/analysis/samples/ioa

[15] Susan Horwitz, Jan Prins, and Thomas Reps. 1989. Integrating Noninterfering Versions of Programs. *ACM Transactions on Programming Languages and Systems* 11, 3 (1989), 345–387.

[16] Free Software Foundation Inc. 2022. Merging From a Common Ancestor. https://www.gnu.org/software/diffutils/manual/html_node/diff3-Merging.html

[17] JSweet. [n. d.]. JSweet Live Sandbox: write Java, run JavaScript. https://www.jsweet.org/jsweet-live-sandbox/

[18] Jayadev Misra. 2001. *A Discipline of Multiprogramming: Programming Theory for Distributed Applications*. Springer. 14 pages.

[19] Amanda Moraes. [n. d.]. Dynamic Analysis for detecting overriding assignments in JavaScript code. GitHub. https://github.com/amandascm/SCAz.js

[20] Hung Viet Nguyen, Christian Kastner, and Tien Nhut Nguyen. 2014. Exploring Variability-Aware Execution for Testing Plugin-Based Web Applications. In *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014, 907–918.

[21] Hung Viet Nguyen, My Huu Nguyen, Soncuu Dang, Christian Kastner, and Tien Nhut Nguyen. 2015. Detecting semantic merge conflicts with variability-aware execution. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015, 926–929.

[22] Delano Oliveira, Reydne Bruno, Fernanda Madeiral, and Fernando Castor. 2020. Evaluating code readability and legibility: An examination of human-centric studies. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 348–359.

[23] Thomas W Reps and Susan Horwitz. 1992. The use of program dependence graphs in software engineering. *ICSE '92: Proceedings of the 14th international conference on Software engineering* (1992), 392–411.

[24] Thomas W Reps, Susan Horwitz, and Dave Binkley. 1988. Interprocedural slicing using dependence graphs. In *ACM SIGPLAN Notices*, Vol. 23. ACM, 35–46.

[25] Samsung. [n. d.]. Jalangi2. GitHub. https://github.com/Samsung/jalangi2

[26] Galileu Santos, Paulo Borba, Rodrigo Bonifácio, and Matheus Barbosa de Oliveira. 2023. Detecting Semantic Conflicts using Static Analysis. *arXiv:2310.04269 [cs.SE]* (2023).

[27] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript. In *ESEC/FSE 2013: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 488–498. https://doi.org/10.1145/2491411.2491447

[28] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. 2013. Quality analysis of source code comments. In *2013 21st international conference on program comprehension (icpc)*. Ieee, 83–92.

[29] Alberto Tavares, Paulo Borba, Guilherme Cavalcanti, and Sérgio Soares. 2019. Semistructured Merge in JavaScript Systems. In *34th IEEE/ACM International Conference on Automated Software Engineering (ASE 2019)*. 1014–1025.

[30] Junji Zhi, Vahid Garousi-Yusifoğlu, Bo Sun, Golara Garousi, Shawn Shahnewaz, and Guenther Ruhe. 2015. Cost, benefits and quality of software development documentation: A systematic mapping. *Journal of Systems and Software* 99 (2015), 175–198.