Concolic Testing of Full-Stack JavaScript Applications

Vandercammen, Maarten; Christophe, Laurent; De Meuter, Wolfgang; De Roover, Coen

Published in:
Proceedings of the 17th Belgium-Netherlands Software Evolution Workshop

Publication date:
2018

Concolic Testing of Full-Stack JavaScript Applications

Position Paper

Maarten Vandercammen
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
Maarten.Vandercammen@vub.be

Laurent Christophe
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
Laurent.Christophe@vub.be

Wolfgang De Meuter
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
Wolfgang.De.Meuter@vub.be

Coen De Roover
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
Coen.De.Roover@vub.be

***Abstract*—Recent years have seen the rise of so-called *full-stack* JavaScript web applications, where both the client and the server side of the web application are developed in JavaScript. Both sides communicate with each other via asynchronous messages, as enabled by e.g., WebSockets.**

Traditionally, automated whitebox testing of web applications involves testing both sides of the application in isolation from each other. However, this approach lacks a holistic overview of the entire web application under test. This leads to inaccuracies in the types of program bugs that are reported by the tester, and makes it more difficult for developers to understand how the behaviour of the client may affect the behaviour of the server, and vice versa. An interesting side-effect of the evolution towards full-stack applications is that a single automated tester can be developed that observes the execution of both parts of the system simultaneously, thereby remedying the aforementioned issues. In this paper, we examine the benefits and design challenges in employing such a holistic approach towards testing full-stack applications, and we demonstrate StackFul, the first concolic tester for full-stack JavaScript applications.

***Index Terms*—Automated testing, Concolic testing, Full-Stack JavaScript applications**

I. Introduction

The World Wide Web is increasingly composed of dynamic web *applications*, such as online spreadsheets, chat applications, and file sharing services. They are defined by their heavy reliance on user interactions and their event-driven nature. These web applications can usually be divided in a client and a server side component. JavaScript has long been a very attractive implementation language for building the client side of these applications, but since the advent of Node.js and its associated ecosystem, JavaScript is becoming an increasingly attractive choice for developing the server side of the web application as well. Web applications where the entire technology stack is implemented in JavaScript are called *full-stack* web applications. In these applications, the client and the server often communicate with each other by using JavaScript libraries such as WebSocket or Socket.IO.

The dynamic nature of JavaScript, featuring e.g., prototype-based inheritance, dynamic code evaluation, and dynamic property creation and deletion, makes it difficult to statically verify the correctness of programs. In order to detect program errors, developers therefore often rely either on manually written tests or on automated testing tools that can find these bugs for them.

Traditionally, automated whitebox testing tools test the client and server side of the application in isolation from each other. In this paper, we assert that the lack of a global overview of the behaviour of the application is detrimental to the accuracy of the bug reports and to the programmer's comprehension of how program bugs that are reported by the tester may arise in practice. As an example, consider an application where users fill in some input forms, and where the data is then sent to the server. The client side of this application could check whether the entered data is correct: for example, when creating a new account for a website, a user may be required to enter their e-mail address. The client could check whether the entered string indeed refers to a valid e-mail address by matching the string against a regex. However, when testing the server of such an application in isolation from the client, the tester is unaware of the fact that the e-mail address passed in the registration form has already been verified by the client, and may incorrectly assume that invalid e-mail strings are possible, thereby leading the tester to flag superfluous program errors.

Fortunately, the advent of full-stack applications has made it possible to test both sides of the application simultaneously, using the same testing tool, thereby making it possible to alleviate these issues. Until now however, this advantage has not yet been exploited by existing tools. In this paper, we give a brief overview of StackFul, a concolic tester for full-stack JavaScript applications, where the tester observes the execution of both sides of the web application simultaneously.

II. Background

We first give some background information on concolic testing. *Concolic testing* [1], [2] is a technique for performing automated testing of programs. The goal of the tester is to iteratively explore all feasible paths in a program by manipulating the values of non-deterministic parameters to

```
1  function twice(v){
2    return v * 2;
3  }
4  function f(x, y){
5    var z = twice(y);
6    if(z==x) {
7      if(x > y + 10) {
8        throw new Error();
9      }
10   }
11 }
12 var x = Math.random();
13 var y = Math.random();
14 f(x,y);
```

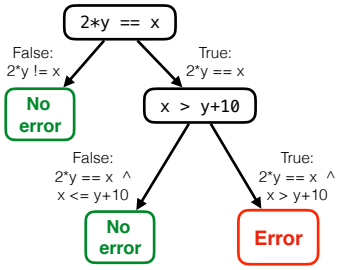

Fig. 1. A program and its symbolic execution tree, based on Figure 1 of [5]

the program, such as program arguments, user input, random numbers, values that are read from files etc. [3], [4]. To this end, the concolic tester simultaneously performs *concrete* and *symbolic* execution. In each iteration, the concrete execution steers the tester towards a certain program path and reports any program error it might encounter, while the symbolic execution gathers constraints on the program's execution so the tester may visit other program paths in subsequent iterations.

We illustrate the working of a concolic tester via the JavaScript program depicted in Figure 1. In this program, the variables `x` and `y` receive a random value. They are symbolically represented as the input parameters x and y. Suppose that in the first iteration, the concolic tester randomly assigns the values `3` to `x` and `5` to `y`. These values cause the condition on line 6 to be false, and the program terminates without reporting an error. Simultaneously with this concrete execution, the tester also collects the symbolic representation of the conditional predicate that was encountered on line 6 in the form of a so-called *path constraint*, i.e., $2y \neq x$. After completing this iteration, the tester attempts to explore another path, such as the path leading to the if statement on line 7. To do this, the tester negates the path constraint, resulting in the constraint $2y = x$ and feeds this new constraint to an SMT solver, which solves it by assigning values to x and y, e.g., 2 and 1. The concolic tester re-executes the program and assigns the values `2` to `x` and `1` to `y`. Concrete execution reaches the if-statement on line 7, then takes the (non-existent) else-branch there, and the program terminates again without an error. Meanwhile, the symbolic execution gathered the path constraint $2y = x \wedge x <= y + 10$. The tester uses this path constraint to generate a new constraint by negating the last conjunct of the constraint, resulting in $2y = x \wedge x > y + 10$, and feeds the new constraint into a solver. The solver outputs e.g., the values 30 and 15 for `x` and `y`. A new iteration is started with these values, and concrete execution reaches line 8, at which point the tester can report the error encountered there. As no new branches were encountered by the tester during this last run, the tester concludes that it has explored all feasible program paths and concolic testing terminates. In general, for realistic programs with a large or even infinite number of possible program paths, concolic testing is terminated either when its given time budget is exceeded, or when the desired code coverage level is reached. Figure 1 also shows the symbolic tree that can be constructed from the path constraints.

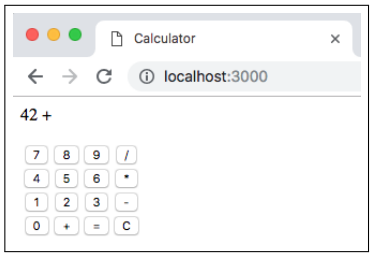

Fig. 2. The Calculator motivating example.

III. Motivating Example

In this section, we introduce a simple, full-stack web application that demonstrates how performing concolic testing in a full-stack context enables more accurate reporting of program errors than concolic testing that is performed on a per-process basis. Figure 2 depicts the client side of a simple online calculator where users may enter simple arithmetic expressions of the form *n1 op n2*, where *n1* and *n2* are numbers, and *op* is an arithmetic operator. When the user presses the button labelled =, the expression is sent to the server where its result is calculated. Afterwards, the server sends back this result to the client, which in turn shows the number to the user.

Part of the source code for the client side is listed in Figure 3. The client connects with the server by creating a new `WebSocket` (line 2). Afterwards, it registers separate event handlers for a mouse click on each individual button. Importantly, the event handler for the button labelled = calls the `compute` function (line 13). The client represents the arithmetic expression as an object `input` (line 16) containing three fields `left`, `op`, and `right`. `compute` checks whether the expression that was entered is a valid arithmetic expression and, if the user intents to perform a division, that the right operand is not zero. If either check fails, the function shows an appropriate error message to the user (line 19 and line 21). Otherwise the function sends the input via the web-socket over to the server (line 24). Finally, the client registers a callback for messages from the server (line 27). The server sends these messages to the client whenever it has completed a calculation. The `result` parameter of this callback represents the result of this calculation. Upon receiving such a message, the client simply shows the result to the user (line 29).

Part of the server side code for this example is listed in Figure 4. The code creates a `WebSocket` server instance (line 2), and makes this instance listen to incoming connections from clients (line 3). When a new client connects, the corresponding callback is triggered (lines 3-22) with the socket through which the client is connected to the server passed as an argument to the callback. The server registers a callback (line 5) on the socket to listen for messages coming from the client that contain the expression to compute. When the client sends such a message, the server retrieves the left and right operand, as well as the operator (line 7). The result is computed from these three elements (lines 9-20), and sent back to the client via the web-socket (line 21). Importantly, the server throws an error when it detects a division by zero (line 15) or when it does not recognize the operator to be applied (line 19).

```
1  // Connect with the server via a WebSocket
2  var ws = new WebSocket('ws://localhost:3000');
3  document.getElementById("0").addEventListener("click",
4    function (e) { // User clicked the button labelled '0'
5      clickDigit(0); // The clickDigit function is elided
6  })
7  document.getElementById("+").addEventListener("click",
8    function (e) { // User clicked the button labelled '+'
9      clickOperator("+");//The clickOperator function is elided
10 })
11 document.getElementById("=").addEventListener("click",
12   function (e) { // User clicked the button labelled '='
13     compute();
14 })
15 ... // Register event handlers for other buttons
16 var input = {left:0, op:"", right:0}; // Arithmetical exp
17 function compute() {
18   if (! isValidExpression(input)) {
19     resultElement.innerHTML = "Expression is invalid";
20   } else if (input.op === "/" && input.op === 0) {
21     resultElement.innerHTML = "Cannot divide by zero";
22   } else {
23     // Send the expression to the server
24     ws.send(input);
25   }
26 }
27 ws.onmessage = function(result) {
28   // Receive computation result from server
29   resultElement.innerHTML = result; // Show result to user
30 }
```

Fig. 3. Part of the client side code for the Calculator example.

```
1  // ... Setting up the server
2  var ws = new require('ws').Server({port: 3000});
3  ws.on("connection", function(socket) {
4    // A new client has connected
5    ws.onmessage = function(msg) {
6      // Receive input from client
7      var left = msg.left, op = msg.op, right = msg.right;
8      var result;
9      switch (op) {
10     case "+": result = left + right; break;
11     case "-": result = left - right; break;
12     case "*": result = left * right; break;
13     case "/": result =
14       if (right === 0) {
15         throw new Error("Dividing by zero");
16       }
17       result = left / right; break;
18     default:
19       throw new Error("Unknown operator");
20     }
21   socket.send(result); // Send the result back to the client
22 })
```

Fig. 4. Part of the server side code for the Calculator example.

A. Testing the Calculator

Traditional concolic testing would test both sides of the application in isolation from each other. A traditional tester examining the client should be able to exercise the event handlers registered for all of the buttons, and even the callback for receiving server messages (lines 27-30). This callback could be exercised either by mocking the server and generating messages containing random result values, or by actually requiring the testing setup to run a server besides the client process under test. In either case, a traditional concolic tester should be able to achieve 100% line coverage for the client.

When testing the server side, the tester could again opt to exercise the `compute` callback by mocking these messages. As the server is being tested in isolation from the client, the tester does not have any information on the contents of the message, and can therefore only assume that the message may contain any operand and operator. This would lead the tester to falsely conclude that the errors on lines 15 and 19 are feasible when in fact they can never occur in practice, as the client checks the appropriate condition. Just as with the client, a traditional concolic tester should hence be able to achieve 100% line coverage, but, importantly, it will also report these two errors even though these discoveries are actually false positives. The problem here is that the tester is not aware of the restrictions imposed upon the client's message by the conditions checked by the client on lines 18 and 20.

IV. Overview

In this section, we provide an overview of StackFul and detail how StackFul considers the execution of both sides of the web applications. StackFul is implemented in the Aran-Remote dynamic analysis platform [6]. This platform is specifically geared towards developing and applying dynamic analyses of distributed JavaScript processes. Aran-Remote instruments each of the processes in a distributed application with both generic Aran-Remote code, as well as instrumentation code that is specific to the analysis, in our case the concolic testing process, that is being applied. At run time, Aran-Remote spawns a central analysis process that regulates the execution of the application.

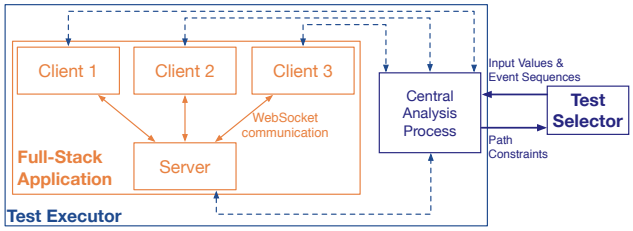

Fig. 5. The architecture of StackFul.

The architecture of StackFul is depicted in Figure 5. Components that are specific to the application under test are shown in orange, while components that belong to StackFul are shown in blue. The full-stack application under test is divided in a server process and one or more client processes. Both kinds of processes are instrumented by Aran-Remote, which applies both generic Aran-Remote instrumentation, e.g., to enable the resulting processes to communicate with the central analysis process, and concolic testing-specific instrumentation, e.g., to perform symbolic execution simultaneously alongside the concrete execution. The instrumented server code is executed as a regular Node.js process, while the instrumented client code is loaded by a browser, such as Firefox or Chrome. The client and server processes continuously communicate with each other by emitting Socket.IO messages.

A. Symbolic Execution

For the sake of brevity, we omit the details of how the code resulting from the instrumentation performs the symbolic execution. Conceptually, the generated code wraps every computed value in the application in a tuple containing the *concrete* value and a corresponding representation of

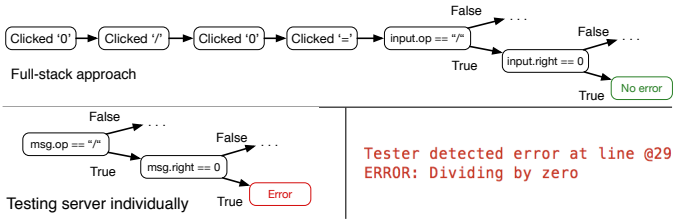

Fig. 6. A path leading to a division-by-zero warning on the client (top), and a path leading to a division-by-zero error on the server side (bottom left), with the superfluous error that is reported (bottom right).

the *symbolic* value. In the Calculator application, when the client sends the `input` object over to the server, StackFul also wraps this object's constituent fields. When the server extracts the fields from `input`, the server's instrumented code therefore has access to the symbolic representation of the input, and can use these symbolic values for the remainder of the symbolic execution of the application. As an example, suppose that an uninstrumented client would transmit the object `{left:42, op:"+", right:5}` , then the instrumented client would actually transmit:

```
{ left: { conc:42, symb:SymInt(42)},
  op: {conc:"+",symb:SymString("+")},
  right: {conc:5,symb:SymInt(5)} .
```

The server would then respond with a tuple of the form:

```
{conc: 47, symb: SymArithExp(SymInt(42), SymOp("+"),
SymInt(5)}
```

A concolic tester that lacks a holistic overview of both sides of the Calculator application would not be able to string together the symbolic values of both the client and the server. For example, when testing the client program, the tester would be incapable of symbolically representing the server's `result` value as the result of symbolically computing the result of the `left` and `right` operand. The only feasibly way of symbolically representing the server's result, would be as a generic symbolic input parameter. Figure 6 shows how StackFul handles the Calculator example. When running StackFul with the holistic approach, it finds a path that leads to a division-by-zero operation (e.g., when the user presses the `0`, `/`, `0`, and = buttons), but this only leads to the *warning* that is printed by the client (line 21), instead of an actual error. In the bottom half of the figure, StackFul does not use the holistic approach but instead tests the server individually: it reports a superfluous division-by-zero error on the server side.

B. Concolic Tester

StackFul consists of the *test executor*, itself composed of the client, server, and analysis processes, and the *test selector*. The test selector maintains the symbolic execution tree and suggests new program paths to explore. To this end, the selector suggests concrete values for the symbolic input parameters encountered in the path constraints, as well as sequences of events to play out by the client and server processes. The test executor is responsible for actually executing the application and following the prescribed path. To build up a holistic overview of the application's execution, the client and server processes communicate synchronously with the central analysis process (depicted as dotted lines in Figure 5). Specifically, this communication is tasked with the goals of *recording program behaviour* and *enforcing program paths*.

C. Recording Program Behaviour

The client and server processes continuously communicate to the analysis process all information that is relevant for determining which program paths are available. This includes information obtained via the symbolic execution, such as symbolic condition predicates that a re encountered. However, in event-driven programs such as web applications, a path constraint must contain not only the symbolic conditional predicates that were encountered, but also the sequence of events that were triggered, as e.g., clicking button A before clicking button B results in a different execution path than if the reverse were to happen. To discover all feasible program paths, StackFul therefore not only records the various conditional predicates that were encountered in each process, but also keeps track of the various event handlers that were triggered, such as the mouse click events for the buttons in the Calculator application. Furthermore, as StackFul works on full-stack applications, it builds up a holistic overview of the complete application's execution. It is not sufficient to record the various event handlers and symbolic conditions that are encountered for one particular process, but the tester must build up a *global* path constraint that incorporates conditions and events from *every* process of the application under test.

D. Program Path Enforcement

The central analysis process continuously communicates with the client and server processes to ensure that they follow the program path that was prescribed by the test selector for the current iteration. To enforce a particular program path, the analysis process transmits the appropriate concrete value for a symbolic input parameter whenever such a parameter is encountered by the client or server processes. Furthermore, the analysis regulates which events must be fired, and in which order. To ensure that no race condition can arise while executing the application, the analysis only fires an event if the event handler for the previously fired event has been terminated.

V. Conclusion and Future Work

We have proposed a novel approach for automatically testing full-stack JavaScript programs via concolic testing. This approach has the advantage of symbolically representing values more precisely, as values that are the result of handling user input in e.g., the client are precisely represented in the server, thus enabling the tester to filter out more superfluous errors. A prototype of this approach is implemented in StackFul. In the future, we aim to first extend StackFul with support for string operations, and having StackFul's test selector employ search strategies for selecting program paths that more closely follow the state of the art [7] [8] [9]. Afterwards, we can commence a proper evaluation of StackFul to consider the advantages and disadvantages of this holistic approach for real-world full-stack JavaScript applications.

REFERENCES

[1] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[2] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, June 2005.

[3] Ting Chen, Xiao-song Zhang, Shi-ze Guo, Hong-yuan Li, and Yue Wu. State of the art: Dynamic symbolic execution for automated test generation. *Future Generation Computer Systems*, 29(7):1758–1773, 2013.

[4] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), 2018.

[5] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, February 2013.

[6] Laurent Christophe, Coen De Roover, Elisa Gonzalez Boix, and Wolfgang De Meuter. Orchestrating dynamic analyses of distributed processes for full-stack javascript programs. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2018, Boston, MA, USA, November 5-6, 2018*, pages 107–118, 2018.

[7] Guodong Li, Esben Andreasen, and Indradeep Ghosh. Symjs: automatic symbolic testing of javascript web applications. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 449–459, 2014.

[8] Hideo Tanida, Tadahiro Uehara, Guodong Li, and Indradeep Ghosh. Automatic unit test generation and execution for javascript program through symbolic execution. In *The 9th International Conference on Software Engineering Advances (ICSEA2014)*, 2014.

[9] Koushik Sen and Gul Agha. Automated systematic testing of open distributed programs. In *Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, pages 339–356, 2006.